Lightweight Methods for Effective Verification of Software Product Lines with Off-the-Shelf Tools

Alexandru F. Iosif-Lazăr

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Supervisor: Andrzej Wąsowski

IT University of Copenhagen
September 2016

Abstract

Certification is the process of assessing the quality of a product and whether it meets a set of requirements and adheres to functional and safety standards. In some situations, certification is required legally in order to provide a guarantee for human safety and to make the product available on the market. The certification process relies on objective evidence of quality, which is produced by using qualified and state-of-the-art tools and verification and validation techniques.

Software product line engineering distributes costs among similar products that are developed simultaneously. However, product line certification faces major challenges, partly due to the lack of qualified, state-of-the-art tools to support the development process in the presence of variability. In practice, there are cases in which it is too costly or complicated to qualify the support tools; in other cases state-of-the-art tools exist, but do not handle variability.

In this thesis I identify three key challenges to product line certification. I address them by proposing effective, lightweight methods that employ off-the-shelf tools for supporting and verifying software product line development. All the proposed methods are generalizable to a large part of existing tools, they are formally specified and accompanied by proof-of-concept implementations.

The first challenge is that of qualifying variant derivation tools. Variant derivation is an essential part of software product line development. Any error that occurs during variant derivation can be propagated into the final product. Thus for safety critical product lines it is crucial that the derivation tool is trustworthy (qualified). The derivation algorithm depends on the modeling language in which the variability is expressed so proving that a derivation tool is correct requires a formal specification. However, many variability modeling languages lack a complete formal specification and the derivation tools are implemented ad hoc. To produce evidence of correctness for the product derivation I propose a lightweight verification technique based on translation validation which uses formal specification written in a core variability modeling language.

The second challenge is that of validating product line reengineering projects that involve complex source code transformations. To facilitate product (re)certification, the reengineering transformation must preserve certain qualitative properties such as code structure and semantics. This is a difficult task due to the complexity of the transformation and because certain properties, such as semantic equivalence, are generally difficult to verify. I present a number of lessons learned from a code modernization project that shows how, under certain conditions, it is feasible to design and implement a non-trivial rewriting transformation, such that it is highly efficient in practice. The project also contains a validation phase which identifies and analyzes a number of transformation bugs that could not be detected with regular unit testing.

The third challenge is that of using state-of-the-art quality assurance tools to verify product families. Most of the analysis tools that can be used off-the-shelf cannot handle variability and cannot be used to verify product families as a whole. The two alternatives are either to verify each product individually (a brute force approach) or to develop lifted analysis tools that can handle variability, both approaches being costly and time-consuming. To address this issue, I investigate the possibility of enabling the use of off-the-shelf analysis tools by rewriting compile-time variability into run-time variability. The technique is evaluated by attempting to use the analysis tools to identify bugs and vulnerabilities both in derived product variants and in reconfigured code.

This thesis provides an introduction and a background to software product line certification in Chapters 1 and 2. In Chapter 3 I formulate distinct problems that contribute to the three challenges, as well as hypotheses that address each problem. Chapters 5 through 7 describe how the hypotheses are tested by designing and implementing lightweight verification techniques for each challenge. Finally, Chapter 8 discusses the degree to which the proposed solutions have covered the initial problems and presents perspectives for future work.

Resumé

Certificering er den proces der omhandler kvalitetssikring af et produkt, om den følger en bestemt mængde krav og om den overholder funktionelle og sikkerhedsmæssige standarder. I nogle tilfælde er certificering lovmæssigt påkrævet for at sikre en vis garanti omkring personsikkerhed og for at indføre produktet på markedet. Certificeringsprocessen afhænger af objektive beviser omkring kvalitet, der produceres ved brug af kvalificeret moderne værktøjer og verifikations- og valideringsteknikker.

Udvikling af software produktlinjer sørger for en spredning af omkostninger når flere lignende produkter udvikles på samme tid. Der mødes dog stadig store udfordringer ved certificering af disse produktlinjer, til dels på grund af manglen på kvalificeret moderne værktøjer til at understøtte udviklingsprocessen når der er variabilitet tilstede. I praksis, er der tilfælde hvor det er for omkostningsfuldt eller kompliceret at kvalificere de understøttende værktøjer, og de moderne værktøjer der allerede er kvalificeret understøtter ikke håndtering af variabilitet.

I denne afhandling finder jeg frem til tre afgørende udfordringer ved certificering af produktlinjer. Jeg håndterer dem ved at foreslå effektive men simple metoder der benytter sig af lettilgængelige værktøjer til at understøtte og verificere udviklingen af software produktlinjer. Alle de foreslået metoder kan generaliseres til benyttelse af en stor del af de allerede eksisterende værktøjer, de er formelt specificeret og der medfølger prototypiske implementeringer af dem.

Første udfordring omhandler kvalificering af værktøjer til udledning af varianter, som er en essentiel del af udviklingen af software produktlinje. Enhver fejl der forekommer under udledningen af varianter kan brede sig ud til det endelige produkt. Derfor er det afgørende for sikkerhedskritiske produktlinjer at udledningsværktøjet er pålideligt (kvalificeret). Udledningsalgoritmen afhænger af modelleringssproget som variabiliteten udtrykkes i, og der kræves derfor en formel specifikation for at bevise at udledningsværktøjet er korrekt. Imidlertid, mangler mange variabilitetsmodelleringssprog en formel specifikation, og derfor er mange af de eksisterende værktøjer implementeret ad-hoc. For at bevise korrektheden af produktudledningen, foreslår jeg derfor en simpel verifikationsteknik baseret på *translation validation* (overs. oversættelsesvalidering) som benytter sig af en formel specifikation beskrevet i et kerne variabilitetsmodelleringssprog.

Anden udfordring handler om validering af produktlinje omstruktureringsprojekter, der indholder komplekse transformationer af kildekode. For at understøtte produkt (gen)certificeringen, skal omstruktureringstransformationen bibeholde visse kvalitative egenskaber såsom kodestruktur og semantik. Dette er en svær opgave på grund af kompleksiteten af transformationen, og på grund af den overordnet besværlighed i at verificere semantisk ækvivalens af programmer. Jeg præsenterer en række erfaringer fra en kodemoderniseringsprojekt der viser hvordan, under bestemte forhold, det er muligt at designe og implementere en ikke-triviel omskrivningstransformation der fungerer effektivt i praksis. Projektet indeholder også en valideringsfase som finder frem til og analyserer en række bugs i transformationen, der ikke var fundet ved brug af almindelige unit tests.

Tredje udfordring er brugen af moderne kvalitetssikringsværktøjer til at verificere produktfamilier. De fleste analyseværktøjer der er lettilgængelige kan ikke håndtere variabilitet og kan derfor ikke benyttes til at verificere produktfamilier som en helhed. De to andre alternativer er at enten verificere hvert enkelt produkt enkeltvis (en bruteforce metode), eller at udvikle specialiseret analyseværktøjer der kan håndtere variabilitet, hvor begge disse metoder er omkostningsfulde og tidskrævende. For at håndtere dette problem, undersøger jeg muligheden for at tillade brugen af lettilgængelige analyseværktøjer ved at omskrive oversættelsestidsvariabilitet til køretidsvariabilitet. Teknikken evalueres ved at benytte analyseværktøjerne til a finde frem til bugs og sårbarhedder, både i udledte produkt varianter og i den omskrevet kode.

Denne afhandling giver en introduktion til software produktlinjer i Kapitlerne 1 og 2. I Kapitel 3, beskriver jeg særskilte problemer der bidrager til de tre udfordringer, således også hypoteser der griber hvert problem an. Kapitlerne 5 til 7 beskriver hvordan hver hypotese er afprøvet ved at designe og implementere simple verifikationsteknikker for hver udfordring. Endeligt, diskuterer Kapitel 8 den grad de foreslået løsninger har dækket de umiddelbare problemer og præsenterer perspektiver til fremtidigt arbejde.

To Lavinia and Casandra

Contents

Acknowledgments

There are many people that have made this thesis possible through their support and patience.

I would like to thank my supervisor, Andrzej Wąsowski, who was always ready to share his knowledge and time. From him I have learned a great deal about software engineering, about doing research and about writing papers. It was through his guidance that my work has converged into a single thesis. I would also like to thank him for his patience and understanding when it came to balancing my research with my family time and a newborn child.

A great appreciation goes to the members of the ITU SQUARE group, many of whom are co-authors in my papers. Claus Brabrand, Ahmad Salim Al-Sibahi, Aleksandar Dimovski, Iago Abal, Jean Melo and Ștefan Stănciulescu have always been a source of good advice and a lot of fun. Their extensive knowledge in key areas have helped my research move forward.

I would also like to thank my other co-authors: Ina Schaefer, Juha Erik Savolainen and Krzysztof Sierszecki and the colleagues from my stay abroad at the Imperial College London: Cristian Cadar, Luís Pina and Tomasz Kuchta for their valuable input.

A special thanks goes to Ahmad and Luís for the many hours of philosophical discussions. In hindsight they did not relate to my research and were mainly a distraction, but an awesomely fun distraction, nonetheless.

I am very grateful to my Master thesis supervisor, Joe Kiniry, who, learning that I wanted to get a PhD, has given me my first insights into academic research and pushed me to write a quality thesis.

Finally, I would like to warmly thank my wife, Lavinia, and my daughter, Casandra, for their constant encouragement and support through all the deadlines and late working evenings and for reminding me that every now and then I should sleep and enjoy life.

Chapter 1

Introduction

Modern engineering has generated a wide range of smart products. From the mundane smart phone and smart TV to highly specialized systems such as intelligent cars, industrial robots and medical apparatuses, we have surrounded ourselves with complex devices that are both beneficial and potentially hazardous. On the one hand these devices automate a great deal of processes and decisions, reducing the chance of human error: an intelligent car can slow down automatically if it detects an imminent collision. On the other hand their increasing complexity can lead to design and implementation errors: a malfunctioning industrial robot can produce a lot of damage and even endanger human lives. To increase the customer's trust, manufacturers can certify their products by having an external auditor inspect the product and its development process and then issue a quality certificate. Certifying a product provides an assurance that it meets specific requirements, e.g. functioning as intended and adhering to safety standards. In some situations adhering to a certain standard is even required by law.

Product line engineering is a systematic approach to developing multiple product variants simultaneously. When a certain component can be reused in multiple product variants, then the cost of designing, implementing and testing the component is divided among all the variants. This approach increases product quality and reduces costs and time-to-market. However, product certificates are not straightforward reusable and there are serious challenges to dividing the certification costs among similar product variants. In software engineering these challenges are noticeable especially in the case of software product lines of embedded systems. My work is addressing some of these challenges by proposing lightweight solutions with off-the-shelf tools.

Smart devices (products running embedded software) have an ever growing range of applications from organizing our daily schedules (i.e. smart phones and tablets) to flying us across the globe (i.e. airplanes on autopilot). As our needs differ from one situation to another, there is a constant endeavor to attune these products to each specific user. We may decide on which smart phone to buy by choosing a combination of computing power, video camera and applications that best suits our needs. When we buy our car we go through a step-by-step process of deciding whether we want gas, diesel, electric or hybrid, front-wheels or 4-wheel drive, body-style, color and other options. Highly configurable products are rooted in the industrial revolution. Mass production of individual and interchangeable parts that would be combined into a product through an assembly line has been used throughout all manufacturing industries for the past two and a half centuries. The process has many benefits such as reducing costs and increasing product quality, but it has also allowed for reusing the same parts in different products in order to obtain a large variety of product configurations. In fact, Flores et al. [1] report a vehicle product line at General

Motors, consisting of over 60 models under seven brands and divisions. GM vehicles comprise some 300 engineered subsystems in which the variability measures to a few thousand features. The number of variants that GM actually produces is in the low tens of thousands.

The great variety of configurations that exist for a physical product is often reflected in the embedded software that runs it. As the number of configurations grows, so does the development cost and product complexity, making it difficult and highly inefficient to maintain all the product variants. *Software product line engineering* is a systematic approach to developing multiple product variants simultaneously, by building them from a common set of core assets while managing variability. The software is often developed modularly, with each module corresponding to a characteristic of the physical product. Modules can then be composed and interchanged to match specific configurations. Among the benefits of software product lines there are: overall increased quality of products, reduced costs, reduced time-to-market etc.

Human safety is a critical concern when any new product is engineered. A malfunctioning road vehicle can lead to an accident; a medical apparatus that behaves in an unexpected way can harm its operator or the patient; even a smart phone can explode and injure its user under certain conditions. In order to reduce the potential risks, the product developer must follow certain practices such as writing a formal specification of the product, using qualified tools, following certified processes, performing rigorous testing and adhering to safety standards.

The certification requirements are often derived from standards that define basic product functionality as well as guidelines for assessing quality and safety. These standards can be defined by standardization bodies, industrial consortia or scientific associations. Adhering to standards is generally optional and manufacturers can choose to do it to increase the customer's trust in the product. However, some standards can become a legal requirement for some devices in order to be allowed on the market (e.g. automobiles on the European market must adhere to the *ISO 26262 Road vehicles — Functional safety* standard). In each industry there are multiple standards that can be applied. Some are more generic and have a wider application; others are mode domain-specific. A lot of the domain-specific safety standards (over a hundred in total) are derived from *IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE, or E/E/PES)* which is the most basic and the broadest safety standard.

In the case of smart devices, the embedded software must be certified just as well as the physical product. Software certification is a process based on objective evidence for compliance with requirements. This objective evidence can be established by documentation review, audit or testing. The documentation typically reviewed for certification based on the standards *IEC 61508* and *ISO 13849*[1] includes:

- **verification and validation** plan;
- **software architecture and design specification** (structured or semi-formal);
- documentation of the software **tool qualification**;
- **coding standards**;
- **software criticality analysis**;
- graphical explanation and **source code review** report;
- **system integration test** report;

The list is not exhaustive, but it shows that the audit covers all development phases from planning and design to implementation, testing and validation. It also highlights that one of the certification

[1] Safety of machinery – Safety-related parts of control systems

requirements is the use of qualified tools. Most of the software development activities—such as system and architecture design, code editing, static analysis, compilation and testing—are supported by other software tools. These are generally known as *off-line tools*, meaning that they supports a phase of the software development lifecycle, but cannot directly influence the safety-related system during its run time. Depending on whether or not they can introduce errors in the final product, the off-line tools must be qualified for the tasks that they are fulfilling. Tool qualification means—similarly to product certification—an evaluation of objective evidence that the tool behaves according to its specification.

However, tool verification and qualification is not a trivial task and it gives rise to three major challenges for software product line certification.

Challenge 1: Qualifying product variant derivation tools.

Software product line engineering employs variability modeling languages to model the commonalities and differences that exist among the product variants. Once the variability models are in place, a process called variant derivation is used to extract the desired variants based on specific configurations of features. A trustworthy variant derivation tool is essential to ensuring the quality of the products. To achieve this, two requirements must be met. The first is verifying that the variability model does not allow configurations that may introduce variability bugs due to undesired feature interactions. The second requirement is verifying that the product variant derivation tool implements the derivation algorithm correctly. While there is a need for good verification methods for both validating variability models and tools, most of the research is focused on the models. Qualifying the tools that implement variability modeling and variant derivation is one of the challenges of software product line certification, partly because the tools employ complex algorithms and depend on external libraries which makes it impossible to formally prove their correctness.

Challenge 2: Trustworthy reengineering of software product lines.

Another situation where tool quality assurance is required is product line reengineering. This encompasses multiple scenarios:

- when a set of product clones are analyzed with the purpose of identifying commonalities and variability and their respective requirements, specification, models and code are merged into a single product line we want to assure that the merging of the variants preserves their quality properties;
- when a legacy product line must be updated to a new programming language in order to increase maintainability and efficiency we want to preserve the behavior of the legacy code, while optimizing it with the capabilities of the new language;
- when a successful product line must be migrated to a new platform in order to expand on the market, we want to maintain the essential functionality of the products while integrating them into the new platform.

Code reengineering implies that the code is changed in some form. From the perspective of certification every new variant or changed system needs a new safety assessment/certification. However, this does not mean the whole certification process has to be gone through again. If the baseline products or product lines were previously certified, then there is objective evidence of their quality, which can still be useful after the reengineering phase. The condition is that the quality of the baseline system is carried over to the reengineered product line.

Challenge 3: Lightweight tools for quality assurance in the presence of variability.

Both the product derivation tools and the reengineering tools have two main functions: (1) transforming the code to produce a variant/update the product line and (2) carrying over the objective evidence for the quality of the variant/product line. Much of this evidence exists in the form of documentation and quality assurance (QA) artifacts: structured or semi-formal specifications, verification and validation results, test results and source code review reports. Oftentimes these artifacts are produced with the support of analysis tools. However, most of the state-of-the-art QA tools do not handle variability so they cannot be integrated in the software product line development process. One approach is redeveloping the QA tools to handle variability, but this is generally difficult and costly. Alternatively, we need to be able to use existing single program QA tools in software product line engineering.

Summary. These three challenges to software product line certification are a result of two main factors. First, variability management is a difficult problem and it reflects onto every verification and analysis problem. Many state-of-the-art tools do not handle variability and lifting them to product line level is a costly and time-consuming task. Second, the great variety of tools makes it difficult to develop a cost-effective generic qualification technique. For example, most variant derivation tools are implemented ad-hoc, based on various variability modeling languages which means that each tool would have to be qualified individually. These two factors are deterring tool qualification.

In my research I investigate how off-the-shelf tools for variant derivation, product line reengineering and quality assurance are used in practice. For each challenge I propose a lightweight technique for using these tools in software product line development with increased confidence in their results. In each case, the effort of applying the technique is smaller than implementing a fully certified tool.

1.1 Goal and Objectives

Software product line development suffers from the lack of state-of-the-art, qualified support tools. In practice there is a wide range of tools that either can handle variability, but are not qualified or they are state-of-the-art, but do not handle variability. My goal is to facilitate certifiable product line development by creating lightweight techniques that employ off-the-shelf tools.

In addressing each of the three challenges, I was guided by a set of objectives.

1. Whether the task is variant derivation, code reengineering or some form of verification, any extra tool development has to be minimal; instead I must attempt to use existing off-the-shelf support tools as much as possible.
2. The technique should be generalizable to a large part of the existing tools and practical scenarios, this means that it should be implementation agnostic with regard to the support tool.
3. The technique itself should be lightweight and it should not reduce the efficiency or effectiveness of the support tools below a reasonable operational level.
4. The technique should be backed by a formal specification.
5. The technique should be demonstrated by a proof-of-concept.

1.2 Papers

This dissertation is based on the following papers.

Paper A: Alexandru F. Iosif-Lazăr and Andrzej Wąsowski. **"Trustworthy variant derivation with translation validation for safety critical product lines".** In: *Journal of Logical and Algebraic Methods in Programming* vol. 85, nr. 6, (2016), pp. 1154–1176.

Paper B: Alexandru F. Iosif-Lazar, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wasowski. **"Experiences from Designing and Validating a Software Modernization Transformation (E)".** In: *Automated Software Engineering, ASE 2015, 30th IEEE/ACM International Conference on, Lincoln, NE, USA, November 9-13, 2015.* Ed. by Myra B. Cohen, Lars Grunske, and Michael Whalen. IEEE Computer Society, 2015, pp. 597–607.

Paper C: Alexandru F. Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. **"Effective Analysis of C Programs by Rewriting Variability".** In: *The Art, Science, and Engineering of Programming* vol. 1, nr. 1, (2017). arXiv preprint arXiv:1701.08114.

Paper A is a journal article which encompasses two other papers[5, 6] that I authored during my PhD.

1.3 Contributions

It is not possible to provide a complete, generic solution to all three challenges in the scope of a single thesis. However, it is possible to contribute methods and tools that help advancing them in some context. For each challenge I propose a solution aligned to the above objectives.

Contributions to challenge 1: Qualifying product variant derivation tools. The difficulty of qualifying product variant derivation tools comes from the wide range of variability modeling languages and the complexity of the implementation. In **Paper A** I bring the following contributions to this challenge:

- A verification approach based on translation validation: instead of proving that the implementation of the tool is correct, this approach validates each execution separately.
- A core language for separate variability modeling, Featherweight VML, which is used to formalize the derivation algorithm that is implemented by the concrete tool. Featherweight VML is as expressive and versatile as other existing variability modeling languages used in practice (i.e. CVL, OVM, Delta Models). This enables the translation validation of tools based on any language that can be abstracted to Featherweight VML.
- A formal specification of semantics for Featherweight VML, including a confluence result.
- A proof-of-concept implementation by embedding Featherweight VML into the semantic framework of the Coq theorem prover. This includes a formal, mechanically checkable proof of correctness for the embedding itself and an executable. The technique is demonstrated on an implementation of the Base-Variability-Resolution language.

Contributions to challenge 2: Trustworthy reengineering of software product lines. Source code reengineering is a process which often benefits from automated code rewriting transformations. These transformations are often implemented ad-hoc and are based on complex rule-based rewrite systems which, combined with the fact that the grammar of the input and output languages can be ambiguous, makes it impractical to prove them correct. In **Paper B** I bring the following contributions to this challenge:

- A non-trivial modernization transformation for a real project; it is applied on a sample of over 4000 imperative C++ functions that are transformed into a pure expression model. The transformation is implemented in the TXL transformation language, using an off-the-shelf C++ grammar.
- A validation technique which checks each function transformation and provides a proof of correctness, or a counter-example that is used to identify bugs in the transformation. The validation technique uses the KLEE symbolic execution tool to establish the semantic relation between the input and the output of the transformation. The technique is also agnostic to the actual implementation of the transformation, which makes it generally usable to validate any similar transformation.
- A formal specification of the technique, an analysis of some non-trivial bugs that were identified in the transformation and a collection of lessons learned from the entire process.

Contributions to challenge 3: Lightweight tools for quality assurance in the presence of variability. The verification and validation of code is supported by a wide range of tools: abstract interpreters, model checkers, symbolic executors etc. However, a lot of these tools do not handle variability and lifting them to product line level is infeasible. In **Paper C** I bring the following contributions to this challenge:

- A stand-alone variability-related transformation, which transforms a C program family into a single program by replacing compile-time variability with non-determinism. This opens the possibility of using any analysis tool that cannot otherwise handle variability.
- The correctness proof of the proposed transformation, which shows that the set of outcomes of the transformed program is equal to the union of sets of outcomes of variants from the input family.
- A prototype tool, C Reconfigurator, which implements the above variability-related transformation.
- A demonstration of the approach by using several off-the-shelf analysis tools.
- An evaluation of the effectiveness of our transformation-based approach for finding real variability bugs in large variability-intensive C software systems.

1.4 Outline

The remainder of the document is structured as follows. Chapter 2 presents the concepts and terminology used in the certification and software product line domains. In Chapter 3 I offer a technical view of the three challenges and my hypotheses for how they can be addressed. Chapter 4 gives an overview of state-of-the-art related work. The techniques and tools used to address each challenge are summarized in Chapters 5 through 7. I provide conclusions and introduce future work in Chapter 8. **Papers A**, **B** and **C** are collected at the end of the dissertation.

Chapter 2

Terminology

2.1 Product certification and tool qualification.

Definition 1. *Certification* is the provision by an independent body of written assurance (a certificate) that the product, service or system in question meets specific requirements.

A certificate is an attestation of quality which results from the certification process. The process must be carried out by an independent certification body, a third party corporation or institute that must be accredited (i.e. formally recognized) to operate according to international standards. Certificates can be issued for products (components, devices), processes, organizational systems (like management systems) or persons.

For products, the certification process usually includes inspecting and testing the products themselves, but also an inspection of the development process. One of the key requirements for certification is the use of qualified tools.

Definition 2. Tool *qualification* is defined as the process necessary to obtain certification credit, meaning the acceptance by the certification body that using a tool satisfies a certification requirement, for a software tool.

Although the terms of *certification* and *qualification* are very similar, in practice *certification* is used when referring to products and *qualification* when referring to tools.

2.2 Software product line engineering.

The most complete definition of a software product line was provided by Clements and Northrop [7]:

Definition 3. A *software product line* is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

Each product is a software-intensive system (e.g. an operating system, a database etc.). The product line must manage all the characteristics (called here *features*) that are used to identify any one product from the rest. The products must meet a particular market segment or mission and their differences are due to different functioning conditions. For example, an operating system must manage the computer hardware, but a different variant of the system must be built for each architecture. The products are developed from a common set of core assets (e.g. source code, system models etc.). The product development is done in a prescribed way, meaning that the product line must specify how the core assets can be composed, how to relate each feature to the core assets and which configurations of features are valid.

In practice a *software product line* is often called a *software product family*. However, in this thesis I adopt the following definition:

Definition 4. A *software product family* is the set of systems or products that can be derived from a product line. The products of a family are also called *variants*.

The main difference is that a product family refers to the products in their finalized form while a product line often includes the languages, models and techniques used to develop the products.

In the following I will define the main aspects of a software product line.

Definition 5. *Variability* is the degree to which the individual products of a product family differ amongst themselves.

Variability is specified in a product line at multiple levels. At a high, more abstract, level variability is specified as distinguishable characteristics of the products such as different functions and behavior. At a lower, more concrete, level variability is specified as variation points that identify which core assets can be combined into a final product.

Measuring variability depends on how the it is specified in the first place. For example, one could measure variability by counting the number of optional features in a product line or the number of valid configurations (i.e. how many product variants can actually be built). When looking at concrete variability (e.g. differences in the source code) one could count how many lines of code differ between two variants. This way a system where variations in a single line of code could generate tens of variants could still be considered having low variability; while a system with only three variants, but hundreds of different lines of code between them could be considered highly variable. Ultimately, an appropriate variability measurement technique must highlight the impact that all the differences between variants have over the design, development, testing and maintenance of the product line.

Abstract variability. Variability is specified at an abstract level in order to understand product differences using domain-specific concepts that all stakeholders can understand. For example operating system variants may differ in CPU architecture (x86, or x64, or ARM), in target platform (PC or mobile) etc. These concepts are usually seen as product features or as design decision that can define a product.

Definition 6. *Features and decisions* are representations of variability that describe product behavior and components in an implementation-agnostic manner.

Definition 7. *Feature/decision models* are an abstract, structured representation of the variability that manifests in a product family as a whole.

Feature/decision models use features/decisions to describe product functions and behavior as well as the relations between these. Selecting/enabling a feature or decision means that all the core assets that are associated with it will be part of the finalized product. A variability model also specifies the relations between the features/decisions such as composition (one feature can be composed of several others), implications (one feature implies or requires another one) and exclusion (one feature excludes another such that they cannot be part of the same product).

Definition 8. A *configuration* is a set of features/decisions that when selected/resolved uniquely describe a single product of the product family.

Configurations are used to describe a desired product variant before actually building it. Product configurations are constrained by the feature model as they must subscribe to all the relations between features.

Concrete variability. Variability is specified concretely over the core assets to define how they can be used to build products.

Definition 9. A *variation point* represents a possible choice between core assets.

Since the core assets can be any kind of development artifact, the variation points are similarly diverse (e.g. a line of source code, a fragment of a class diagram etc.)

Definition 10. A *variability realization* technique is a way in which one can implement a variation point.

Variability realization techniques are concrete methods of implementing variation points over core assets. They describe an algorithm by which the assets can be selected, replaced and combined to build a concrete product variant.

Variability models. Given all of the above definitions we can take a look at variability models as a whole.

Definition 11. A *variability model* is a model that represents all variability aspects of a product line: a feature/decision model that describes and constrains features/decisions and a variability realization technique that defines a set of variation points over the core assets.

A variability model can be used to identify valid product configurations and derive concrete product variants.

Definition 12. *Variant derivation* is the process of building a product variant by applying a variability realization technique over a set of core assets, based on a particular configuration conforming to a feature/decision model.

Through variant derivation all variability choices are resolved, the desired core assets are selected and assembled into a concrete product variant.

Chapter 3

Problem Definition

Product line certification requires objective evidence for the quality of the product line in general and that of the derived product variants in particular. In many cases, this evidence is produced by using state-of-the-art tools to verify and validate the correctness of the models and of the source code. Furthermore, the evidence of quality must be carried over by any automated process that changes the model/code, such as variant derivation or reengineering transformations. However, the support tools that are used in practice rarely produce this evidence. In my work I identify three major challenges to certifiable product line development that have proved to be difficult and costly to handle. I hypothesize that these challenges can also be handled effectively with lightweight techniques and off-the-shelf tools, while still maintaining a good balance between cost and effectiveness.

Software product line certification requires objective evidence for the quality of the development process, of the intermediary artifacts and of the finalized products. In practice, the lack of qualified and state-of-the-art tools makes is difficult to assess the quality of the products. In the following I will detail three major challenges to certifiable product line development.

3.1 Challenge 1: Qualifying product variant derivation tools.

Software product lines are developed from a common platform of core artifacts that are selected and assembled into product variants based on specific configurations (i.e. variant derivation). The differences that exist among the products are expressed at the artifact level through variation points and the set of valid configurations is constrained by a variability model.

Variant derivation is a process generally *assumed* to be correct in product line engineering, even though any error that occurs at this stage can propagate into the actual product. Qualifying variant derivation tools is essential to certifying the product line as a whole and any derived variant in particular. However, variant derivation tools are language processing programs for which few verification techniques exist. In this section I present three problems that increase the difficulty of qualifying product variant derivation tools.

The lack of a standard understanding of variability modeling. The first problem is the lack of a standard understanding of variability modeling which means that each variability modeling language and every variant derivation tool require a different qualification strategy.

Problem 1.1. *Due to the great variation among variability modeling languages, it is impossible to develop a standard specification of variability modeling and of the variant derivation algorithm.*

There is a wide range of variability modeling and product derivation tools that are used in practice: Atego Vantage[1], FeatureMapper[8], pure::variants[9], BigLever Software Gears[2], FeatureIDE[10] etc. There is also much variation among these tools. Pereira et al.[11] have found that out of 41 selected tools 63,4% were developed exclusively in the academic environment, while only 12,2% were developed exclusively in the industry, and the remaining 24,4% tools were developed in both academic and industrial environments. Some tools implement strictly a variability modeling language, while others are combining concepts from various languages in order to cover specific use cases.

The variability modeling languages themselves use different techniques for describing variability and specifying the variant derivation algorithm. Some use feature models, others use decision models. Some allow artifacts to be composed freely, others impose a partial order on the artifacts forcing them to compose incrementally. For variant derivation, some languages employ code preprocessing, while others use transformations, generation or composition. Some languages focus on handling variability in source code or models, others are orthogonal to the development process and can handle variability in any kind of artifact. Some languages use standard terminology, while others try to optimize handling variability by introducing new concepts and new terminology. Some examples are: the Common Variability Language[12], Delta Modeling[13], the Orthogonal Variability Model[14], TVL—a text-based variability language[15], Clafer[16] and the Koalish modeling language[17].

Some attempts of standardizing variability modeling are currently under development. The Common Variability Language has been submitted to the Object Management Group as a standardization language and there have been experiments for mapping other languages to it, i.e. Clafer offers some support for CVL notation. However the standardization problem is far from being resolved. This means that any attempt to formalize and reason about variability modeling must be done per language and is not generalizable to other languages. I illustrate this point with a comparison of three popular variability modeling languages. The comparison is presented in **PAPER A** as well as in Chapter 5.

Many of the variability modeling languages do not have complete formal specifications. The second problem is the lack of fully formal specifications for many variability modeling languages which leaves space for ambiguity of semantics and makes it impossible to prove their correctness.

Problem 1.2. *Not all variability modeling languages have a formal specification, making it impossible to prove the correctness of the variant derivation algorithm.*

In order to formally prove that a variability modeling language is implemented correctly (e.g. that there are no ambiguities in the syntax and semantics, that the variant derivation algorithm terminates and is confluent) it must be formally specified. While some languages are formally specified, others only provide a semi-formal specification. It is often that a language specification is written in a combination of predicate logic, visual descriptions and informal text. In these situations it is difficult to automate any formal reasoning about the language such as using theorem provers or model checkers.

[1] `http://www.atego.com/products/atego-vantage/`

[2] `http://www.biglever.com/solution/product.html`

The difficulty of proving variant derivation implementation to be correct. The third problem is the difficulty of formally proving that the variant derivation implementation actually matches the specification of the derivation algorithm due to the complexity or unavailability of the source code.

Problem 1.3. *The complexity of the variant derivation implementations, combined with the possible lack of a formal specification of the derivation algorithm, increases the difficulty of proving the correctness of the tools.*

Many of the variability modeling tools are implemented on third party platforms or are integrated in large development suites. For example Feature Mapper[8], FeatureIDE[10] and pure::variants[9] are implemented using Eclipse which means that they rely on the correctness of the underlying development environment. Atego Vantage[3] is a complex tool suite that includes various subsystems such as Atego Modeler, Atego Asset Library, Atego Process Director and Atego Check which increases the complexity of the tool and the possibility of errors occurring due to the interactions of the subsystems. Although these tools undergo rigorous testing and quality assurance checks before being launched in production, right now it is still impossible to provide a formal proof of correctness using state-of-the-art verification methods.

3.2 Challenge 2: Trustworthy reengineering of software product lines.

Software reengineering is normally used when a code base must be migrated to a different platform or updated to newer technologies. It is not employed as frequently as other software development processes, but it is a costly process that poses huge risks. When a project is updated to a newer platform a part of the available resources must be reallocated from regular development to the reengineering process. If the reengineering is gradual such that until the process completes the legacy code is obsolete and the new code is not yet complete then regular development may be paused altogether. Besides having to be performed as quickly as possible, the reengineering process also has to be trustworthy and preserve the properties and the quality of the source code and models.

The costs and lack of trust in reengineering transformations are making companies reluctant to update their legacy code and port their products to new platforms. In this section I present two problems with producing trustworthy automatic reengineering transformations.

Complex transformations are difficult and costly to implement. The first problem is that the transformation required might be very complex which depends largely on the complexity of the languages in which the input and the output are written. This can lead to increased costs and time for developing the transformation.

Problem 2.1. *Depending on the size and complexity of the programming languages in which the input and the output code is written, a code rewriting transformation may be very difficult and costly to implement, and it may even be impossible to reach completeness.*

Software reengineering is a process which often benefits from automated code rewriting transforma-

[3] `http://www.atego.com/products/atego-vantage/`

tions that migrate source code from an old platform to a new one. In order to facilitate recertification, these rewriting transformations must preserve certain properties such as code structure and semantics. In the case of software product lines, the modernization process must also account for the variability existent in code. This is both costly to implement and difficult to verify.

A common approach to automatic reengineering is to develop a rule-based transformation. A rule matches a pattern in the input code/model and provides another pattern to be written in the output code/model. The input and output patterns are written in the input, respectively output, languages. To understand how the languages can influence the complexity of the rules we must look at two factors.

The first factor is **completeness**. *Should the transformation handle any arbitrary program written in the input language?* If completeness is one of the requirements of the transformation, e.g. modernizing COBOL programs to C++, then all the syntactic constructs of COBOL must be considered as possible input patterns for the rules. Such a transformation would reach the size and complexity of a full-blown compiler. By contrast, a transformation that must only handle a subset of the input language is much cheaper to develop. For example generating an object-relational mapping (from an object oriented language to a relational database, e.g. from C# to T-SQL) only deals with the language subset that is used for declaring data types and structures. The transformation will only require rules that cover the actively used subset.

The second factor is the **similarity** between the input and the output languages. *Do the languages use the same concepts and are they similarly expressive?* For some transformations the conversion from one language to the other might be straightforward—for example the object-relational mapping associates tables with classes, attributes with fields and relations with references. Implementing such a transformation is almost trivial as the transformation rules only have to translate one small pattern into another. Other transformations may deal with concepts that are more difficult to translate (e.g. converting C pointers to Java references [18]). In some cases it might even be impossible to generalize—for example Terekhov and Verhoef [19] show that when translating COBOL to Visual Basic, "no single data type in Visual Basic can handle the fixed-length record structure in COBOL" so for "different contexts different solutions are necessary". This means that the transformation rules will use complex patterns that cover very specific contexts.

By looking at these factors it is clear that the input and output languages have a great influence over the complexity of the transformation.

Rule-based transformations are often impossible to prove correct. The second problem is the difficulty of producing a proof of correctness for a rule-based transformation which is largely influenced by the complexity of the input/output languages, but also by the practical aspects of implementing it.

Problem 2.2. *The practical aspects of implementing a rule-based transformation often make it impossible produce a proof of correctness of the implementation.*

The correctness of rule-based transformations entails that the transformation preserves a number of properties from the input to the output. For example a transformation that converts a relational database to object-oriented classes must preserve the data structure and types, while a transformation from a high level language to another must preserve the semantics of the program. To demonstrate that a transformation is correct is to prove that it is sound with respect to these properties. However, the soundness of a tool is dependent on the source code of the implementation, on the language in which it is implemented and on any third party code that is called.

Rule-based transformations are usually implemented in specialized transformation languages (e.g. TXL, ATL, ETL). These languages provide a complex framework for writing language grammars, parsing input code, traversing the abstract syntax tress and performing pattern matching, replacing input patterns with output patterns based on custom rules and, finally, writing the output. In order to prove that a particular transformation is correct, one must prove that each one of these subprograms is correct.

Another issue is the rule system itself. While it may be relatively easy to prove that a single rule is correct, when there are multiple interacting rules the problem grows exponentially. In many cases, the order in which the rules are applied is determined by the traversal algorithm and by the structure of the abstract syntax tree of the input which is usually arbitrary. This makes it very difficult to reason about the way rules are applied and about the soundness of the transformation as a whole.

3.3 Challenge 3: Lightweight tools for quality assurance in the presence of variability.

State-of-the-art analysis tools are not easily available for software product line development. As product certification requires objective evidence of quality, software engineering employs a wide range of quality assurance tools that can identify various problems in the source code. However, most of these tools can only handle single programs and cannot be used to verify program families. In this section I present the problem of using quality assurance tools in the presence of variability.

Problem 3. *Most of the state-of-the-art analysis tools cannot handle variability. Lifting an analysis tool to product line level is a costly task which is often performed after the single product analysis tool has reached a certain stage. This means that state-of-the-art analysis tools are not easily available for software product line development.*

Quality assurance tools can help identify bugs and vulnerabilities in software programs by performing various analyses on the source code and on the compiled code. *Type checking* is used to verify that the appropriate methods and functions are used based on their signatures and the data types. This gives certain assurances over the control and data flow of the program and it identifies some memory vulnerabilities. *Syntax checking* usually finds bugs by looking for specific syntactic patterns. *Static analysis* and *model checking* are used by building and validating abstractions of the program.

Many of such tools are state-of-the-art, but they are also highly specialized and are usually developed for specific programming languages. This makes them inapplicable in the development of certain product lines where the variability is specified in a different language than the source code. For example, the C/C++ product lines use the C Preprocessor to handle variability. Usually the C Preprocessor directives are interpreted before the compilation of the C/C++ code even begins, while the quality assurance tools are applied on the code after preprocessing. This means that traditionally there was little incentive for the analysis tools to ever handle C Preprocessor directives.

In the case of C/C++ software product lines, the C Preprocessor directive `#ifdef` is used to control variability through the conditional compilation of code, based on a set of user-selectable options called features. To use a quality assurance tool, usually means attributing values to the features, preprocessing the code to derive a particular product variant and then applying the tools on the variant code. As the number of features increases, the number of product variants increases exponentially so it becomes impractical to verify one variant at a time (also known as brute force verification). The goal of verifying product lines is to check the common code only once and, when an error or vulnerability is detected, to determine which are the product variants that it affects.

In recent years a number of analysis tools that can handle variability have been proposed. Examples of successful family-based analysis tools cover syntax checking [20, 21], type checking [22, 23], static analysis [24, 25], model checking [26, 27]. These tools are usually developed by lifting analysis techniques that have already been implemented in single product analysis tools. However, the lifting process is not straightforward. Instead of parsing just the C/C++ code, the C Preprocessor directives have to be parsed simultaneously and represented in the abstract syntax tree. Type checkers must handle the possibility of a data type or a function having different definitions in different product variants, while coexisting in the program code. Generally, updating an analysis tool to handle variability implies an almost complete redesign of the technique from the theoretical foundation to the actual implementation. This is a difficult and costly task. As a result, most of these tools are still experimental and do not yet have a stable release.

3.4 Contributions

The problems posed by the three challenges are not trivial. Extensive research is allocated to solving them in the fields of software product line engineering, variability management and product line certification. The key issue is the difficulty of finding a solution that works 100% of times:

- it is difficult to prove that a variant derivation tool will always produce correct variants;
- it is difficult to prove that a code reengineering transformation will always cover all the input patterns correctly, especially when there is variability in the source code;
- it is difficult to produce lifted versions—that would handle variability and maintain effectiveness—for all analysis tools.

However, I believe that in practice it is possible to strike a balance between efficiency and effectiveness. In the following, I provide my hypotheses for how each of the problems may be addressed.

For Challenge 1, qualifying product variant derivation tools, I propose the following hypotheses:

Hypothesis 1.1. *Although it is impossible to develop a standard specification of variability modeling, many of the variability modeling languages that are used in practice share sufficient core characteristics so that a core variability modeling language can be designed.*

Hypothesis 1.2. *A core variability modeling language can be used to develop a formal specification for a real variability modeling language. While the core language would not be powerful enough to capture the real language entirely, it would be sufficiently flexible to specify common concepts of popular variability modeling languages.*

Hypothesis 1.3. *While it is difficult to prove that a variant derivation tool will always produce correct variants, one can use a core specification of the variant derivation algorithm and translation validation to validate each execution and either formally produce evidence of correctness or identify any errors that might have occurred.*

For Challenge 2, trustworthy reengineering of software product lines, I propose the following hypotheses:

Hypothesis 2.1. *While it is difficult to design and implement complete code rewriting transformations, in practice there are situations where an incomplete transformation that focuses on a specific subset of the input language grammar is less costly to implement while it will be sufficiently effective.*

Hypothesis 2.2. *If a rule-based code rewriting transformation aims to preserve certain properties of the code, one can use a formal specification of the property and translation validation to produce a proof of correctness and identify any errors that might have occurred.*

For challenge 3, lightweight tools for quality assurance in the presence of variability, I propose two hypotheses. Most of the state-of-the-art analysis tools cannot handle variability because it is often specified in a different language than the language of the source code itself.

Hypothesis 3.1. *By rewriting the variability in the language of the source code, some of the analysis tools that do not normally handle variability will be applicable to the entire product line. Analyzing the reconfigured code with existing tools is a cost-efficient alternative to lifting the analysis tools to product line level.*

Hypothesis 3.2. *Although variability rewriting does not provide a guarantee of effectiveness with respect to analysis, some analysis tools will retain sufficient effectiveness on the reconfigured code so that they may be used in practice.*

In chapters 5, 6 and 7 I describe in more detail the research done in my PhD project to test the above hypotheses.

Chapter 4

Related work

This chapter collects a high-level summary of the related work with respect to each challenge. This is meant to provide a context for my solutions presented in chapters 5–7. In addition, each paper discusses the related work in detail with respect to each particular contribution.

4.1 Challenge 1: Qualifying product variant derivation tools.

Tool qualification. To understand the requirements for tool qualification I have looked for studies that link the qualification requirements to specific standards. Conrad et al. [28] present an overview of qualification approaches with regard to various standards (DO-178B, IEC 61508, ISO/DIS 26262, MISRA-C). They present several tool classification methods used by the aforementioned standards to decide the qualification methods.

For example, the standard IEC 61508 (Ed. 2.0) provides a classification of software offline tools (i.e. tools that supports a phase of the software development life cycle and cannot directly influence the safety-related embedded system during its run time):

T1: tools that cannot directly or indirectly contribute to the executable code of the system;

T2: tools supporting verification or test of the design or the executable code; errors in these tools can fail to reveal defects;

T3: tools generating outputs that can directly or indirectly contribute to the executable code of the system.

By this classification, variant derivation tools fall under the category T3, which require evidence that the tool conforms to its specification or manual either based on confidence from use or on application independent validation.

Asplund et al. [29] propose a method for qualifying software tools as parts of tool chains. They identify a number of safety goals such as the possibility to trace between artifacts at different steps of the process to ensure that they are consistent and complete in regard to each other. This is especially applicable to variant derivation where the input and the output must be consistent.

They also identify the safety goal of well defined data semantics. For variant derivation this means that a qualified tool requires a well defined language for modeling the variability and the base models.

The great variety of variability modeling approaches. In my work I investigate three popular variability modeling languages: CVL [12], Delta Modeling [13] and OVM [14] which are presented in detail in Chapter 5. Aside from these three, there are numerous other variability modeling languages and various approaches to formalizing the elements of a variability model.

To my knowledge, there are no comparative studies of variability modeling approaches with the purpose of identifying core concepts; such a study could form the basis for a generic formalization of variability modeling. Instead there are numerous studies that illustrate and classify the differences.

Istoan et al. [30] use a metamodel-based classification that splits variability modeling techniques into:

1. methods that use a single (unique) model to represent the SPL assets and the SPL variability:
   - base model annotations by means of extensions;
   - combining a general reusable variability meta-model with base meta-models;
2. methods that distinguish and keep separate the assets model from the variability model:
   - connecting feature diagrams to model fragments;
   - Orthogonal Variability Modeling (OVM);
   - ConIPF Variability Modelling Framework (COVAMOF);
   - decision model based approaches;
   - combining a common variability language (CVL) with different base modelling languages.

Many of the variability representations used for software product lines are derived from feature models [31] or decision models [32]. Czarnecki et al. [33] provide a comparison of features and decisions and show how they are adapted to various modeling techniques.

Approaches for variability realization. The mapping from features to development artifacts has been specified by using feature modules (feature-oriented programming [34]) to wrap the artifacts pertinent to each feature and applying module composition to derive new variants; alternatively [35, 36], by annotating the artifacts with presence conditions thus specifying when each artifact must be present in a product variant. Other approaches [37] involve model transformations where each feature can both remove and add artifacts to existing models. While some formalisms are richer than feature models (e.g. Koala [17] employs a topology of components that is not a tree and interfaces between components) , a lot of these formalisms provide a fixed representation of base models (e.g. component models) and do not relate to base models specified in customized domain-specific languages or to concrete implementation artifacts such as source code [36, 38]. All these approaches bring different advantages and challenges to the domain of variability modeling, usually compromising between expressive power and simplicity and which influences the possibility of validating actual derivation tools.

Formalizing the variability models. So far, most work on variability was dedicated to analyzing feature models [39, 40]. Recent work has provided valuable insight such as formalizing feature models represented in a textual language [15] or even providing full proofs in the PVS proof assistant [41]. However, the formalization is limited to feature models and do not touch on the subject of variant derivation. Czarnecki et al. [16] show how to represent the three layers of variability modeling within the single Clafer syntax. However no actual mapping to implementation artifacts is considered, just a Boolean abstraction of dependency. Such a formalization cannot directly be used as a specification of correctness for a variant derivation tool. Other works consider analyzing variability models as a whole, including checking for consistency (for instance [42–46]). All these methods assume correctness of the variant derivation implementation.

4.2 Challenge 2: Trustworthy reengineering of software product lines.

Reengineering transformations. In my work I present a collection of lessons learned from the design and implementation of an automated code modernization transformation. A similar automation-based modernization effort is presented in a joint project by Semantic Designs (SD)[1] and Boeing[2] [47] which uses Semantic Designs's commercially-available transformation and analysis tool DMS [48] to convert an old component-based C++ codebase to a standardized CORBA [49] architecture.

A series of papers [50–52] discuss a similar case study that aims to design and verify a model transformation for modernizing an existing collection of proprietary models such that they conform to the standardised AUTOSAR [53] format. The transformation [50] was initially encoded in a (non-Turing complete) subset of the ATL model transformation language [54] and then verified for structural properties [51]. The same verification effort was then repeated [52] more efficiently by symbolically executing a version of the transformation re-encoded in DSLTrans [55].

Verifying code rewriting transformations. Proof of program transformation correctness is often studied in conjunction with optimization. Parametrized equivalence checking (PEC) [56] uses a form of translation validation which tries to find a bisimulation between the control flow graphs of the original and optimized programs in a way that parameterizes over some program components (such as expressions, statements and declarations). Other work [57] specifies the optimisation rewrite rules as temporal logic formulas, and proves the correctness of the transformation manually. Both techniques lack tools that support them.

Currently, there exist various robust techniques for verifying preservation of properties by model transformations [58–61]. These techniques use (bounded) model finders and SMT solvers to verify the preservation of structural properties of model transformation rules. In contrast to this, I present a verification technique for checking behavioral equivalence between the input and output, which is significantly more complex to check than structural properties.

4.3 Challenge 3: Lightweight tools for quality assurance in the presence of variability.

Specifically designed variability-aware techniques. Various lifted techniques have been proposed which lift existing single-program verification techniques to work on the level of program families. This includes lifted syntax checking [20, 21], lifted type checking [22, 23], lifted static analysis [24, 25, 62], lifted model checking [26, 27], etc. TYPECHEF [20] and SUPERC [21] are variability-aware parsers, which can parse languages with preprocessor annotations. The results are ASTs with variability nodes. Several approaches have been proposed for type checking program families directly. In particular, lifted type checking for Featherweight Java was presented in [22], whereas variational lambda calculus was studied in [23]. Lifted model checking for verifying variability intensive systems has been introduced in [26]. Transition systems enriched with features are used for compact modelling of such systems, where

[1] `https://www.semanticdesigns.com`

[2] `http://www.boeing.com`

system parts that vary are annotated using features. SNIP, a specifically designed family-based model checker, is implemented for efficient verification of temporal properties of such systems. An approach for lifted software model checking using game semantics has been introduced in [27]. It verifies safety of `#ifdef`-based program families containing undefined components, which are compactly represented using symbolic game semantics models [63]. Brabrand et al. [24] and Midtgaard et al. [62] show how to lift any single-program dataflow analysis from the monotone framework to work on the level of program families. The obtained lifted dataflow analyses are much faster than ones based on the naive variant-by-variant approach that generates and analyzes all variants one by one. Another efficient implementation of lifted analysis formulated within the IFDS framework for inter-procedural distributive environments has been proposed in SPL$^{\text{LIFT}}$ [25]. In order to speed-up the lifted verification techniques, variability abstractions have been introduced in [64–66]. They tame the exponential blowup caused by the large number of features and variants in a program family. In this way, variability abstractions enable deliberate trading of precision for speed in the context of lifted (monotone) data-flow analysis [64] and lifted model checking [65, 66].

Lifting by simulation. Variability encoding [67] and configuration lifting [68] are based on generating a product-line *simulator* which simulates the behaviour of all products in the product line. Then, an existing off-the-shelf single-program analyzer is used to verify the generated product-line simulator, which represents a single program. The work in [67] defines variability encoding on the top of TypeChef parser for C and Java program families. They have applied the results of variability encoding to testing [69], model checking [70], and deductive verification [71].

Chapter 5

Challenge 1: Qualifying product variant derivation tools

5.1 Many variability modeling languages share similar core characteristics.

The first hypothesis is addressing the lack of a standard specification of variability modeling. While it acknowledges that there cannot be a single specification for all variability languages, it puts forward the idea that the core concepts of most variability languages are similar and that a common representation of these core concepts can be designed.

Hypothesis 1.1. *Although it is impossible to develop a standard specification of variability modeling, many of the variability modeling languages that are used in practice share sufficient core characteristics so that a core variability modeling language can be designed.*

To test this hypothesis I analyzed three of the more popular variability modeling languages aiming to gather requirements for a core variability modeling language. The purpose of this core language is not to become yet another variability modeling language, nor is it intended to become a standard. Instead it aims to provide a generic way of formalizing essential aspects of variability modeling such as feature models, configurations and variation points. A core language that is formally specified can form the basis of developing a generic method for validating the correctness of variant derivation tools.

A precondition to designing a core variability modeling language is that the real languages share enough characteristics. To determine this, I have compared CVL[12], Delta Modeling[13] and OVM[14], aiming to find similarities in the way these languages represent and execute variability models over system models.

5.1.1 Example.

In Fig. 5.1 I introduce an example of a product line architecture from which several product variants can be derived. This example will help illustrate the characteristics of Delta Modeling and CVL. To keep the example concise, the characteristics of OVM are only mentioned in the comparative analysis. In the following sections the same example will be used to describe Featherweight VML.

The system represents a safety critical monitoring device. A minimal product variant is composed of the deviceCpu, the sensor and one of the possible outputs. The deviceCpu receives raw input from the

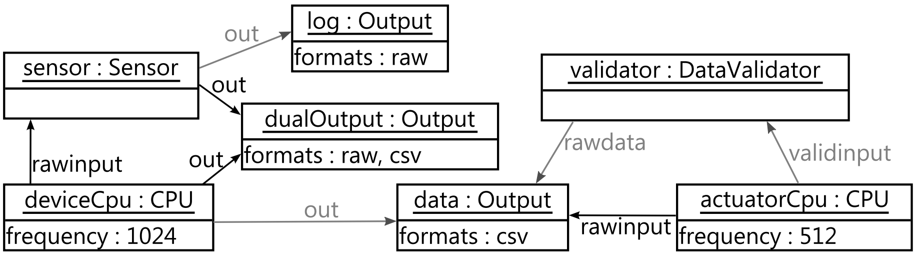

Figure 5.1: Example base model for a product line of devices

sensor and relays the data as comma-separated values (csv) to the dualOutput to which the sensor also sends the raw data.

Alternatively, sensor may output raw data directly to a log as represented by the gray link out. Yet another possibility is for the deviceCpu to send csv data to a data output which serves as input for an actuatorCpu. As an extra check, the actuatorCpu may compare the raw data with validated input from a validator.

```
features Dual, Log, Actuator,  Validator
configurations Dual ⊕(Log ∨(Actuator ∨ (Actuator ∧ Validator)))

core Dual {
    adds objects {deviceCpu, sensor, dualOutput}
    adds links {deviceCpu.rawinput -> sensor, deviceCpu.out -> dualOutput,
    sensor.out -> dualOutput}
}

delta DLog when Log {
    removes objects {dualOutput}
    removes links {deviceCpu.out -> dualOutput, sensor.out -> dualOutput}
    adds objects {log}
    adds links {sensor.out -> log}
}

delta DActuator when Actuator {
    removes objects {dualOutput}
    removes links {deviceCpu.out -> dualOutput, sensor.out -> dualOutput}
    adds objects {data, actuatorCpu}
    adds links {deviceCpu.out -> data, actuatorCpu.rawinput -> data}
}

delta DValidator when (Actuator ∧Validator) after DActuator {
    adds objects {validator}
    adds links {validator.rawdata -> data, actuatorCpu.validinput -> validator}
}
```

Figure 5.2: Delta model of the device product line.

Figure 5.2 shows how Delta Modeling can handle the variability in the above device architecture. It begins by specifying a list of **features** and a constraint for valid **configurations**. The **core** and **delta** modules add and remove objects and links. The **core** module adds the objects deviceCpu, sensor and dualOutput identified by the object name. It also adds the rawinput link from deviceCpu to sensor and the two out links from deviceCpu and sensor to dualOutput. It is executed when the Dual feature is selected.

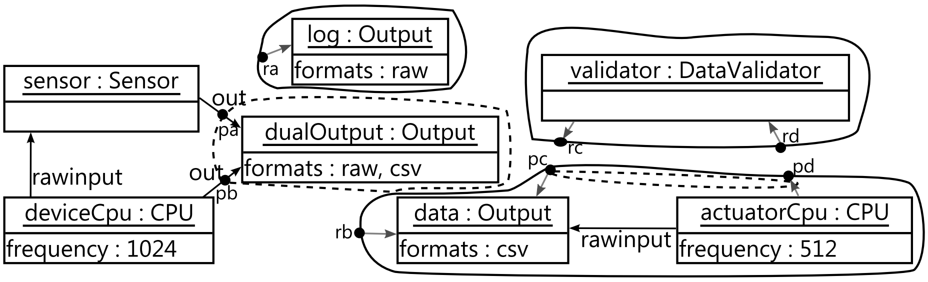

Figure 5.3: CVL model of the device product line

The rest of the **delta** modules add and remove objects and links to express the alternative product variants. The **delta** DValidator is executed if both Actuator and Validator features have been selected, but only after the execution of the **delta** DActuator, as specified with the **when** and **after** clauses.

Another example of variability modeling using CVL is illustrated in Fig. 5.3 where it is considered that the objects deviceCpu, sensor and dualOutput and the links between them are included by default. CVL uses fragment substitutions to specify additional transformations. A fragment substitution is similar to a delta module. It consists of a *placement* fragment (surrounded by dashed lines) containing the elements that will be removed and a *replacement* fragment (surrounded by solid lines) containing the elements that will be inserted instead. CVL defines fragments by surrounding them with an imaginary closed curve and placing boundary points whenever links cross the curve. Boundary points are elements that fully define all references going in and out of placement/replacement fragments.

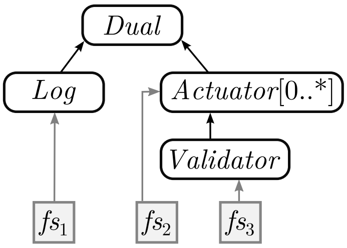

Figure 5.4: A variability specification tree

The following fragment substitutions—where the fragments are specified by the surrounding boundary points pa, pb, ra, rb etc.—specify the possible changes to the base model. The CVL fragment substitutions fs_1, fs_2 and fs_3 correspond to the previously introduced Delta modules DLog, DActuator and DValidator, respectively.

$\mathsf{fs}_1\{\ placement\{\mathsf{pa},\mathsf{pb}\}\ replacement\{\mathsf{ra}\}\ binding\{(\mathsf{pa},\mathsf{ra})\}\ \}$

$\mathsf{fs}_2\{\ placement\{\mathsf{pa},\mathsf{pb}\}\ replacement\{\mathsf{rb}\}\ binding\{(\mathsf{pb},\mathsf{rb})\}\ \}$

$\mathsf{fs}_3\{\ placement\{\mathsf{pc},\mathsf{pd}\}\ replacement\{\mathsf{rc},\mathsf{rd}\}\ binding\{(\mathsf{pc},\mathsf{rc}),\ (\mathsf{pd},\mathsf{rd})\}\ \}$

Finally, CVL organizes the features in a variability specification tree and binds the fragment substitutions accordingly as shown in Fig. 5.4. In this example the *Actuator* is a VClassifier meaning that fs_2 can be executed multiple times to obtain multiple actuators.

5.1.2 Comparative analysis.

The results of the comparative analysis of CVL, Delta Modeling and OVM are summarized as follows:

- **Modeling the variations and the configurations** is done for all three variability modeling languages using some form of feature models or decision models. OVM is closely related to decision modeling; CVL employs a variability specification tree as shown in Fig. 5.4 (i.e. an enhanced feature model with cardinalities [72]); Delta modeling accepts any form of feature or decision model.

Each language includes some form of constraint language over the features/decisions/variability specifications.

- **The realization of features by artifacts** is done in multiple ways. OVM uses an annotative approach, i.e. it uses language annotations to mark which artifacts from the base model are implementing specific decisions. Delta modeling uses a transformational approach, i.e. the model consists of partially ordered deltas where each one adds, removes and modifies artifacts from previous ones. CVL variation points, especially the fragment substitution, can define complex transformations which are constrained by the variability specification tree structure.
- **Product derivation** requires a clear understanding of how to execute a variability model given a specific configuration. CVL defines how each kind of variation point is executed. The variation points are partially ordered by the resolution tree structure (e.g. Fig. 5.4 shows how the fragment substitution fs_3 which is guarded by *Validator* can only be executed after fs_2 which is guarded by *Actuator*). However, execution is not confluent as two variation points at the same level can have contradictory effects resulting in different variants depending on the order. OVM uses a projection on the model artifacts referenced by the selected variants. Delta Modeling executes each delta module by adding, modifying and removing elements as specified by the modules. The modules also specify a partial order using special clauses. The execution can be made confluent by adding conflict resolving deltas for any pair of conflicting deltas [73].
- **Orthogonality** is the degree to which variability models can be reused on different kinds of artifacts (requirements, models, code, all specified using different languages)[33]. Orthogonality allows using standard tools to manipulate and process base models at every step of the product line engineering process. CVL defines a clear distinction between feature modeling (via a variability specification tree), and the model transformations over artifacts (via variation points). The variability model is completely separate from the artifacts. OVM design is based on orthogonality. The artifacts can be anything from requirements to model elements or code fragments. Delta Modeling can be applied to any language, textual and graphical alike. Delta modules can use references to artifacts in a separate model to specify what is added, removed and modified.

The analysis has shown that languages such as CVL, Delta Modeling and OVM use different concepts to address similar tasks. This means that while they are different in many ways, they still share many core characteristics.

5.2 Using a core variability modeling language to formally specify real languages.

The second hypothesis is proposing that if a core language can be used to formally specify variability modeling and variant derivation in general, then it can be used to formally specify how these core concepts are represented in real world variability modeling languages.

Hypothesis 1.2. *A core variability modeling language can be used to develop a formal specification for a real variability modeling language. While the core language would not be powerful enough to capture the real language entirely, it would be sufficiently flexible to specify common concepts of popular variability modeling languages.*

To test this hypothesis I have designed and developed a core language for separate variability modeling, Featherweight VML, using requirements based on the comparative analysis of CVL, Delta Modeling and OVM. In the following I briefly describe how concepts that are specific to the aforementioned variability modeling languages can be represented in Featherweight VML.

- **Modeling the variations and the configurations** is done in Featherweight VML through feature trees and by allowing abstract features with no implementation [74]. This way feature models, decision models and CVL's variability specification models can be represented. Also, by employing a constraint language we can define any kind of dependencies between features.
- **The realization of features by artifacts** is done in Featherweight VML through fragment substitutions exclusively. The other CVL variation points, delta modules and the OVM annotative technique can be reproduced by employing fragment substitutions.
- **Product derivation** is formalized in Featherweight VML through a number of inference rules which describe exactly which artifacts must be copied from the product line base model into the variant. Featherweight VML also specifies constraints on the feature model and on the fragment substitutions so that the execution is always confluent.
- **Orthogonality** is also an essential characteristic of Featherweight VML which borrows the layered architecture of CVL as it is general enough to be used with both OVM and Delta Modeling.

5.2.1 Featherweight VML Syntax.

Featherweight VML is a core language, meaning that it can represent abstractions of real languages. Figure 5.5 illustrates the syntax of Featherweight VML through a small example based on the device product line from the previous section. It shows an entire variability model composed of a feature model in panel (a) and a set of fragment substitutions in panel (b) over a base model in panel (c).

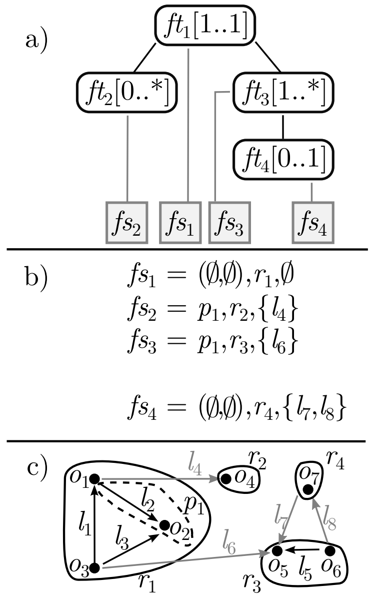

Figure 5.5: a) The feature model.
b) Fragment substitutions.
c) Core artifacts.

The variability model and all its elements feature model, fragment substitutions are defined formally below.

Definition 6/PAPER A. A *variability model* is a triple, $(Fs, Fm, \textsf{mapping})$, where Fs is a set of fragment substitutions, Fm is a feature model and $\textsf{mapping} : Fs \rightarrow Fm$ maps each fragment substitution to a feature.

Fig. 5.5.a shows a feature model composed of four features: ft_1, ft_2, ft_3 and ft_4 (an abstraction of the features *Dual*, *Log*, *Actuator* and *Validator* in Fig. 5.4). Each feature has an associated cardinality which specifies how many times it may be instantiated in a configuration. The model also specifies how features are composed (illustrated as black lines) where a parent feature must be instantiated before any of its child features can also be instantiated. Finally, Fig 5.5.a shows how a set of fragment substitutions, fs_1, fs_2, fs_3 and fs_4, can be mapped to the features (illustrated as gray lines), meaning that a fragment substitution will be executed if and only if the feature to which it is mapped is instantiated by a product configuration.

The feature model is defined formally:

Definition 5/Paper A. A *feature model* is a rooted directed tree of features, $Fm = (Ft, ft_0, \mathsf{parent})$, where Ft is a set of features, $\mathsf{parent} \subseteq Ft \times Ft$ is a connected acyclic parent relation with no sharing (a tree), and $ft_0 \in Ft$ is the root of the tree. We write $\mathsf{parent}\, ft_2 = ft_1$, if feature ft_1 is a parent node of ft_2 in Fm.

Each feature ft has an associated cardinality constraint $\mathsf{card}\, ft = (\mathsf{min}\, ft, \mathsf{max}\, ft)$, where $\mathsf{min}\, ft, \mathsf{max}\, ft \in \mathbb{N} \cup \{*\}$, $\mathsf{min}\, ft \leq \mathsf{max}\, ft$ (the symbol $*$ is considered greater than any natural).

Figure 5.5.b shows a set of fragment substitutions which represent the delta modules Dual, DLog, DActuator and DValidator in Fig. 5.2 from the previous section. Each fragment substitution is formed of a placement fragment containing all the elements that must be removed from the variant, a replacement fragment containing all the elements that must be added to the variant and a *binding* which is a set of new links that bind the fragments together. In the example, fs_2 has an empty placement fragment, $(\emptyset, \emptyset)$, a replacement fragment referred by its identifier, r_3 and a binding which is described explicitly as a set of links (l_7, l_8).

Definition 2/Paper A. A *fragment substitution* fs is a triple (p, r, b) where p is a placement fragment containing all the elements that must be removed, r is the replacement fragment and b is a set of new links called a *binding*. The placement and replacement fragments are disjoint, $p \dot{\cap} r = (\emptyset, \emptyset)$.

Figure 5.5.c presents the base model of the product line. The base model is represented abstractly as a pair of sets of finitely many objects (illustrated as black dots) and finitely many links (illustrated as unidirectional black arrows)—essentially, a graph where the objects and links represent the model elements from Fig. 5.1. Placement fragment $p1$ is encircled by a dashed line, replacement fragments r_1, r_2, r_3 and r_4 are encircled by solid lines and boundary links are illustrated as unidirectional gray arrows.

Similarly to the definitions presented above, **Paper A** contains definitions for the base models as well as definitions for additional concepts (i.e. fragment substitution boundary, $\mathsf{boundary}\, fs$, and closure, $\lceil p \rceil_{fs}$) that help isolating the effects of each fragment substitution from those of other fragment substitutions.

5.2.2 Featherweight VML Semantics.

The execution of a Featherweight VML model is done in two steps. This is illustrated in Fig 5.6 which extends Fig 5.5 with three new panels. Figure 5.6.d shows a product configuration. In this example there are four feature instances, i_1, i_2, i_3 and i_4 which instantiate features ft_1, ft_3, ft_3 again and ft_4 respectively. There is no instance of ft_2. As a rule, the structure of the instance tree is constrained by that of the feature tree, e.g. the instances of feature ft_3 can only exist as children of an instance of ft_1, and are bounded by the cardinality of ft_3.

The model is executed in two steps. First, the variability model (a), the fragment substitutions (b) and the configuration (d) are *flattened*, which means that for each feature in the feature model, all the fragment substitutions that are mapped to it are being cloned once for each instance of the feature. The figure shows how fragment substitution fs_1 mapped to feature ft_1 is cloned once for instance i_1. The new fragment substitution is fs_{11}. Fragment substitution fs_2 is not being cloned at all because feature ft_2 has not been instantiated, while fragment substitution fs_3 is being cloned twice for the two instances of ft_3: i_2 and i_3. The flattening process can be seen in the transition from panel (b) to panel (e) in the figure.

When fragment substitutions are being cloned, all the model elements to which they refer are being cloned also. Figure 5.6.f shows how the flattening affected the base model: fragment r_3 has been duplicated into r_{12} and r_{13}, while the binding link l_6 has also been duplicated into l_{15} and l_{17}, keeping both clones well connected to the rest of the system.

The flattening step is formalized by two inference rules. The rule COPY-INDEP clones a fragment substitution fs_i whose effects are independent from any other fragment substitutions (does not remove or connect to any artifacts added by another substitution). The rule COPY clones a fragment substitution fs_i that is influenced in some manner by another fragment substitution fs_j. In both cases, the rule makes use of a function, new-fs, that generates a new fragment substitution by cloning all fragments, objects and links from the base model and updating references accordingly.

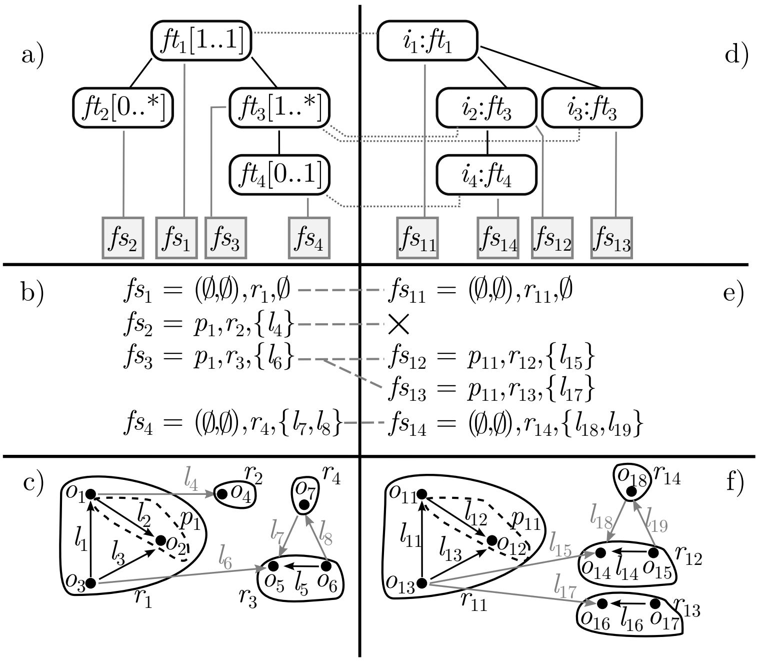

Figure 5.6: a) The feature model. b) Fragment substitutions. c) Core artifacts. d) A configuration. e) Flattened substitutions. f) New artifacts after flattening.

$$\frac{i \in \mathit{Cfg} \quad \mathsf{mapping}^{-1}(\mathsf{ty}\, i) = fs_i \quad \lceil p \rceil_{fs_i} = (\emptyset, \emptyset)}{\mathsf{new\text{-}fs}_{i,\_}\, fs_i\, \emptyset \in [\![M, \mathit{Cfg}]\!]}\ (\textsc{copy-indep})$$

$$\frac{\begin{array}{c} i, j \in \mathit{Cfg} \quad \mathsf{mapping}^{-1}(\mathsf{ty}\, i) = fs_i \quad \mathsf{mapping}^{-1}(\mathsf{ty}\, j) = fs_j \quad fs_i \sqsubset fs_j \\ fs_i = (p_i, r_i, b_i) \quad fs_j = (\_, r_j, \_) \quad j \in \mathsf{parent}^{*} i \end{array}}{\mathsf{new\text{-}fs}_{i,j}\, fs_i\, r_j \in [\![M, \mathit{Cfg}]\!]}\ (\textsc{copy})$$

After the model has been flattened, the variability model (a), the fragment substitutions (b), the configuration (d) and even the original base model (c) can be discarded. The second step of the execution is concerned with the new set of fragment substitutions (e) and the new artifacts (f), which contain all the information required to build a product variant for this specific configuration. For this step two rules, OBJ-COPY and LNK-COPY, are extracting from the objects and links that were generated in the flattening step only those that are added by some fragment substitution and are not removed by any other fragment substitution, effectively executing all the flattened fragment substitutions.

$$\left[\begin{array}{cc} \dfrac{\begin{array}{c} o \in (\bigcup_{(\_,r,\_)\in Fs} r_{\mathsf{Obj}}) \\ o \notin (\bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Obj}}) \end{array}}{o \in [\![Fs]\!]_{\mathsf{Obj}}}\ (\textsc{obj-copy}) & \dfrac{\begin{array}{c} l \in (\bigcup_{(\_,r,b)\in Fs} r_{\mathsf{Lnk}} \cup b) \\ l \notin (\bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Lnk}}) \\ \mathsf{src}\, l, \mathsf{tgt}\, l \notin \bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Obj}} \end{array}}{l \in [\![Fs]\!]_{\mathsf{Lnk}}}\ (\textsc{lnk-copy}) \end{array}\right]$$

The formal specification of Featherweight VML is completed by a set of well-formedness constraints, a proof of correctness and a theorem of confluence that are detailed in **PAPER A**.

Featherweight VML is designed to express core characteristics that are shared by many variability modeling languages. Although some real languages use additional concepts that are not currently supported by Featherweight VML, the language can be easily extended. For example, CVL uses a constraint language over the variability specification tree; Featherweight VML can be extended with a constraint language over the feature model.

Featherweight VML should not be considered "yet another variability modeling language". Being a minimal, core language it would be impractical to use for managing the variability of a real system; instead, a more specialized language would be more user-friendly and would provide shortcuts for concrete use cases. The main advantages of Featherweight VML over a specialized variability modeling language is its declarative and generic nature which makes it especially applicable in automated verification scenarios

5.3 Effective verification of variant derivation through translation validation.

The third hypothesis is proposing a way of building evidence for the correctness of variant derivation tools. Featherweight VML provides a way of formalizing the specification of variability modeling for a wide range of languages that have similar core concepts. There is still the problem that the source core of any implementation might be too complex or unavailable thus it cannot be used to prove the correctness of the tool. Hypothesis 1.3 puts forward the idea of treating the implementation as a black box and using a formal specification of the derivation algorithm and translation validation to build evidence for each execution of the tool.

Hypothesis 1.3. *While it is difficult to prove that a variant derivation tool will always produce correct variants, one can use a core specification of the variant derivation algorithm and translation validation to validate each execution and either formally produce evidence of correctness or identify any errors that might have occurred.*

To present this contribution I start by explaining the principle of translation validation. Then I will describe how I implemented the technique using the Coq theorem prover, followed by a demonstration.

5.3.1 Translation validation

Translation validation has been first proposed as a method for verifying programming language compilers [75]. It is a technique that produces evidence that a tool which implements a translation/transformation behaves according to a formal specification. Unlike other verification techniques, translation validation does not prove that the tool/compiler output will always be correct. Instead, it requires that the tool be executed and then validates that the output corresponds to the input according to a formal relation between the two. So it only proves that an actual execution of the tool was correct. My contribution is adapting and evaluating the method for verifying product derivation tools with the aim of qualifying them.

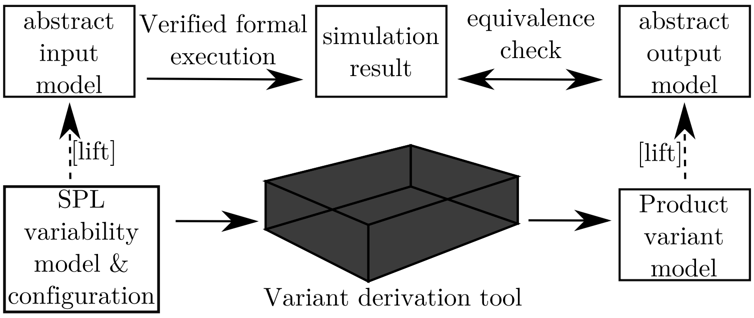

Figure 5.7: The translation validation process

Figure 5.7 shows an overview of the translation validation process. The variant derivation tool is considered a black box so the implementation details (e.g. complexity, platform, dependencies) are irrelevant to the technique. It can be an implementation of any variability modeling language (e.g. CVL, Delta Modeling, OVM). Its input is a variability model along with the core artifacts of the product line and a specific configuration which are written in some modeling language (e.g. UML, Ecore). It produces a product variant based on the input and the derivation algorithm prescribed by the variability language.

In order to validate that the output is produced correctly, translation validation requires the following ingredients:

1. A common semantic framework for the representation of the source code and the generated target code.
2. A formalization of the notion of *correct implementation* as a refinement relation, based on the common semantic framework.
3. A proof method which allows to prove that one model of the semantic framework, representing the produced target code, correctly implements another model which represents the source.
4. Automation of the proof method, to be carried out by an analyzer which, if successful, will also generate a proof script.
5. A rudimentary proof checker that examines the proof script produced by the analyzer and provides the last confirmation for the correctness of the translation.

I have adapted the ingredients presented above to the domain of software product lines and variant derivation as follows.

For the original translation validation method, a *common semantic framework* for the representation of the input and output meant that a new semantic framework would be needed for each different language specification. By using Featherweight VML as the common semantic framework we can reuse the same setup for validating any tool for any language that can be abstracted to Featherweight VML. In Fig. 5.7 the abstraction is represented by the [lift]ing arrows.

I simulate the variant derivation via a formal execution of the abstract input model. The simulation is an embedding of Featherweight VML in the semantic framework of the Coq theorem prover, which allows me to prove its correctness. The simulation result is then compared to the abstraction of the actual derivation result. If the two results are equivalent (isomorphic models) then we can say that the product variant model conforms to the input configuration.

The advantages of simulating the derivation on an abstract model are multiple. The simulation tool can be reused to validate various production tools; the same simulation tool could work for translation validation of both Delta Models and CVL. While the actual tools depend on external libraries, the simulation tool can be implemented stand-alone, without dependencies to unverifiable components. Since the simulation is performed on abstractions of the models, the simulation tool can have a smaller source code and is easier to verify. While most production tools are written in imperative languages, the simulation tool can be written using declarative or functional languages for which it is easier to write proofs of correctness.

5.3.2 Fragment substitution embedding in Coq.

As a proof-of-concept I have embedded the fragment substitution syntax and semantics in Coq. The current implementation assumes an already flattened model and it does not yet embed the feature models and configurations. Through this embedding I obtain:

- a textual format for the fragment substitution syntax (Coq uses the Gallina specification language);
- executable functions that implement the inference rules of the fragment substitution copying semantics;
- theorems of correctness and confluence for the variant derivation algorithm and automatically checkable proofs.

To exemplify the implementation of Featherweight VML in Coq, I will revisit some of the definitions and inference rules mentioned above.

Fragment substitution definition in Featherweight VML.	**Fragment substitution implementation in Coq.**

Definition 2/PAPER A. A *fragment substitution fs* is a triple (p, r, b) where p is a placement fragment containing all the elements that must be removed, r is the replacement fragment and b is a set of new links called a *binding*. The placement and replacement fragments are disjoint, $p\hat{\cap}r = (\emptyset, \emptyset)$.	`Inductive FragSubst := fragsubst :`
	`  Fragment -> Fragment -> LinkSet -> FragSubst.`
	
	`Notation "p ,. r ., b" := (fragsubst p r b) (`
	`    at level 65).`

Table 5.1: The fragment substitution in Featherweight VML and Coq.

Table 5.1 shows how the fragment substitution definition from Featherweight VML is embedded in Coq. The inductive type `FragSubst` provides a constructor, `fragsubst`, that takes as input two fragments of the `Fragment` type and a set of links of the `LinkSet` type. The fragments and the set of links correspond to the placement, the replacement and the binding, respectively. The `Fragment` type is also defined in Coq as a pair of an `ObjectSet` and a `LinkSet` which form the nodes and edges of a graph.

Copying objects from the fragment substitution set *Fs* in Featherweight VML.	**Copying objects from the base model object set `obj`, given the fragment substitution set `fss` in Coq.**

$\dfrac{\begin{array}{c} o \in (\bigcup_{(\_,r,\_)\in Fs} r_{\mathsf{Obj}}) \\ o \notin (\bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Obj}}) \end{array}}{o \in [\![Fs]\!]_{\mathsf{Obj}}}\ (\textsc{obj-copy})$	`Fixpoint objCopy (obj : ObjectSet) (fss :`
	`    FragSubstSet) : ObjectSet :=`
	`  match obj with`
	`    | [ ] => [ ]`
	`    | h::t =>`
	`    if bSetContainsObject (replacementObjects`
	`        fss) h`
	`      && !(bSetContainsObject (placementObjects`
	`          fss) h)`
	`    then h::(objCopy t fss)`
	`    else (objCopy t fss)`
	`  end.`

Table 5.2: The OBJ-COPY rule in Featherweight VML and Coq.

Table 5.2 shows how the OBJ-COPY rule from Featherweight VML is embedded in Coq. In Featherweight VML the rule copies objects from the replacement fragments of all fragment substitutions, after filtering out the objects found in placement fragments.

In the Coq embedding the objects are not actually contained by the fragments, but are only addressed by reference. Therefore the OBJ-COPY rule is implemented as a fixpoint function, `objCopy`, which takes as input the entire set of objects that exist in the base model, `obj` of type `ObjectSet`. It also takes as input the set of fragment substitutions, `fss` of type `FragSubstsSet`. The function recursively traverses the set of objects, `obj`, and it determines for each object whether it is added by a replacement, but not removed by any placement fragment. The objects that pass the test are copied into the variant.

Confluence in Featherweight VML.	**Confluence in Coq.**

Lemma 2/PAPER A. *Given a set of fragment substitutions, there exists a unique product variant model created by the above rules.*	`(* The order of fragment substitutions in a`
	`    set does not influence the result. *)`
	`Theorem objCopyOnEqualFss :`
	`  forall obj fss1 fss2,`
	`  FragSubstSetEqual fss1 fss2`
	`  -> ObjectSetEqual (objCopy obj fss1)`
	`                    (objCopy obj fss2).`
	
	`(* The order of objects in a set does not`
	`    influence the result. *)`
	`Theorem objCopyOfEqualSets :`
	`  forall obj1 obj2 fss,`
	`  ObjectSetEqual obj1 obj2`
	`  -> ObjectSetEqual (objCopy obj1 fss)`
	`                    (objCopy obj2 fss).`

Table 5.3: Confluence theorems for the `objCopy` function.

Table 5.3 shows the confluence theorems for the `objCopy` function. In Featherweight VML the confluence of the entire semantics is partly based on Lemma 2 which states that for any arbitrary set of fragment substitutions there is only one way of building a product variant. Lemma 2 refers to the inference rules OBJ-COPY and LNK-COPY and it is shown that it holds by construction as the objects and links are deterministically selected from a finite set.

In Coq the `objCopy` function takes as input an object set, `ObjectSet`, and a fragment substitution set, `FragSubstsSet`. To prove that the function is indeed confluent I have formally proved two theorems in the Coq proof system: `objCopyOnEqualFss` and `objCopyOfEqualSets`. The first one, `objCopyOnEqualFss`, shows that applying `objCopy` on the same object set `obj`, but on two equal fragment substitution sets `fss1` and `fss2` produces equal results. The second one, `objCopyOfEqualSets`, shows that applying `objCopy` on two equal object sets, `obj1` and `obj2`, and a single fragment substitution set `fss` also produces equal results. The equality properties, `FragSubstSetEqual` and `ObjectSetEqual`, hold if the sets contain the same elements, but do not consider the order of these elements.

The confluence of the `lnkCopy` function is proven using similar theorems. The final result of the two copying functions is a pair of an object set and a link set which together form a model graph for the product variant. It is self evident that the order of elements in the two sets is irrelevant to the correctness of the result since two results where the elements are in different orders will still produce isomorphic graphs which are considered to be the same result.

The Coq file contains approximately *300 lines of code* implementing FwVML syntax and semantics and over *3000 lines of code encoding theorems and proofs* of correctness.

5.3.3 Demonstration.

To demonstrate the verification of variant derivation tools through translation validation I implemented Micro CVL: a *small* variant derivation tool designed as a subset of CVL. In **PAPER A** I describe how Micro CVL can handle variability over any Ecore-based model. I also provide a lifting algorithm from Micro CVL to Featherweight VML and a minimal example of input and output models on which the tool and validation technique are applied.

Another—perhaps more compelling—example is that of the Base-Variability-Resolution (BVR) language presented in the VARIES deliverable *D4.7 Variability Analysis Solutions* [76]. BVR is a considerably bigger subset of CVL developed by SINTEF in Norway, so it stands as a better example than the homegrown Micro CVL. BVR is implemented as an Eclipse plug-in. The implementation project has *over one thousand Java classes* where the product derivation code is mixed with Eclipse plug-in code and tests. The size of the code base and the heavy dependence on Eclipse libraries makes it impossible to build any formal proof that the tool would always produce correct results. By comparison, I implemented the lifting operation, which abstracts the UML and BVR input and output files to Coq representations of Featherweight VML, with just *six Java classes*. This implementation along with the Coq code can formally ascertain that the output produced by the complex BVR code adheres to the structural properties captured by Featherweight VML.

Overall I have shown that a variant derivation tool built from a large code base and depending on many third party libraries can be verified by implementing a considerably smaller lifting operation and using the provided Featherweight VML embedding in Coq to validate each execution.

Chapter 6

Challenge 2: Trustworthy reengineering of software product lines.

6.1 Incomplete transformations are less costly to implement while still being sufficiently effective.

The first hypothesis is addressing the difficulty of designing and implementing a complete reengineering transformation. An ideal transformation would be able to handle any arbitrary input program. However, we have seen that the complexity of the input language increases the complexity of the transformation so developing a complete transformation is often infeasible in practice. The hypothesis puts forward the idea that an incomplete transformation which focuses on handling an input language subset that is used in the project at hand, would be less costly to implement while still being sufficiently effective.

Hypothesis 2.1. *While it is difficult to design and implement complete code rewriting transformations, in practice there are situations where an incomplete transformation that focuses on a specific subset of the input language grammar is less costly to implement while it will be sufficiently effective.*

To support this hypothesis I present a software modernization transformation applied to an industrial project of considerable size. The project, provided by Danfoss Power Electronics[1], involves a configuration tool used to adapt frequency converters to a particular application (i.e. controlling electric motors that drive various machinery). The configuration tool consists of thousands of C++ functions for accessing and validating the configuration parameters. The purpose of the modernization is to convert the parameter functions from imperative C++ code to a declarative configuration model that can be used with an off-the-shelf verifier. All the functions compute a result (either the value of the parameter or a Boolean value checking its accessibility and validity). The declarative form of a function is an expression that must compute the same result as the original imperative code.

[1] `http://www.danfoss.com`

6.1.1 Example.

The input of the transformation is exemplified in Fig. 6.1. The program is a simplified and anonymized version of a real function from the Danfoss code base. It is a constraint-enforcing function which checks that the input variables `selectedConfigParameter` and `selectedOptionParameter` have correct values with respect to each-other and to some constants `config1` and `option1`. The transformation must turn this imperative program into a declarative constraint that must preserve the names of the input variables and must compute the same truth value.

```
Configuration config = selectedConfigParameter;
Option opt = selectedOptionParameter;
bool result = false;
switch (config) {
  case config1:
    if (opt == option1) result = true;
    break;
  default:
    result = true;
    break;
}
return result;
```

Figure 6.1: An example input program.

The result of the transformation is shown in Fig. 6.2. It is a pure (i.e. side effect free) C++ expression that accepts the same input parameters and computes the same value as the input function. It is produced in several steps by applying a series of syntactic transformations:

- The `switch`-statement is replaced with a nested `if-else` conditional statement.
- A series of simplifications are performed on the conditional statements, so we end up with a straightforward control flow.
- The conditional statement is finally reduced to a ternary expression of the form `e1 ? e2 : e3`.

For this particular example a handful of transformation rules are sufficient. The only syntactic constructs that are matched are variable declaration, initialization and allocation, the switch statement, the conditional statement, the return statement and the equality operator. Other constructs such as the true/false values are preserved by the transformation while any other C++ feature that is not in this example can be ignored completely.

By contrast, implementing a generic transformation that can perform this modernization task on any arbitrary input is a difficult and costly task for the following reasons:

```
(selectedConfigParameter == config1
  && selectedOptionParameter == option1)
```

Figure 6.2: The output program after the transformation.

- C++ is a large programming language, with a complex specification and occasional undefined behavior. Understanding the semantics of a C++ program is often complicated by pointer aliasing, dereferencing unknown memory addresses (null pointers, array index overflows) among other language features. Writing a generic transformation that would have to cover most or all of the C++ specification is comparable to writing a full-blown compiler.
- Many of the C++ language constructs cannot be directly translated to a declarative model. For example: unbounded loops and recursive function calls would have to be unrolled indefinitely, potential side effects from accessing input/output streams would have to be handled, and any inlined assembly code would have to be interpreted.

- The allocated development resources and available time are limited, making it unfeasible to develop a tool from scratch. The transformation would have to employ existing tools and frameworks which may have an impact on its scope.

The true scope of the transformation, somewhere in between the trivial example above and handling the complete C++, is determined by the project at hand.

6.1.2 Case size.

The input code consists of 4119 C++ functions from a configuration application. These functions encode dependencies, visibility constraints, and default values of approximately one thousand configuration parameters of a frequency converter. The use of C++, as opposed to C, is modest. No object-oriented aspects are used, except for member access and limited encapsulation. Most functions have straightforward control flow, without `goto` statements, loops and recursion. The most common statements are conditional branching and `switch` statements. The rare `for` loops all have a constant number of iterations. Other constructs include variable declarations and usage of local variables in arithmetic and comparison expressions, calls to pure functions (for instance for converting physical units), and casts between different types (both C-style casts and `static_casts`). There are few references to static and singleton member variables and functions, which act similarly to other function calls.

There are 14502 source lines of code (SLOC) in total that need to be modernized in the pilot project, and more similarly-looking configurators for other products waiting for modernization afterwards. As many as 3348 of the 4119 functions are already in expression form; these do not need to be modernized, but should not be broken by the modernization. The remaining 771 functions have 14.47 SLOC on average.

6.1.3 Research questions.

In **Paper B** I present lessons learned from designing this modernization transformation, a process guided by two research questions:

RQ1 Is it feasible to design the aforementioned transformation using off-the-shelf technology in a limited time?

RQ2 What are the main obstacles and challenges in designing and implementing the transformation? Are the transformation tools sufficiently efficient for the task?

The transformation was designed in four stages: (i) establishing the design principles, (ii) selecting the development tools, (iii) implementing the transformation and (iv) calculating basic metrics and evaluating the efficiency of implementing the transformation. **Paper B** contains observations about the design and implementation process for each one of the stages.

6.1.4 Design principles established for the project.

The first stage was to understand the exact purpose of the modernization and how it would integrate in the regular development process.

By discussing the process with Danfoss engineers it became clear that the transformation would have to be automatic in order to minimize down-time on the main development of the code base. The code to be modernized is only locked for the time it takes to run the transformation and it can be evolved freely while the transformation is being implemented and tested.

Observation 1/PAPER B. *Automatic transformations allowed us to decouple and parallelize the regular development and the modernization activities.*

Translation of imperative C++ code into a declarative form is in general very complex. However, it was possible to save resources by sacrificing generality by handling only the code at hand (and similar). In fact, for small number of complex fragments (involving loops) it was even simpler to modernize manually by hard coding rules that addressed these special cases. This way the transformation remained automatic on the subsequent reruns that would include those special cases.

Observation 2/PAPER B. *Since modernization is a one-off transformation it was economically beneficial to sacrifice generality, and instead focus on the code at hand whenever it simplifies things.*

Because the transformation is incomplete by design it should succeed on the inputs that it was designed for, but it should fail as early as possible on inputs that violate the assumptions made when sacrificing generality (e.g. arbitrary nesting of constructs in C++, or use of unexpected language elements). This was achieved by following a *fail-fast* programming style, i.e. making preconditions for rules as precise as possible (so that rules are not applied when failing) and writing explicit assertions when possible (when a rule is in principle applicable, but it does not cover all the cases, for simplicity of development).

Observation 3/PAPER B. *The fail-fast programming approach helped to avoid implementing anything that is not strictly required for the modernization project to succeed, while retaining quality on the expected inputs.*

Because the transformation should preserve the semantics of the input code, various semantic mechanisms (e.g. type analysis, static single assignment form) were considered to solve the task. However, the initial analysis indicated that this would raise the complexity of the implementation considerably, and likely also lead to polluting the output with identifiers generated in the process (unfamiliar to developers). For an incomplete transformation of a known code base it seems much easier to work with syntactic transformations. Even typing and simple data flow information can be captured with a finite number of patterns if we only need to work with limited code base.

Observation 4/PAPER B. *We found working with syntax much more effective for an ad hoc transformation task, than when using semantic data and semantically informed rewrites.*

6.1.5 Selecting the development tools.

Having the design principles in place, the second stage was to select the development tools. The choice was made based on the availability of a C++ grammar, and the flexibility and usability of the tool itself.

The candidates that already had usable C++ grammars were Spoofax [77] and TXL [78]. Out of these two, the grammar offered by TXL was up to date, handled ambiguity quite well and TXL provides mechanisms for relaxing the grammar, making it easy to adapt to this particular case. TXL is a standalone command-line tool, with few dependencies, so it took virtually no time to get it to run. In fact, simple proof of concept rewrites were working after only 4 hours of experiments with TXL

> **Observation 5/Paper B.** *Simplicity and the integration with the languages to be transformed influenced the selection of tools stronger than the properties of the transformation language or a rewriting paradigm.*

6.1.6 Implementing the transformation.

TXL is a functional/rule-based transformation language that rewrites syntax trees into syntax trees. The transformation consists of a series of smaller steps that gradually prepare, transform and clean up the code. The output of each step serves as input for the next one. The fail-fast approach means that each transformation step checks whether the input has the expected form, otherwise it throws an error message.

The overall algorithm applied by the transformation is:

1. The program fragment is checked for format assumptions: all branches return a value, there are no loops and goto jumps, no calls to non-pure methods, etc. The transformation does not establish the purity of functions itself, but consults a white-list of names of pure functions provided as a parameter.
2. All preprocessor `#ifdef` directives in the program are cleaned up, and converted to ordinary `if` statements.
3. Local variable assignments are inlined in the following expressions in the right order (i.e. going from the last assignment to the first). When all references to the local variables are eliminated, their declarations are also removed.
4. All `switch` statements are converted to a series of `if` statements.
5. Series of sequential `if` statements are simplified into nested `if` statements, such that the fragments are reduced to a single root statement, all additional functionality being implemented through substatements of the root.
6. `if` statements are converted to ternary expressions and `return` statements are replaced by the expression they return.

Paper B contains a larger account of technical challenges and of decisions made during the implementation as well as the following observation:

> **Observation 6/Paper B.** *We succeeded to implement a flow-aware syntactic transformation, including constant folding and variable inlining, which enabled us to produce code that uses the same identifiers, and reminiscent structure of the input programs.*

6.1.7 Basic metrics.

To evaluate the efficiency of implementing such a transformation with off-the-shelf tools **PAPER B** presents a series of metrics such as source code size, implementation time and execution time.

The entire transformation (including grammar definitions, excluding white space and comments) spans 6515 lines of code. The core C++ grammar, which is provided as a resource from the TXL website, has 137 nonterminal rules (595 lines of code). In addition, 98 grammar rules are defined or redefined in the transformation to adapt the grammar to our needs.

The transformation has 468 definitions (rules or functions) in total, where 171 of the definitions are function definitions and the remaining 297 are rule definitions (as opposed to functions, rules are called repeatedly until a fixed point is reached).

It took 3 months full time work of experienced software developer to implement this transformation (including learning TXL, domain understanding, unit testing and meetings with the industrial partner). The cost is deemed acceptable, especially given that the company has several more products to modernize, that include configuration code, for which the transformation would be largely reusable.

The transformation execution lasts 30 minutes on the 4119 functions out of which 105 functions are not handled—the transformation reported errors, marking that these functions contain special cases and should be migrated manually to keep the whole process cost-effective.

6.2 Effective verification of complex transformations through translation validation.

The second hypothesis is addressing the problem of verifying the transformation.

Hypothesis 2.2. *If a rule-based code rewriting transformation aims to preserve certain properties of the code, one can use a formal specification of the property and translation validation to produce a proof of correctness and identify any errors that might have occurred.*

6.2.1 Example.

```
Configuration config = selectedConfigParameter;
Option opt = selectedOptionParameter;
bool result = false;
switch (config) {
  case config1:
    if (opt == option1) result = true;
    break;
  default:
    result = true;
    break;
}
return result;
```

Figure 6.3: An example input program.

To understand the difficulty of verifying the correctness of such a complex transformation consult Fig. 6.3 which revisits the example input code. Figure 6.4 shows the declarative expression resulting from transforming the code. Even for such a small example, checking that the transformation has preserved the semantics of the input program is non-trivial. To check the semantic equivalence of the programs all possible execution paths must be considered, where each path forms a relation (known as a path condition) between the values of the input parameters and the return value. To demonstrate semantic equivalence we need to demonstrate the equivalence between the path conditions of the imperative program and those of the declarative program.

In this case the transformation introduced an error in the code: the path condition $selectedConfigParmeter \neq config_1$ $\wedge selectedOptionParameter = option_1$ $\wedge\ result = \texttt{true}$ is valid for the input program in Fig. 6.3 but is not for the transformed program in Fig. 6.4, which shows that these two programs are not semantically equivalent.

```
(selectedConfigParameter == config1
  && selectedOptionParameter == option1)
```

Figure 6.4: The output program after the transformation.

The expected output is shown in Fig. 6.5. Even in this example the difference is subtle, but easy to spot due to the small size of the source code and the low number of execution paths. The real input programs reach over one hundred lines of code and there are too many of them to verify manually.

```
(selectedConfigParameter == config1)
  ? (selectedOptionParameter == option1)
  : true
```

Figure 6.5: The expected output.

6.2.2 Verification challenges.

TXL is a very powerful transformation language that can express arbitrary computation using deep pattern matching, complex side conditions, global state and rewriting. Individual rules and functions of a transformation are non-modular and might intermediately break syntactic and semantic correctness properties. Verifying individual rules alone would not guarantee the correctness of the transformation. Instead, it is important that the transformation is validated as a whole.

Observation 7/PAPER B. *Our transformation was written in a non-modular fashion (as most transformations we have seen). This made it hard to verify the transformation rules individually.*

Because the transformation design sacrificed generality to speed up the development, many of the transformation rules were implemented in an ad hoc fashion specifically to handle patterns present in the use case. For example, the rule $\texttt{if}\,(E)\,\texttt{return}\,\mathit{true} \rightarrow \texttt{return}\,E$ is not correct in general, but it is correct under assumption that the `if` statement occurs last in the `main` function, which is the case in which we want to apply it. Besides such irregular special cases, the transformation depends heavily on the implementation of the TXL engine and on an external Ruby script which controls the sequence of transformation steps, adding even more complexity to the algorithm.

To combat this complexity, the verification method treats the transformation as a black-box and only checks that the input and corresponding transformed output agree semantically. This works since the size of input is manageable and it was not expect for the transformation to be generally applicable in other unrelated settings.

Observation 8/PAPER B. *We were able to treat a complex transformation as a black-box, reducing the validation to checking whether the provided input and transformed output agree according to specific semantic properties.*

6.2.3 Research questions.

Expanding the research questions that guided the development of the transformation, in **PAPER B** I also present two research questions that were answered by designing and running the verification:

RQ3 Can high assurance methods be used at acceptable costs to validate the transformation?

RQ4 What kind of errors are found in a transformation implemented by experts?

6.2.4 Verification approach.

By design, any transformation implemented in TXL is constrained to produce well formed syntax trees. This means that the verification technique would only have to focus on verifying the semantic correctness of the transformation output with respect to the input.

Several semantic analysis techniques (such as abstract interpretation) were investigated. The symbolic executor KLEE [79] stood out as being able to build a very precise semantic model of the C++ programs, handling the majority of features used in the code at hand, and proving to be cost-effective in integrating it in the automation process.

The verification process is as follows:

1. The transformation is executed on an input function and produces an output model. The fail-fast approach catches any special case that triggers a precondition.
2. Both the input and output are embedded in a test wrapper. This step may statically catch some errors.
3. The input and output code is compiled to LLVM−IR using Clang. Any compilation errors are caught at this step.
4. The LLVM−IR is symbolically executed with KLEE to produce semantic models (as logic formulas) which KLEE then compares for equivalence. If the input and output are not equivalent, then KLEE produces a counter example which can be used to pinpoint the error in the transformation.

One of the bigger challenges of using this approach was that KLEE requires the input code be compiled to LLVM[80] intermediate representation (LLVM−IR), including all external libraries (otherwise the symbolic execution may not terminate or provide incomplete results). However, the code-base at hand consisted of individual functions which called external functions with unknown implementations. Therefore, before compilation the code of each function had to be closed with suitable instrumentation: creating stubs for unknown functions and for data structures with straight-forward constructors.

Observation 9/PAPER B. *We were able to use off-the-shelf tools to perform semantic verification of programs, with a moderate amount of effort required to pre-process the input to the tool.*

I provide a more detailed explanation of how the code had to be prepared for verification in **PAPER B**.

6.2.5 Validation results.

The validation execution lasts 7 minutes on all 4119 transformed functions out of which 3348 are evaluated trivially to pass verification–the output is identical to the input. For the 771 cases which could not be validated trivially (the output was not identical to the input) we present the results in Table 6.1.

The fail-fast approach of the transformation led to the identification of 105 erroneous cases during translation. The rest of the cases went through the verification algorithm. Three more error cases were caught during the preliminary steps of the verification, i.e. preparing the case for compilation. The C++ compiler detected three other error cases, leaving 640 cases to be checked for equivalence with KLEE. By comparing the semantic models computed by KLEE 562 cases were proven to be equivalent, 50 concrete error cases were identified and 28 were false positives. The last 20 cases in the table were not handled intentionally as they contained assertions which could not be represented by the standard version of KLEE. Assessing and integrating other versions of KLEE that could handle assertions was deemed infeasible and the cases were left to be handled manually.

The verification stage led to the following observations:

Observation 10/PAPER B. *It was possible to analyze a substantial amount of the modernized code automatically, and only 20 corner cases were left to be handled manually.*

Table 6.1: Erroneous transformation cases caught by each step of the validation process.

	Step	#Cases
1	Failing transformation precondition (not handled, requiring manual inspection)	105
2	Failing silently due to unhandled syntactic structures (caught statically during preliminary steps of verification)	3
3	Caught by C++ compiler	3
4	Checked for equivalence using KLEE	640
4a	Validated being equivalent	562
4b	Concrete bug cases with provided counter-examples	50
4c	False positives with spurious counter-examples (due to over-approximation of functions, and representation mismatch)	28
5	Unhandled cases containing assertions (intentional, due to design limitations of the validation technique)	20

Observation 11/Paper B. *When the code base of our modernization project reached a certain complexity it became infeasible to find all bugs through expertise and unit testing. Validation of semantics was essential to ensure that the output code worked correctly.*

By manually analyzing the erroneous cases, the errors were traced back to bugs in the transformation rules. In total eight kinds of bugs were identified throughout the implementation.

- *Bug 1*: Function call is dropped in some paths. The bug was caused by an incomplete rewrite rule. When a rule matches a functional call in a return statement and forgets to reinsert the function call on replacement.
- *Bug 2*: Structure replaced by a constant integer. This bug also happens due to misuse of deep pattern search and a broken rule assumption.
- *Bug 3*: Conditional branches are dropped. It was caused by incomplete rewrite rules. The rewrite rule matches a nested conditional followed by other branches, and then rewrites the conditional correctly but forgets to handle the other branches.
- *Bug 4*: The unexpected exceptions. This bug happens due to overconstrained pattern matching and broken rule assumption.
- *Bug 5*: Use of undeclared variables. This bug occurs due to a combination of dynamic reparsing capabilities and wrong target type in expression.
- *Bug 6*: Negation dropped in result. The simplest bug found by the KLEE-based verifier is where the transformation had transformed the whole input correctly except a negation operation which was missing in the output.
- *Bug 7*: Conditional with error code assignment dropped. This bug happens due to the dynamic reparsing capabilities and eager removal of source data. It originates in the inlining phase where some abstract syntax is broken by wrongly inserted textual syntax, and subsequently a rule that removed empty conditional branches was applied.
- *Bug 8*: Variable declarations without assignment not handled. Similar to Bug 2, a combination of a broken rule assumption and misuse of deep pattern search was the cause of this bug.

Paper B contains a more thorough analysis of the bugs as well as a classification and a formal justification for the procedure. By correlating the bugs with the erroneous cases the following observation was formulated:

Observation 12 /Paper B. *Simple bugs hit wide, complex bugs hit deep. Simple semantic errors affected a large number of functions while complex errors were found in a few but bigger functions.*

Overall I have shown that an incomplete transformation designed to handle specific cases can be effectively used in a code reengineering project at low implementation costs. Even if the transformation is incomplete by design, the level of trust in the produced output can be increased using a fail-fast approach to catch unhandled cases and translation validation to assert properties such as semantic equivalence.

Chapter 7

Challenge 3: Lightweight tools for quality assurance in the presence of variability.

7.1 Using off-the-shelf state-of-the-art analysis tools to verify software product lines by rewriting the variability.

Given that the availability of variability-aware analysis tools is limited, another approach for the efficient verification of software product lines would be to replace compile-time variability with run-time variability (or non-determinism) [67]. Effectively transforming the product line into a single product, would enable the use of state-of-the-art analysis tools, even if they cannot handle variability.

Hypothesis 3.1. *By rewriting the variability in the language of the source code, some of the analysis tools that do not normally handle variability will be applicable to the entire product line. Analyzing the reconfigured code with existing tools is a cost-efficient alternative to lifting the analysis tools to product line level.*

In **Paper C** I present a transformation that turns any C product family with variability managed through preprocessor `#ifdef` directives into a single C product with run-time variability. The transformation is outcome-preserving in the sense that the outcomes of the single product (i.e. after the transformation) are equal to the union of all outcomes of all individual variants that can be derived from the product line.

The principle of the hypothesis is illustrated in Fig. 7.1. Instead of using specialized variability-aware tools to analyze program families (which would be tedious and labor intensive), this transformation-based approach facilitates the use the standard off-the-shelf single-program analysis tools to achieve the same goal. The limitation of this approach is that the analysis may not obtain the most precise conclusive results for all individual variants. Of course, this depends on the particular analysis and tool that are used.

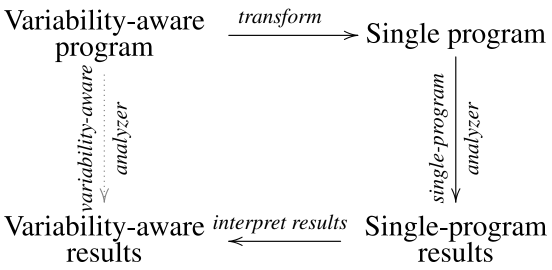

Figure 7.1: The overview of a transformation-based approach for verification of program families. The single-program analyzer can be any verification oracle for single programs, such as: symbolic executor, type checker, static analyzer, model checker.

7.1.1 Example.

To illustrate the concept of variability rewriting I present the example in Fig. 7.2. The left side shows a preprocessor-based family of C programs which uses two (Boolean) features A and B. The two features give rise to a family of four variants defined by the set of configurations $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$.

For each configuration a different variant (single program) can be generated by appropriately resolving `#if` directives. For example, the variant for $A \wedge B$ will have both features A and B enabled (set to true), thus yielding the following body of `foo() {int x := 1; x := x+1; x := x-1; return 2/x;}` The variant for $\neg A \wedge \neg B$ is: `foo() {int x := 1; return 2/x;}`.

In such program families, errors (also known as *variability bugs* [81]) can occur in some variants (configurations) but not in others. In the example program family shown in Figure 7.2, the variant $\neg A \wedge B$ will crash at the `return` statement when it attempts to divide by zero. On the other hand, the other variants do not contain the division-by-zero error since the value of `x` at the `return` statement is: 1 for variants $A \wedge B$ and $\neg A \wedge \neg B$, and 2 for $A \wedge \neg B$. To detect these errors, we would have to either analyze each variant individually, or to use an analysis tool that can parse and process the `#if` and `#endif` directives accordingly.

The alternative is to transform the code in such a way that it can be analyzed by tools that cannot handle C Preprocessor directives. Figure 7.2 (right side) shows a single program obtained by applying a variability-rewriting transformation on the family shown in the left side. All features are first declared as ordinary variables and non-deterministically initialized to 0 or 1, then all `#if` and `#endif` directives are transformed into ordinary `if` statements with the same conditions. Thus, the division-by-zero error is present in this single program and corresponds to a trace when A is initialized to 0 and B to 1.

The set of outcomes of the transformed program (Figure 7.2, right side) is equal to the union of outcomes of all individual variants from the family (Figure 7.2, left side). Therefore, the division-by-zero error is present in the transformed program. In general, the transformed program obtained from the original program family can be analyzed by various single-program verification tools, in order to find variability errors or to confirm the absence of errors in the given program family.

```
int foo() {


  int x := 1;
  #if (A) x := x+1; #endif
  #if (B) x := x-1; #endif
  return 2/x;
}
```

```
int foo() {
  int A := rand() % 2;
  int B := rand() % 2;
  int x := 1;
  if (A) x := x+1;
  if (B) x := x-1;
  return 2/x;
}
```

Figure 7.2: Variability-related transformations in practice.

7.1.2 A formal model for the transformation.

To formalize and prove the correctness of the transformation **Paper C** provides a formal model based on the IMP language. The paper describes two extensions of IMP: IMPor used to represent run-time variability (non-determinism), and $\overline{\text{IMP}}$ used to represent compile-time variability.

IMP is an imperative language with two syntactic categories: expressions and statements. Expressions include integer constants, variables, and binary operations. Statements include a "do-nothing" statement `skip`, assignments, statement sequences, conditional statements, while loops, and local variable declarations. Its abstract syntax is summarized using the following grammar:

$$\begin{array}{lcl} e & ::= & n \mid \mathtt{x} \mid e_0 \oplus e_1 \\ s & ::= & \mathtt{skip} \mid \mathtt{x} := e \mid s_0 \; \mathtt{;} \; s_1 \mid \mathtt{if}\; e \;\mathtt{then}\; s_0 \;\mathtt{else}\; s_1 \mid \mathtt{while}\; e \;\mathtt{do}\; s \mid \mathtt{var\ x}\mathtt{:=}e \;\mathtt{in}\; s \end{array}$$

In the above, n stands for an integer constant, $\mathtt{x}$ stands for a variable name, and $\oplus$ stands for any binary arithmetic operator. The sets of all statements, s, and expressions, e, generated by the above grammar are denoted by *Stm* and *Exp*. A state of a program is a *store* mapping variables to values (integer numbers), $\mathit{Val} = \mathbb{Z}$. The set of all possible stores is denoted by $\mathit{Store} = \mathit{Var} \to \mathit{Val}$. Expressions are computed in a given store, denoted by σ below. A function $\mathcal{E} : \mathit{Exp} \times \mathit{Store} \to \mathit{Val}$ defined below by induction on e, maps an expression and a store to a single value, thereby formalizing evaluation of expressions.

$$\mathcal{E}(n,\sigma) = n, \qquad \mathcal{E}(\mathtt{x},\sigma) = \sigma(\mathtt{x}), \qquad \mathcal{E}(e_0 \oplus e_1,\sigma) = \mathcal{E}(e_0,\sigma) \oplus \mathcal{E}(e_1,\sigma)$$

The language **IMPor** is obtained by extending IMP with a non-deterministic choice operator '`or`' which can non-deterministically choose to evaluate either of its arguments.

$$e \quad ::= \quad ... \mid e_0 \;\mathtt{or}\; e_1$$

With the non-deterministic construct '`or`' it is possible for an expression to evaluate to a set of different values in a given store. Therefore, now the evaluation function $\mathcal{E} : \mathit{Exp} \times \mathit{Store} \to \mathcal{P}(\mathit{Val})$ is redefined as follows:

$$\begin{array}{l} \mathcal{E}(n,\sigma) = \{n\}, \qquad \mathcal{E}(\mathtt{x},\sigma) = \{\sigma(\mathtt{x})\}, \qquad \mathcal{E}(e_0 \;\mathtt{or}\; e_1,\sigma) = \mathcal{E}(e_0,\sigma) \cup \mathcal{E}(e_1,\sigma) \\ \mathcal{E}(e_0 \oplus e_1,\sigma) = \{v_0 \oplus v_1 \mid v_0 \in \mathcal{E}(e_0,\sigma), v_1 \in \mathcal{E}(e_1,\sigma)\} \end{array}$$

The language $\overline{\textbf{IMP}}$ is also an extension of IMP, so it does not contain the '`or`' construct. Its abstract syntax includes the same expression and statement productions as IMP, extended by new compile-time conditional statements for encoding multiple variants of a program. A finite set of Boolean variables

$\mathbb{F} = \{A_1, \ldots, A_n\}$ describes the set of available *features* in the program family. Each feature may be *enabled* or *disabled* in a particular variant. A *configuration* k is a truth assignment or a valuation which gives a truth value to each feature, i.e. k is a mapping from $\mathbb{F}$ to $\{\text{true}, \text{false}\}$. Any configuration k can also be encoded as a conjunction of literals: $k(A_1) \cdot A_1 \wedge \cdots \wedge k(A_n) \cdot A_n$, where $\text{true} \cdot A = A$ and $\text{false} \cdot A = \neg A$. The set $\mathbb{K}$ contains all *valid* configurations defined over $\mathbb{F}$ for a family. The set of *feature expressions*, denoted *FeatExp*, is the set of well-formed propositional logic formulas over $\mathbb{F}$ generated using the grammar: $\phi ::= \text{true} \mid A \in \mathbb{F} \mid \neg\phi \mid \phi_1 \wedge \phi_2$. $\overline{\text{IMP}}$ envelops variable code with the statements "$\texttt{#if}\ (\phi)\ s\ \texttt{#endif}$" and "$\texttt{#if}\ (\phi)\ \texttt{var x:=}n\ \texttt{in #endif}\ s$" which contain a feature expression $\phi \in \textit{FeatExp}$ as a presence condition, such that only if ϕ is satisfied by a configuration k then the code between `#if` and `#endif` will be included in the variant for k.

$$s ::= \ldots \mid \texttt{#if}\,(\phi)\ s\ \texttt{#endif} \mid \texttt{#if}\,(\phi)\ \texttt{var x:=}n\ \texttt{in #endif}\ s$$

The semantics of $\overline{\text{IMP}}$ has two stages: first, given a configuration k compute an IMP single program without `#if`-s; second, evaluate the obtained variant using the standard IMP semantics. The first stage is a simple *preprocessor* specified by the projection function π_k mapping an $\overline{\text{IMP}}$ program family into an IMP single program corresponding to the configuration k. The projection π_k copies all basic statements of $\overline{\text{IMP}}$ that are also in IMP, and recursively pre-processes all sub-statements of compound statements. For example, $\pi_k(\mathsf{skip}) = \mathsf{skip}$ and $\pi_k(s_0;s_1) = \pi_k(s_0);\pi_k(s_1)$.

In **PAPER C** the semantics of the languages is completed with small-step operational semantic rules and a description of special cases.

7.1.3 Variability-related transformations.

The aim of the transformation is to rewrite an input $\overline{\text{IMP}}$ program family $\overline{s}$ into an output IMPor program $\overline{s'}$. A pre-transformation phase is converting each feature $A \in \mathbb{F}$ into the variable A, which is non-deterministically initialized to 0 or 1 (meaning to false or true). Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be the set of available features in the family $\overline{s}$, then we have the following initialization fragment in the resulting program $\texttt{pre-t}(\overline{s})$:

$$\texttt{pre-t}(\overline{s}) = \texttt{var}\ A_1 := 0\ \texttt{or}\ 1\ \texttt{in} \ldots \texttt{var}\ A_n := 0\ \texttt{or}\ 1\ \texttt{in}\ \overline{s}$$

After all the features are translated into variables, a number of rewrite rules are applied on the program. The rules have the form: $\psi \vdash s \rightsquigarrow s'$ meaning that: if the current program family being transformed matches any abstract syntax tree (AST) node of the shape s nested under $\#\texttt{if}$-s with the resulting presence condition that implies $\psi \in \textit{FeatExp}$ (i.e. in context ψ) then *replace s by s'*. Formally, applying the rule $\psi \vdash s \rightsquigarrow s'$ to a family:

$$\ldots \texttt{#if}\ (\phi_1) \ldots \texttt{#if}\ (\phi_n) \ldots ; s; \ldots \texttt{#endif} \ldots \texttt{#endif} \ldots$$

where $\phi_1 \wedge \ldots \wedge \phi_n \implies \psi$, then results in the transformed program:

$$\ldots \texttt{#if}\ (\phi_1) \ldots \texttt{#if}\ (\phi_n) \ldots ; s'; \ldots \texttt{#endif} \ldots \texttt{#endif} \ldots$$

The function $Rewrite(\overline{s}, \psi \vdash s \rightsquigarrow s')$ represents the final transformed program $\overline{s'}$ obtained by repeatedly applying the rule $\psi \vdash s \rightsquigarrow s'$ on $\overline{s}$ and its transformed versions until a point where this rule can not be applied is reached (a fixed point of the rule).

7.1.4 Transformation rules.

Below I present a few of the rules used by the transformation.

Conditional variable declaration. In C product lines, this rule is used to transform local variables that are declared conditionally. Figure 7.3 shows how a variable that is declared with different types under different features is rewritten to include both versions in a single program.

```
#if (A) int x; #endif
#if (B) long x; #endif
```

```
int xA;
long xB;
```

Figure 7.3: Rewriting conditional variable declarations.

Formally, the rule transforms a local variable that is declared conditionally within a given context $\psi \in \mathit{FeatExp}$:

$$\psi \vdash \texttt{\#if}\ (\phi)\ \texttt{var x:=}n\ \texttt{in \#endif}\, s, \delta \rightsquigarrow \texttt{var x}_{new}\texttt{:=}n\ \texttt{in}\ s, \delta[(\texttt{x}, \phi) \mapsto \texttt{x}_{new}] \tag{7.1}$$

where $\texttt{x}_{new}$ is a fresh variable name that does not occur as a free variable in s.

Conditional variable use. This rule is used to transform expressions in which variables depend on the configuration. In Fig. 7.4 the variable `x` is declared conditionally for two mutually exclusive presence conditions `A` and `!A`. Later, when it is used in an expression, the transformation uses a conditional expression to split the execution.

```
#if (A) int x; #endif
#if (!A) long x; #endif
y = x + 1;
```

```
int xA;
long xNA;
y = (A ? xA : xNA) + 1;
```

Figure 7.4: Rewriting conditional variable use.

Formally, the rule handles the case when a local variable is used within a context $\psi \in FeatExp$:

$$\psi \vdash \texttt{y:=}e[\texttt{x}], \delta \rightsquigarrow \begin{cases} \texttt{y:=}e[\delta(\texttt{x}, \phi)], \delta & \text{if } \exists!\phi{\in}\delta^{\texttt{fe}}(\texttt{x}).\psi{\Longrightarrow}\phi \\ \{\texttt{\#if}\ (\phi_1)\ \texttt{y:=}e[\delta(\texttt{x}, \phi_1)]\ \texttt{\#endif;} & \text{if } \exists\phi_1,\ldots\phi_n{\in}\delta^{\texttt{fe}}(\texttt{x}). \\ \ldots & \phi_1{\Longrightarrow}\psi,\ldots,\phi_n{\Longrightarrow}\psi \\ \texttt{\#if}\ (\phi_n)\ \texttt{y:=}e[\delta(\texttt{x}, \phi_n)]\ \texttt{\#endif}\}, \delta & \\ \texttt{y:=}e[\texttt{x}], \delta & \text{otherwise} \end{cases} \tag{7.2}$$

where $e[\texttt{x}]$ means that the variable `x` occurs free in the expression e.

Conditional variable define. The rule applies when a value is assigned to a conditionally declared variable. As opposed to the previous rule, this transformation uses `if` statements instead of conditional expressions, as shown in Fig. 7.5.

```
#if (A) int x; #endif
#if (!A) long x; #endif

x = y + 1;
```

```
int xA;
long xNA;
if (A) { xA = y + 1; }
else   { xNA = y + 1; }
```

Figure 7.5: Rewriting conditional variable assignment.

Formally, when a local variable is assigned to within a context $\psi \in \mathit{FeatExp}$:

$$\psi \vdash \texttt{x:=}e,\delta \rightsquigarrow \begin{cases} \delta(\texttt{x},\phi)\texttt{:=}e,\delta & \text{if } \exists!\phi{\in}\delta^{\texttt{fe}}(\texttt{x}).\psi{\Longrightarrow}\phi \\ \{\texttt{\#if }(\phi_1)\ \delta(\texttt{x},\phi_1)\texttt{:=}e\ \texttt{\#endif;} & \\ \dots & \text{if } \exists\phi_1,\dots\phi_n{\in}\delta^{\texttt{fe}}(\texttt{x}). \\ & \phi_1{\Longrightarrow}\psi,\dots\phi_n{\Longrightarrow}\psi \\ \texttt{\#if}(\phi_n)\ \delta(\texttt{x},\phi_n)\texttt{:=}e\ \texttt{\#endif}\},\delta & \\ \texttt{x:=}e,\delta & \text{otherwise} \end{cases} \tag{7.3}$$

After applying the above rules, only statements are conditionally defined using #`if`-s.

#`if` elimination. The last rule simply replaces #`if` directives with `if` statements. This is only applied when the conditional directive envelopes complete statements or statement lists, as shown in Fig. 7.6.

```
#if (A)
x = y + 1;
z = sqrt(y);
#endif
```

```
  if (A) {
    x = y + 1;
    z = sqrt(y);
  }
```

Figure 7.6: Rewriting conditional statements.

Formally, given the set of valid configurations $\mathbb{K}$ can be equated to a propositional formula [82], say $\kappa \in \mathit{FeatExp}$, such that $\kappa = \vee_{k\in\mathbb{K}}k$. The last replaces #`if`-s with ordinary `if`-s whose guards are strengthen with the feature model κ, thus taking into account only valid configurations $\mathbb{K}$ of a family.

$$\psi \vdash \texttt{\#if }(\phi)\ s\ \texttt{\#endif} \rightsquigarrow \texttt{if }(\phi{\wedge}\kappa)\ \texttt{then}\ s\ \texttt{else skip} \tag{7.4}$$

Theorem of outcome preservation. Let $\delta_0 = []$ be the empty environment. Let $\mathit{Rewrite}^{\texttt{preserve}}(\texttt{pre-t}(\overline{s}),\delta_0)$ be the final transformed program $\overline{s'}$ obtained by applying the above rules on the pre-transformed program $\texttt{pre-t}(\overline{s})$. The following result shows that the set of final answers from terminating computations of $\overline{s'}$ coincides with the union of final answers from terminating computations of all variants from $\overline{s}$.

Theorem 1. *Let* $\overline{s'} = \mathit{Rewrite}^{\texttt{preserve}}(\texttt{pre-t}(\overline{s}),\delta_0)$.

$$[\![\overline{s'}]\!] = \bigcup_{k\in\mathbb{K}}[\![\pi_k(\overline{s})]\!]$$

Paper C also contains a couple of normalization rules, as well as a proof of Theorem 1.

7.1.5 Implementation.

To demonstrate the approach I implemented the tool C Reconfigurator which converts program families written in the C language with preprocessor **`#ifdef`** directives into a single product with run-time variability encoded as C `if` statements with the intent of using off-the-shelf analysis tools that would not be applicable otherwise.

All transformations are implemented using Xtend[1]. The tool[2] calls the variability-aware parser SUPERC [21] to parse the code with preprocessor annotations. It uses Binary Decision Diagrams (BDD's) for encoding feature expressions that must be evaluated to determine the conditional compilation. SUPERC returns an abstract syntax tree in which the variability is encoded as choice nodes over feature expressions. In particular, a choice node is a node with two children, such that the left child of the choice node is included in the result of those configurations for which the given feature expression is satisfied; otherwise the right child of the choice node is included in the parsing result when the feature expression is not satisfied. C RECONFIGURATOR implements variability-related transformation rules similar to those described above for the $\overline{\text{IMP}}$ product families. In fact, since IMP is a subset of C, all rewritings described above transfer directly to C. Additionally, C RECONFIGURATOR implements other rules that handle specific variability cases as well as normalization rules that make the reconfigured code more intelligible for humans. Some of these rules that handle functions which have different signatures in different configurations, variables with different types and arrays of different lengths are described in more detail in **PAPER C**.

The hypothesis is confirmed by running various analysis tools such as FRAMA-C [83], CLANG [84] and LLBMC [85] on the reconfigured code. The tools were able to analyze the code and identify bugs and vulnerabilities.

7.2 Using off-the-shelf analysis tools on the reconfigured code is sufficiently effective for practical purposes.

Due to the rewriting transformation, it is expected that some bugs would be missed on the reconfigured code, even thought the analysis would detect them on a product derived from the product line (i.e. if all product variants were analyzed by brute force).

Hypothesis 3.2. *Although variability rewriting does not provide a guarantee of effectiveness with respect to analysis, some analysis tools will retain sufficient effectiveness on the reconfigured code so that they may be used in practice.*

7.2.1 Research questions.

The evaluation aims to show that rewriting variability enables the use state-of-the-art single-program verification tools as oracles to verify realistic C program families. The experiment is set to answer the following research questions:

RQ1 How precise is the technique?
RQ2 How efficient is the verification oracle to identify variability bugs after transforming the code using the reconfiguration technique?

In particular, the experiment uses a collection of reported variability bugs[81, 86] to determine if the

[1] `http://www.eclipse.org/xtend/`.

[2] C RECONFIGURATOR tool is available at: `https://github.com/models-team/c-reconfigurator`.

analysis tools FRAMA-C [83], CLANG [84] and LLBMC [85] can first detect the bugs on specific single product variants and then on the reconfigured product family. FRAMA-C is a framework for modular static (dataflow) analysis of C programs. The CLANG project includes the Clang compiler front-end and the Clang static analyzer for several programming languages, including C. LLBMC (the low-level bounded model checker) is a software model checking tool for finding bugs in C programs.

7.2.2 Subject files and experimental setup.

To evaluate this approach the C RECONFIGURATOR is applied on a collection of source files with real variability bugs extracted from three benchmarks: Linux, BusyBox and Libssh. In particular, the collection contains files extracted from the VDBb [3] database as well as real variability bugs from Libssh provided by Medeiros et al. [86]. For the VDBb files, Abal et al. [81] created a simplified version for each bug they found by capturing the same essential behavior (and the same problem) as in the original bug. Simplified bugs are independent of the kernel code and were derived systematically from the error trace.

The evaluation process is as follows:

1. From the source file with compile-time variability a single program variant, known to manifest the bug, is derived.
2. The analysis tools FRAMA-C, CLANG and LLBMC are used to confirm the bug on the program variant.
3. If the bug is confirmed, then the C RECONFIGURATOR is used on the source file containing compile-time variability.
4. The analysis tools FRAMA-C, CLANG and LLBMC are used on the reconfigured file, attempting to reproduce the bug.

Table 7.1 presents the characteristics of the subject files, listing the file id, bug type, number of features, lines of code, and commit hash (clickable) for each project. This collection consists of a diverse set of bug types such as null pointer dereferences, buffer overflow, and uninitialized variable. In total, there are 11 distinct kinds of bugs. The number of features per file varies from one to seven. In addition, the number of lines of code ranges from 12 to 165 for the simplified files (from VBDb), and from 1404 to 2959 for real files (from Libssh).

All experiments were executed on a 64-bit Mac OS X 10.11.5 machine, Intel®Core^{TM} i7 CPU running at 2.3 GHz with 8 GB memory. The performance numbers reported constitute the median runtime of fifty independent executions.

7.2.3 Results.

To answer the first research question, the precision of the technique is measured by checking how many bugs are preserved by the reconfiguration process. To answer the second question, the performance of the technique is measured by comparing the time it takes to run the analysis tool on the single program variant and on the reconfigured code. All experiment materials are available online at https://github.com/models-team/c-reconfigurator-test (including the tool and scripts).

[3] http://VBDb.itu.dk.

ID	Bug type	$\lvert\mathbb{F}\rvert$	LOC	Hash
VBDb Linux files				
1	null pointer deref.	5	165	76baeeb
2	null pointer deref.	3	112	f7ab9b4
3	null pointer deref.	4	55	ee3f34e
4	null pointer deref.	3	34	6252547
5	buffer overflow	1	58	8c82962
6	buffer overflow	1	33	60e233a
7	read out of bounds	7	69	0f8f809
8	uninitialized var.	2	54	7acf6cd
9	uninitialized var.	1	54	bc8cec0
10	uninitialized var.	1	53	30e0532
11	uninitialized var.	2	38	1c17e4d
12	uninitialized var.	2	26	e39363a
13	undefined symbol	4	25	7c6048b
14	undefined symbol	2	20	2f02c15
15	undefined symbol	2	20	6515e48
16	undefined symbol	2	19	242f1a3
17	undeclared identifier	3	37	6651791
18	undeclared identifier	2	20	f48ec1d
19	wrong # of args	1	12	e67bc51
20	multiple funct. defs	2	21	e68bb91
21	dead code	1	19	809e660
22	incompatible type	2	27	d6c7e11
23	assertion violation	2	79	63878ac
24	assertion violation	2	75	657e964
25	assertion violation	2	41	0988c4c
VBDb BusyBox files				
26	null pointer deref.	1	28	199501f
27	null pointer deref.	2	24	1b487ea
28	uninitialized var.	2	28	b273d66
29	undefined symbol	1	42	cf1f2ac
30	undefined symbol	2	27	ebee301
31	undeclared identifier	1	35	5275b1e
32	undeclared identifier	1	19	b7ebc61
33	incompatible type	3	46	5cd6461
Real Libssh files				
34	null pointer deref.	6	1404	0a4ea19
35	null pointer deref.	4	1428	fadbe80
36	uninitialized var.	3	2959	2a10019

Table 7.1: Characteristics of the benchmark files.

File	Frama-C			
ID	buggy variant		reconfigured	
	y/n	time	y/n	time
VBDb Linux files				
1	✓	218	✓	235
2	✓	220	✓	225
3	✓	215	x	236
4	✓	218	✓	224
5	✓	218	✓	227
6	✓	213	✓	227
7	✓	218	✓	225
8	✓	241	✓	250
9	✓	224	✓	230
10	✓	216	inc	224
11	✓	234	✓	224
12	✓	216	inc	227
13	✓	239	✓	248
14	✓	237	✓	244
15	✓	224	✓	248
16	✓	213	✓	222
17	✓	216	✓	230
18	✓	210	✓	224
19	✓	210	✓	224
20	✓	213	x	228
21	✓	239	x	240
VBDb BusyBox files				
26	✓	230	✓	234
27	✓	224	✓	234
28	✓	237	inc	237
29	✓	230	✓	236
30	✓	231	✓	228
31	✓	220	✓	228
32	✓	216	✓	224

Table 7.2: Verification results using Frama-C. File ID represents the bug file; columns buggy variant and reconfigured show the tool results on the given buggy variant code and on the reconfigured program family code, respectively. The result can be: ✓—bug found; x—bug not found; *inc*—inconclusive. Tool run time is shown in milliseconds (ms).

Evaluating precision. Table 7.2 shows the results of verifying VBDb files by using Frama-C. The table has two main columns: buggy variant and reconfigured that depicts the tool results on the buggy variant code and on the reconfigured program family code, respectively. Each checkmark (✓) means that the same bug was found by the verification tool. Otherwise, the result is either x—bug not found or *inc*—inconclusive, which means that Frama-C was able to detect a bug that is different from

```c
int do_sect_fault()
{
  return 0;
}

int main() {
  #ifndef ARM
  do_sect_fault();
  #endif
  return 0;
}
```

```c
int do_sect_fault()
{
  return 0;
}

int main() {
  if (!_rec_ARM)
  do_sect_fault();

  return 0;
}
```

Figure 7.7: Rewriting conditional variable assignment.

the bug in the product variant.

Frama-C has a 78% success rate in finding bugs on the reconfigured code, detecting the bug in 22 out of 28 cases. The C Reconfigurator preserved a variety of bugs such as buffer overflow and uninitialized variable.

In three specific cases (cf. file ids 10, 12 and 28), Frama-C did not report the original bug (uninitialized variable) as a definitive problem, but it did report that some variable might be uninitialized in some conditions. This happened because Frama-C uses a "may" analysis which describes information that may possibly be true along one path to the given program point and computes a superset of all uninitialized variables in all variants.

Frama-C could not identify the bug in the reconfigured file in three cases (cf. file ids 3, 20 and 21). For example, file 21 contains dead code, which is a function (`do_sect_fault()`) that is never called when feature `ARM` is enabled (see Fig. 7.7, left column). The C Reconfigurator transforms the code by changing the `#ifdef` into an ordinary `if` condition, making the function available for the transformed single program (i.e., the function is not dead any more), as shown in Fig. 7.7, right column. This is a limitation of the technique.

The remaining VBDb files that could not be verified with Frama-C, as it does not handle incompatible type and assertion violation bugs, were instead verified with Clang and LLBMC. Because LLBMC requires the code to be compiled with Clang, the two tools are treated as a single verification oracle. The results are shown in Table 7.3 where there is no difference in reporting whether the bug was found by Clang during the compilation or afterwards by LLBMC. Similarly, the real Libssh files were reconfigured and analyzed with Clang/LLBMC and the results are shown in Table 7.4.

The success of Clang/LLBMC in identifying bugs in reconfigured code, both in simplified and real files confirms that the C Reconfigurator preserves a wide range of variability bugs and enables the use of off-the-shelf analysis tools for software product lines.

Based on analyzing 36 simplified and real variability bugs from Linux, BusyBox and Libssh, the first research question can be answered:

Answer RQ1 (precision). *The* C Reconfigurator *enables single-program verification tools such as* Frama-C, Clang, *and* LLBMC *to successfully detect most of the variability bugs on the reconfigured code.*

FILE ID	CLANG/LLBMC			
	BUGGY VARIANT		RECONFIGURED	
	yes/no	time (ms)	yes/no	time (ms)
VBDb LINUX FILES				
22	✓	18	✓	30
23	✓	10	✓	5
24	✓	9	✓	5
25	✓	8	✓	4
VBDb BUSYBOX FILES				
33	✓	37	✓	48

Table 7.3: Verification results using CLANG and LLBMC on VBDb files.

FILE ID	CLANG/LLBMC			
	BUGGY VARIANT		RECONFIGURED	
	yes/no	time (ms)	yes/no	time (ms)
34	✓	1526	✓	1302
35	✓	792	✓	898
36	✓	145	✓	146

Table 7.4: Verification results using CLANG and LLBMC on real files.

Evaluating efficiency. The performance of the technique is evaluated by comparing time needed by the verification tools to analyze the buggy variant code (BUGGY VARIANT column) and the reconfigured program family code (RECONFIGURED column). Tables 7.2, 7.3 and 7.4 show the measured time in milliseconds. We can see that the analysis times in both cases are similar. In fact, FRAMA-C takes less than half a second to analyze each file regardless whether it is a variant or a reconfigured file. For instance, FRAMA-C analyzes file 1 in 218 and 235 milliseconds on the variant code and on the reconfigured program family code, respectively. Given that file 1 contains a null pointer dereference and has five features, if we apply brute force approach (that analyzes all variants individually one by one) to this file using FRAMA-C it would take approximately 218 ms times the number of valid variants. File 1 has only seven valid variants due to feature restrictions. So, the total brute force time would be 218 × 7, that is, 1526 ms. In this way, we obtain significant speed-up to verify the program family using our approach. Using CLANG/LLBMC gives similar results in terms of performance, on both simplified and real files.

In general, the performance of analyzing a reconfigured code is similar to analyzing only one variant, which gives us a speed-up proportional to the number of valid variants of a program family.

The second research question can be answered by observing that:

> **Answer RQ2 (performance).** *The* C RECONFIGURATOR *speeds-up the family-based analysis via single-program verification tools, so that we can* ***efficiently*** *detect variability bugs on the reconfigured code.*

PAPER C contains a more detailed presentation of the results.

Overall, I have shown that the variability rewriting technique can be efficiently and effectively used to find variability bugs in software product lines, using off-the-shelf state-of-the-art analysis tools that otherwise are unable to handle variability.

Chapter 8

Conclusion and Future Work

8.1 Challenge 1: Qualifying product variant derivation tools.

Problem 1.1. *Due to the great variation among variability modeling languages, it is impossible to develop a standard specification of variability modeling and of the variant derivation algorithm.*

Featherweight VML does not attempt to standardize variability modeling and variant derivation. Instead, it provides a common representation for some core concepts that are similar in popular variability modeling languages. Through a comparative analysis of the Common Variability Language, Delta Modeling and the Orthogonal Variability Model, I show how they handle the main aspects of variability modeling in a similar manner. For example, they all model the variations and the configurations through some form of feature model [31] or decision model [32]. Most CVL variability specifications can be reduced to features with cardinalities and the variation points are all specific cases of the fragment substitution. Featherweight VML can be seen as a generalization of OVM. We can use abstract features to group variation points together, giving OVM a tree structure while retaining the same meaning. Delta modules are almost identical to fragment substitutions.

All of the above modeling languages also use some form of transformation to manipulate the product artifacts by adding, removing and binding elements together. CVL, for example, defines many kinds of variation points such as object existence (which adds or removes an object) or link substitution (which replaces one link with another). Upon inspection, I found that all of the CVL variation points are specialized versions of the fragment substitution. Another example is the Delta module which is a fragment substitution guarded by an application condition.

In fact, many more of the existing variability modeling languages and tools employ concepts derived from feature models and decision models [32], which Featherweight VML can represent. Also, the Featherweight VML fragment substitution is generic enough to represent any kind of transformation on the base model. The graph representation of base models in Featherweight VML can be used to abstract any graph-like model (e.g. class diagrams, object diagrams, transition systems etc.). Even textual representations of base models such as source code can be abstracted to graphs if we can parse the code into abstract syntax trees.

However, there are some limitations to Featherweight VML. As any generic representation, it fails to capture some of the more specific language contracts from real variability modeling languages. For example, in Delta Modeling a delta module is guarded by an application condition over a set of features while Featherweight VML fragment substitutions are each mapped to a single feature. In order to express a Delta model without adding extra concepts we would have to change Featherweight VML's mapping function to a more general expression. Another example it that the graphs used to represent base models in Featherweight VML are untyped. Essentially they can only capture the structure of the base models. In order to represent other data types, Featherweight VML would have to be extended.

Problem 1.2. *Not all variability modeling languages have a formal specification, making it impossible to prove the correctness of the variant derivation algorithm.*

The ideal solution to verifying tools that implement variability modeling languages and variant derivation is to provide every language with a complete formal specification that can be encoded in a theorem prover. Through Featherweight VML I attempt to provide a lightweight formalization technique that can be used almost off-the-shelf.

Featherweight VML has a complete formal specification. As a proof-of-concept, I have embedded part of Featherweight VML in the semantic framework of the Coq theorem proving system and I have proven the correctness of the variant derivation algorithm. A crucial feature of the semantics is that it is confluent. This was achieved by identifying sufficient conditions for confluence, and adopting a copying style for the definition of semantics, to minimize dependencies between executions of individual variation points.

Because Featherweight VML can represent core concepts from other variability modeling languages, any language that shares these concepts can automatically benefit from the formal specification of Featherweight VML by designing a lifting function, which is an abstraction of the real language. To demonstrate this, I have implemented a lifting function from the Base-Variability-Resolution (BVR) language to Featherweight VML. The size of the lifting tool source code was three orders of magnitude smaller than that of the BVR implementation itself, making it much easier to verify with off-the-shelf tools.

The above technique has one caveat: since the lifting tool is an abstraction, the parts of BVR that could not be completely represented in Featherweight VML are not being included in the formal specification of Featherweight VML.

Problem 1.3. *The complexity of the variant derivation implementations, combined with the possible lack of formal specification of the derivation algorithm, increases the difficulty of proving the correctness of the tools.*

By lifting a real variability modeling language to Featherweight VML, thus obtaining a formal specification of the variant derivation algorithm, we can use this specification to reason about the correctness of variant derivation tools. In my thesis I propose using translation validation, a technique that can validate each individual execution of the tool. It uses the lifting function to convert the input and output of the tool to Featherweight VML and then verifies their correctness by applying the Coq implementation of Featherweight VML. Since Coq supports automatic generation of formally verified implementations (in Haskell and OCaml) out of type and function definitions so the simulation is completely formalized and proven correct.

Since the lifting is an abstraction, some information from the input and output may be lost (such as attribute values of objects) which means that no concrete execution can be created automatically from an abstract execution. This use of abstraction is crucial to the success of the method. It allows implementing and growing validators incrementally, without falling into a trap of diminishing returns.

Once the lifting operation is implemented for a variability language, the validation technique can be applied to all projects in which the variability language is used. The nature of the product line or even of the architectural language does not influence the validation technique, meaning that the abstraction to Featherweight VML is a one-time cost. Maintenance of the abstraction is required only in case the variability language itself changes which, in our experience, does not happen very often. In case the changes to the variability language do not influence the derivation algorithm (e.g. it expands to work with a new architectural language) there are, again, no costs to the abstraction. If the changes affect the derivation process (e.g. a new feature is added) then the maintenance implies abstracting the new constructs, which can be defined as syntactic sugar in terms of fragment substitutions. In such cases it is difficult to give clear estimates of the cost, but we believe it is safe to assume that the changes to the abstraction are directly proportional to the changes of the variability language. The translation validation technique is especially useful when creating trustworthy tools for developing safety critical systems. However, it can be applied with no extra costs to any kind of project, as long as the same variability modeling language is used.

The prototype implementation has demonstrated that Featherweight VML can be easily mapped to real variability modeling languages such as BVR. However, no experiment has been made to assess the scalability of the prototype code, and the initial tests have indicated that the mode in which Coq handle set operations can lead to a bottle-neck in the execution of the validation.

Future work. One issue that requires further investigation is the performance of the validation. Since non safety-critical projects tend to have larger and less optimized code-bases, I expect a time increase for the verified formal execution and for the equivalence check between the simulation result and the actual output model.

Another aspect that should be investigated is the possibility of expanding Featherweight VML in order to capture more concepts of variability modeling. Also other languages beside CVL/BVR, Delta Modeling and OVM should be investigated to confirm that the core concepts handled by Featherweight VML have a wide applicability.

8.2 Challenge 2: Trustworthy reengineering of software product lines.

Problem 2.1. *Depending on the size and complexity of the programming languages in which the input and the output code is written, a code rewriting transformation may be very difficult and costly to implement, and it may even be impossible to reach completeness.*

I present lessons learned from designing and implementing a code modernization transformation. The goal is to transform approximately 4000 C++ functions (some of them containing variability) into corresponding declarative expressions that are side-effect free. The correctness criterion is that the behavior of the input function is preserved (semantic equivalence). Instead of implementing a complete transformation that would work on any arbitrary input, the production costs and time are reduced by focusing on the code at hand. I report on a number of challenges and design decisions such as sacrificing generality, adopting a fail-fast programming approach and tool selection.

Problem 2.2. *The practical aspects of implementing a rule-based transformation often make it impossible produce a proof of correctness of the implementation.*

Verifying the correctness of the transformation implies checking the semantic equivalence of the output models with the input functions, which is generally a difficult problem. Additionally it was impossible to reason about the transformation inductively (rule-by-rule), because the intermediate results are incorrect by design.

Instead, the transformation is verified using a translation validation approach. The technique validates concrete translations instead of the transformation tool or algorithm. Both the input and the output of the transformation are symbolically executed with KLEE, which produces a semantic model in the form of path conditions. KLEE uses an SMT solver to prove the equivalence of the path conditions and provides a counter example when they are not equivalent.

Moreover, the method is oblivious to the complexity of the transformation language, but since it is property driven, it depends strongly on the properties of the transformed languages. In contrast, a white-box method would be vulnerable to both kinds of complexity. The validator finds many semantic bugs that have been missed by unit tests of an experienced transformation developer. These errors would be very difficult to find without verification. For each bug, the tool provides a counter-example consisting of execution paths on which the input and transformed programs differ. This way, we have obtained helpful debugging information, which can be used to improve and correct the transformation. The identified bugs are grouped into seven classes. This work is, to the best of my knowledge, the first ever study reporting errors from a realistic transformation project, including validation using real bugs (as opposed to planted bugs) and operational semantic properties of input and output (as opposed to syntactic and typing properties).

Future work. The lessons generated in this modernization project should be assessed and confirmed for other code-rewriting transformations.

Additionally, since the transformation and verification techniques were developed for a sample of the code base (i.e. the complete project is much larger than 4000 functions) it would be useful to attempt to transform the entire code base. This could lead to more insights on the effectiveness of the transformation and on whether the trade-off between completeness and costs was indeed balanced.

8.3 Challenge 3: Lightweight tools for quality assurance in the presence of variability.

Problem 3. *Most of the state-of-the-art analysis tools cannot handle variability. Lifting an analysis tool to product line level is a costly task which is often performed after the single product analysis tool has reached a certain stage. This means that state-of-the-art analysis tools are not easily available for software product line development.*

One approach to verifying software product lines is to use variability-aware analysis tools. However, there are not many such tools available, as most implementations of analysis techniques cannot handle variability (i.e. multiple variant simultaneously). Instead I propose a set of variability-related transformations to translate program families into single programs without variability. The transformed programs can then be effectively analyzed using various single-program analyzers. The evaluation confirms that some interesting variability bugs can be found in real-world C programs in this way.

The main limitation of this transformation based approach is that it may not produce conclusive results for all individual variants, thus losing some precision. This is due to the fact that the transformed program contains all possible paths that may occur in any variant. However, the precision loss depends on the particular analysis in use. Consider the case of model checking. Since (single-system) model checkers stop once a single counter-example is found in the model, we can use this approach to find a variability bug which occurs in some subset of valid variants but we will not be able to give conclusive results (whether the given property is satisfied or not) for the rest of the valid variants. To overcome this issue, we may repeat our technique on the remaining variants for which no conclusive results were reported in the previous iteration.

Future work. The evaluation should be extended to include consider more verification oracles. This will provide a better assessment of the effectiveness of the reconfiguration technique. Testing the technique on different practical case studies (one suitable target is the Linux kernel) would also help making the tool more generic and prepare it for handling arbitrary inputs.

Bibliography

[1] Rick Flores, Charles Krueger, and Paul Clements. “Mega-scale Product Line Engineering at General Motors”. In: *Proceedings of the 16th International Software Product Line Conference - Volume 1*. SPLC ’12. Salvador, Brazil: ACM, 2012, pp. 259–268. ISBN: 978-1-4503-1094-9. DOI: 10.1145/2362536.2362571. URL: http://doi.acm.org/10.1145/2362536.2362571.

[2] Alexandru F. Iosif-Lazăr and Andrzej Wąsowski. “Trustworthy variant derivation with translation validation for safety critical product lines”. In: *Journal of Logical and Algebraic Methods in Programming* 85.6 (2016), pp. 1154–1176. ISSN: 2352-2208. DOI: http://dx.doi.org/10.1016/j.jlamp.2016.02.001. URL: http://www.sciencedirect.com/science/article/pii/S2352220816000146.

[3] Alexandru F. Iosif-Lazar, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wasowski. “Experiences from Designing and Validating a Software Modernization Transformation (E)”. In: *Automated Software Engineering, ASE 2015, 30th IEEE/ACM International Conference on, Lincoln, NE, USA, November 9-13, 2015*. Ed. by Myra B. Cohen, Lars Grunske, and Michael Whalen. IEEE Computer Society, 2015, pp. 597–607. DOI: 10.1109/ASE.2015.84. URL: http://dx.doi.org/10.1109/ASE.2015.84.

[4] Alexandru F. Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. “Effective Analysis of C Programs by Rewriting Variability”. In: vol. 1. 1. arXiv preprint arXiv:1701.08114. 2017.

[5] Alexandru F. Iosif-Lazar, Ina Schaefer, and Andrzej Wasowski. “Towards A Core Language for Separate Variability Modeling”. In: *25th Nordic Workshop on Programming Theory. NWPT 2013. Tallinn, Estonia, 20-22 November 2013. Abstracts*. Ed. by Tarmo Uustalu and Jüri Vain. Tallinn, Estonia: Institute of Cybernetics at Tallinn University of Technology, 2013, pp. 37–39. ISBN: 978-9949-430-70-3. URL: http://cs.ioc.ee/nwpt13/abstracts-book/paper12.pdf.

[6] Alexandru F. Iosif-Lazăr, Ina Schaefer, and Andrzej Wąsowski. “A Core Language for Separate Variability Modeling”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–272. ISBN: 978-3-662-45234-9. DOI: 10.1007/978-3-662-45234-9_19. URL: http://dx.doi.org/10.1007/978-3-662-45234-9_19.

[7] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002. ISBN: 0201703327.

[8] Krzysztof Czarnecki and Michał Antkiewicz. "Mapping Features to Models: A Template Approach Based on Superimposed Variants". In: *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. GPCE'05. Tallinn, Estonia: Springer-Verlag, 2005, pp. 422–437. ISBN: 3-540-29138-5, 978-3-540-29138-1. DOI: 10.1007/11561347_28. URL: http://dx.doi.org/10.1007/11561347_28.

[9] Danilo Beuche and Robert Hellebrand. "Using Pure::Variants Across the Product Line Lifecycle". In: *Proceedings of the 19th International Conference on Software Product Line*. SPLC '15. Nashville, Tennessee: ACM, 2015, pp. 352–354. ISBN: 978-1-4503-3613-0. DOI: 10.1145/2791060.2791114. URL: http://doi.acm.org/10.1145/2791060.2791114.

[10] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. "FeatureIDE: An Extensible Framework for Feature-oriented Software Development". In: *Sci. Comput. Program.* 79 (Jan. 2014), pp. 70–85. ISSN: 0167-6423. DOI: 10.1016/j.scico.2012.06.002. URL: http://dx.doi.org/10.1016/j.scico.2012.06.002.

[11] Juliana Alves Pereira, Kattiana Constantino, and Eduardo Figueiredo. "A Systematic Literature Review of Software Product Line Management Tools". In: *Software Reuse for Dynamic Systems in the Cloud and Beyond: 14th International Conference on Software Reuse, ICSR 2015, Miami, FL, USA, January 4-6, 2015. Proceedings*. Ed. by Ina Schaefer and Ioannis Stamelos. Cham: Springer International Publishing, 2014, pp. 73–89. ISBN: 978-3-319-14130-5. DOI: 10.1007/978-3-319-14130-5_6. URL: http://dx.doi.org/10.1007/978-3-319-14130-5_6.

[12] CVL Joint Submission Team. *Common Variability Language (CVL). OMG Revised Submission, OMG document: ad/2012-08-05*. OMG, 2012.

[13] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. "Delta-Oriented Programming of Software Product Lines". In: *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proc.* Ed. by Jan Bosch and Jaejoon Lee. Vol. 6287. Lecture Notes in Computer Science. Springer, 2010, pp. 77–91. DOI: 10.1007/978-3-642-15579-6_6.

[14] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005. ISBN: 978-3-540-24372-4. DOI: 10.1007/3-540-28901-1.

[15] Andreas Classen, Quentin Boucher, and Patrick Heymans. "A text-based approach to feature modelling: Syntax and semantics of TVL". In: *Sci. Comput. Program.* 76.12 (2011), pp. 1130–1143. DOI: 10.1016/j.scico.2010.10.005.

[16] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. "Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled". In: *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*. Ed. by Brian A. Malloy, Steffen Staab, and Mark van den Brand. Vol. 6563. Lecture Notes in Computer Science. Springer, 2010, pp. 102–122. DOI: 10.1007/978-3-642-19440-5_7.

[17] Timo Asikainen, Timo Soininen, and Tomi Männistö. "A Koala-based ontology for configurable software product families". In: *IJCAI 2003 Configuration workshop*. 2003, pp. 45–52. URL: http://www.soberit.hut.fi/pdmg/papers/ASIK03KOA.pdf.

[18] Erik D. Demaine. "C to Java: Converting Pointers into References". In: *Concurrency - Practice and Experience* 10.11-13 (1998), pp. 851–861. DOI: 10.1002/(SICI)1096-9128(199809/11)10:11/13<851::AID-CPE385>3.0.CO;2-K. URL: http://dx.doi.org/10.1002/(SICI)1096-9128(199809/11)10:11/13<851::AID-CPE385>3.0.CO;2-K.

[19] Andrey A. Terekhov and Chris Verhoef. "The Realities of Language Conversions". In: *IEEE Softw.* 17.6 (Nov. 2000), pp. 111–124. ISSN: 0740-7459. DOI: 10.1109/52.895180. URL: http://dx.doi.org/10.1109/52.895180.

[20] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. "Variability-aware parsing in the presence of lexical macros and conditional compilation". In: *OOPSLA'11*. ACM, 2011, pp. 805–824.

[21] Paul Gazzillo and Robert Grimm. "SuperC: parsing all of C by taming the preprocessor". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, 2012*. 2012, pp. 323–334. DOI: 10.1145/2254064.2254103. URL: http://doi.acm.org/10.1145/2254064.2254103.

[22] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. "Type checking annotation-based product lines". In: *ACM Trans. Softw. Eng. Methodol.* 21.3 (2012), p. 14.

[23] Sheng Chen, Martin Erwig, and Eric Walkingshaw. "An error-tolerant type system for variational lambda calculus". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*. 2012, pp. 29–40. DOI: 10.1145/2364527.2364535. URL: http://doi.acm.org/10.1145/2364527.2364535.

[24] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. "Intraprocedural Dataflow Analysis for Software Product Lines". In: *Transactions on Aspect-Oriented Software Development* 10 (2013), pp. 73–108.

[25] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. "SPL$^{\mathrm{LIFT}}$: Statically Analyzing Software Product Lines in Minutes Instead of Years". In: *ACM SIGPLAN Conference on PLDI '13*. 2013, pp. 355–364.

[26] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. "Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking". In: *IEEE Trans. Software Eng.* 39.8 (2013), pp. 1069–1089. DOI: 10.1109/TSE.2012.86. URL: http://doi.ieeecomputersociety.org/10.1109/TSE.2012.86.

[27] Aleksandar S. Dimovski. "Symbolic Game Semantics for Model Checking Program Families". In: *Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings*. Vol. 9641. LNCS. Springer, 2016, pp. 19–37.

[28] Mirko Conrad, Patrick Munier, and Frank Rauch. *Software Tool Qualification According to ISO 26262*. Warrendale, Penn., 2011.

[29] Fredrik Asplund, Jad El-khoury, and Martin Törngren. "Qualifying Software Tools, a Systems Approach". In: *Computer Safety, Reliability, and Security: 31st International Conference, SAFECOMP 2012, Magdeburg, Germany, September 25-28, 2012. Proceedings*. Ed. by Frank Ortmeier and Peter Daniel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 340–351. ISBN: 978-3-642-33678-2. DOI: 10.1007/978-3-642-33678-2_29. URL: http://dx.doi.org/10.1007/978-3-642-33678-2_29.

[30] Paul Istoan, Jacques Klein, Gilles Perrouin, and Jean-Marc Jezequel. "A Metamodel-based Classification of Variability Modeling Approaches". In: *Variability for You: Proceedings of VARY International Workshop affiliated with ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MODELS'11)*. Ed. by Øystein Haugen, Krzysztof Czarnecki, Jean-Marc Jezequel, Birger Møller Pedersen, and Andrzej Wasowski. IT Unviersity Technical Report Series. Denmark: IT-Universitetet i København, 2011.

[31] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU SEI, 1990.

[32] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. "A comparison of decision modeling approaches in product lines". In: *Fifth International Workshop on Variability Modelling of Software-Intensive Systems, Namur, Belgium, January 27-29, 2011. Proc.* ACM International Conference Proc. Series. ACM, 2011, pp. 119–126. DOI: 10.1145/1944892.1944907.

[33] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. "Cool features and tough decisions: a comparison of variability modeling approaches". In: *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proc.* ACM, 2012, pp. 173–182. DOI: 10.1145/2110147.2110167.

[34] Christian Prehofer. "Feature-oriented programming: A new way of object composition". In: *Concurrency and Computation: Practice and Experience* 13.6 (2001), pp. 465–501. DOI: 10.1002/cpe.583.

[35] Krzysztof Czarnecki and Michal Antkiewicz. "Mapping Features to Models: A Template Approach Based on Superimposed Variants". In: *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proc.* Ed. by Robert Glück and Michael R. Lowry. Vol. 3676. Lecture Notes in Computer Science. Springer, 2005, pp. 422–437. ISBN: 3-540-29138-5. DOI: 10.1007/11561347_28.

[36] Mikolás Janota and Goetz Botterweck. "Formal Approach to Integrating Feature and Architecture Models". In: *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Budapest, Hungary, March 29-April 6, 2008. Proc.* Ed. by José Luiz Fiadeiro and Paola Inverardi. Vol. 4961. Lecture Notes in Computer Science. Springer, 2008, pp. 31–45. ISBN: 978-3-540-78742-6. DOI: 10.1007/978-3-540-78743-3_3.

[37] Krzysztof Czarnecki and Simon Helsen. "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3 (2006), pp. 621–646. DOI: 10.1147/sj.453.0621.

[38] Mathieu Acher, Philippe Collet, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. "Modeling Variability from Requirements to Runtime". In: *16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, Las Vegas, Nevada, USA, 27-29 April 2011*. IEEE Computer Society, 2011, pp. 77–86. DOI: 10.1109/ICECCS.2011.15.

[39] David Benavides, Antonio Ruiz Cortés, Pablo Trinidad, and Sergio Segura. "A Survey on the Automated Analyses of Feature Models". In: *XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2006), Octubre 3-6, 2006, Sitges, Spain*. Ed. by José Cristóbal Riquelme Santos and Pere Botella. 2006, pp. 367–376.

[40] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. "Generic semantics of feature diagrams". In: *Computer Networks* 51.2 (2007), pp. 456–479. DOI: `10.1016/j.comnet.2006.08.008`.

[41] Mikolás Janota and Joseph Kiniry. "Reasoning about Feature Models in Higher-Order Logic". In: *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proc.* IEEE Computer Society, 2007, pp. 13–22. DOI: `10.1109/SPLINE.2007.36`.

[42] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. "Feature-to-Code Mapping in Two Large Product Lines". In: *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proc.* Ed. by Jan Bosch and Jaejoon Lee. Vol. 6287. Lecture Notes in Computer Science. Springer, 2010, pp. 498–499. DOI: `10.1007/978-3-642-15579-6_48`.

[43] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. "SPL$^{\text{LIFT}}$: statically analyzing software product lines in minutes instead of years". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 2013, pp. 355–364. DOI: `10.1145/2491956.2491976`.

[44] Krzysztof Czarnecki and Krzysztof Pietroszek. "Verifying feature-based model templates against well-formedness OCL constraints". In: *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proc.* ACM, 2006, pp. 211–220. DOI: `10.1145/1173706.1173738`.

[45] Øystein Haugen. "CVL: common variability language or chaos, vanity and limitations?" In: *The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa , Italy, January 23 - 25, 2013*. ACM, 2013, p. 1. DOI: `10.1145/2430502.2430504`.

[46] Florian Heidenreich, Jan Kopcsek, and Christian Wende. "FeatureMapper: mapping features to models". In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*. ACM, 2008, pp. 943–944. DOI: `10.1145/1370175.1370199`.

[47] Robert L. Akers, Ira D. Baxter, Michael Mehlich, Brian J. Ellis, and Kenn R. Luecke. "Case study: Re-engineering C++ component models via automatic program transformation". In: *Information & Software Technology* 49.3 (2007), pp. 275–291. DOI: `10.1016/j.infsof.2006.10.012`. URL: `http://dx.doi.org/10.1016/j.infsof.2006.10.012`.

[48] I.D. Baxter, C. Pidgeon, and M. Mehlich. "DMS ®: program transformations for practical scalable software evolution". In: *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. 2004, pp. 625–634. DOI: `10.1109/ICSE.2004.1317484`.

[49] Jon Siegel, ed. *CORBA 3 fundamentals and programming*. 2nd. New York: OMG Press, John Wiley & Sons, 2000.

[50] Gehan M. K. Selim, Shige Wang, James R. Cordy, and Juergen Dingel. "Model transformations for migrating legacy deployment models in the automotive industry". In: *Software and System Modeling* 14.1 (2015), pp. 365–381. DOI: 10.1007/s10270-013-0365-1. URL: http://dx.doi.org/10.1007/s10270-013-0365-1.

[51] Gehan M. K. Selim, Fabian Büttner, James R. Cordy, Jürgen Dingel, and Shige Wang. "Automated Verification of Model Transformations in the Automotive Industry". In: *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*. Ed. by Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke. Vol. 8107. Lecture Notes in Computer Science. Springer, 2013, pp. 690–706. ISBN: 978-3-642-41532-6. DOI: 10.1007/978-3-642-41533-3_42. URL: http://dx.doi.org/10.1007/978-3-642-41533-3_42.

[52] Gehan M. K. Selim, Levi Lucio, James R. Cordy, Jürgen Dingel, and Bentley J. Oakes. "Specification and Verification of Graph-Based Model Transformation Properties". In: *Graph Transformation - 7th International Conference, ICGT 2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings*. Ed. by Holger Giese and Barbara König. Vol. 8571. Lecture Notes in Computer Science. Springer, 2014, pp. 113–129. ISBN: 978-3-319-09107-5. DOI: 10.1007/978-3-319-09108-2_8. URL: http://dx.doi.org/10.1007/978-3-319-09108-2_8.

[53] Stefan Bunzel. "AUTOSAR - the Standardized Software Architecture". In: *Informatik Spektrum* 34.1 (2011), pp. 79–83. DOI: 10.1007/s00287-010-0506-7. URL: http://dx.doi.org/10.1007/s00287-010-0506-7.

[54] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. "ATL: A model transformation tool". In: *Sci. Comput. Program.* 72.1-2 (2008), pp. 31–39. DOI: 10.1016/j.scico.2007.08.002. URL: http://dx.doi.org/10.1016/j.scico.2007.08.002.

[55] Bruno Barroca, Levi Lucio, Vasco Amaral, Roberto Félix, and Vasco Sousa. "DSLTrans: A Turing Incomplete Transformation Language". In: *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*. Ed. by Brian A. Malloy, Steffen Staab, and Mark van den Brand. Vol. 6563. Lecture Notes in Computer Science. Springer, 2010, pp. 296–305. ISBN: 978-3-642-19439-9. DOI: 10.1007/978-3-642-19440-5_19. URL: http://dx.doi.org/10.1007/978-3-642-19440-5_19.

[56] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. "Proving optimizations correct using parameterized program equivalence". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. Ed. by Michael Hind and Amer Diwan. ACM, 2009, pp. 327–337. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542513. URL: http://doi.acm.org/10.1145/1542476.1542513.

[57] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. "Compiler Optimization Correctness by Temporal Logic". In: *Higher-Order and Symbolic Computation* 17.3 (2004), pp. 173–206. DOI: 10.1023/B:LISP.0000029444.99264.c0. URL: http://dx.doi.org/10.1023/B:LISP.0000029444.99264.c0.

[58] Fabian Büttner, Marina Egea, and Jordi Cabot. “On Verifying ATL Transformations Using ’off-the-shelf’ SMT Solvers”. In: *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*. Ed. by Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson. Vol. 7590. Lecture Notes in Computer Science. Springer, 2012, pp. 432–448. ISBN: 978-3-642-33665-2. DOI: 10.1007/978-3-642-33666-9_28. URL: http://dx.doi.org/10.1007/978-3-642-33666-9_28.

[59] Fabian Büttner, Marina Egea, Jordi Cabot, and Martin Gogolla. “Verification of ATL Transformations Using Transformation Models and Model Finders”. In: *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*. Ed. by Toshiaki Aoki and Kenji Taguchi. Vol. 7635. Lecture Notes in Computer Science. Springer, 2012, pp. 198–213. ISBN: 978-3-642-34280-6. DOI: 10.1007/978-3-642-34281-3_16. URL: http://dx.doi.org/10.1007/978-3-642-34281-3_16.

[60] Fabian Büttner, Marina Egea, Esther Guerra, and Juan de Lara. “Checking Model Transformation Refinement”. In: *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*. Ed. by Keith Duddy and Gerti Kappel. Vol. 7909. Lecture Notes in Computer Science. Springer, 2013, pp. 158–173. ISBN: 978-3-642-38882-8. DOI: 10.1007/978-3-642-38883-5_15. URL: http://dx.doi.org/10.1007/978-3-642-38883-5_15.

[61] Xiaoliang Wang, Fabian Büttner, and Yngve Lamo. “Verification of Graph-based Model Transformations Using Alloy”. In: *ECEASST* 67 (2014). URL: http://journal.ub.tu-berlin.de/eceasst/article/view/943.

[62] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. “Systematic derivation of correct variability-aware program analyses”. In: *Sci. Comput. Program.* 105 (2015), pp. 145–170. DOI: 10.1016/j.scico.2014.10.002. URL: http://dx.doi:10.1016/j.scico.2015.04.005.

[63] Aleksandar S. Dimovski. “Program verification using symbolic game semantics”. In: *Theor. Comput. Sci.* 560 (2014), pp. 364–379. DOI: 10.1016/j.tcs.2014.01.016. URL: http://dx.doi.org/10.1016/j.tcs.2014.01.016.

[64] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. “Variability Abstractions: Trading Precision for Speed in Family-Based Analyses”. In: *29th European Conference on Object-Oriented Programming, ECOOP ’15*. Vol. 37. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 247–270.

[65] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. “Family-Based Model Checking without a Family-Based Model Checker”. In: *22nd International SPIN Workshop on Model Checking of Software, SPIN ’15*. Vol. 9232. LNCS. Springer, 2015, pp. 282–299.

[66] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. “Efficient family-based model checking via variability abstractions”. In: *STTT* (2016). DOI: 10.1007/s10009-016-0425-2.

[67] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. "Variability encoding: From compile-time to load-time variability". In: *J. Log. Algebr. Meth. Program.* 85.1 (2016), pp. 125–145. DOI: 10.1016/j.jlamp.2015.06.007. URL: http://dx.doi.org/10.1016/j.jlamp.2015.06.007.

[68] H. Post and C. Sinz. "Configuration Lifting: Verification meets Software Configuration". In: *ASE'08*. L´Aquila, Italy: IEEE Computer Society, 2008, pp. 347–350.

[69] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. "Toward variability-aware testing". In: *FOSD '12*. 2012, pp. 1–8.

[70] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. "Strategies for product-line verification: case studies and experiments". In: *35th Intern. Conference on Software Engineering, ICSE '13*. 2013, pp. 482–491.

[71] Thomas Thüm, Ina Schaefer, Martin Hentschel, and Sven Apel. "Family-based deductive verification of software product lines". In: *Generative Programming and Component Engineering, GPCE'12*. ACM, 2012, pp. 11–20. DOI: 10.1145/2371401.2371404. URL: http://doi.acm.org/10.1145/2371401.2371404.

[72] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. "Formalizing cardinality-based feature models and their specialization". In: *Software Process: Improvement and Practice* 10.1 (2005), pp. 7–29. DOI: 10.1002/spip.213.

[73] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. "Abstract delta modeling". In: *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*. ACM, 2010, pp. 13–22. DOI: 10.1145/1868294.1868298.

[74] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. "Abstract Features in Feature Modeling". In: *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*. IEEE, 2011, pp. 191–200. DOI: 10.1109/SPLC.2011.53.

[75] Amir Pnueli, Michael Siegel, and Eli Singerman. "Translation Validation". In: *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Lisbon, Portugal, March 28 - April 4, 1998, Proc.* Ed. by Bernhard Steffen. Vol. 1384. Lecture Notes in Computer Science. Springer, 1998, pp. 151–166. DOI: 10.1007/BFb0054170.

[76] Raghava Rao Mukkamala, Andrzej Wasowski, Thorsten Berger, Alexandru Florin Iosif-Lazăr, Stefan Stanciulescu, Remy Haemmerle, Vanstraelen Geert, Antti Välimäki, Øystein Haugen, Daisuke Shimbara, Martin F. Johansen, Maud van den Broeke, Jukka Kääriäinen, and Susanna Teppola. *D4.7 Variability Analysis Solutions*. Tech. rep. The VARIES Consortium, 2015. URL: http://www.varies.eu/wp-content/uploads/sites/8/2013/05/VARIES_D4.7_v01_PU_FINAL.pdf.

[77] Lennart C. L. Kats and Eelco Visser. "The spoofax language workbench: rules for declarative specification of languages and IDEs". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Ri-

nard. ACM, 2010, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497. URL: http://doi.acm.org/10.1145/1869459.1869497.

[78] James R. Cordy. "The TXL source transformation language". In: *Sci. Comput. Program.* 61.3 (2006), pp. 190–210. DOI: 10.1016/j.scico.2006.04.002. URL: http://dx.doi.org/10.1016/j.scico.2006.04.002.

[79] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224. ISBN: 978-1-931971-65-2. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

[80] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. ISBN: 0-7695-2102-9. DOI: 10.1109/CGO.2004.1281665. URL: http://dx.doi.org/10.1109/CGO.2004.1281665.

[81] Iago Abal, Claus Brabrand, and Andrzej Wasowski. "42 variability bugs in the linux kernel: a qualitative analysis". In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14*. ACM, 2014, pp. 421–432. DOI: 10.1145/2642937.2642990. URL: http://doi.acm.org/10.1145/2642937.2642990.

[82] Don Batory. "Feature Models, Grammars, and Propositional Formulas". In: *9th International Software Product Lines Conference, SPLC '05*. Vol. 3714. LNCS. Springer-Verlag, 2005, pp. 7–20.

[83] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective". In: *Formal Asp. Comput.* 27.3 (2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7. URL: http://dx.doi.org/10.1007/s00165-014-0326-7.

[84] *Clang Static Analyzer*. clang: a C language family frontend for LLVM. URL: http://clang-analyzer.llvm.org/.

[85] Florian Merz, Stephan Falke, and Carsten Sinz. "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR". In: *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Proceedings*. Vol. 7152. LNCS. Springer, 2012, pp. 146–161. DOI: 10.1007/978-3-642-27705-4_12. URL: http://dx.doi.org/10.1007/978-3-642-27705-4_12.

[86] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Christian Kästner, and Sven Apel. "An empirical study on configuration-related bugs". In: *Submitted for publication at IEEE TSE* (2016).

Paper A

Journal of Logical and Algebraic Methods in Programming 85 (2016) 1154–1176

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Trustworthy variant derivation with translation validation for safety critical product lines [☆]

Alexandru F. Iosif-Lazăr [*,1], Andrzej Wąsowski [1]

IT University of Copenhagen, Denmark

A R T I C L E I N F O

Article history:
Received 16 May 2014
Received in revised form 28 January 2016
Accepted 4 February 2016
Available online 10 February 2016

A B S T R A C T

Software product line (SPL) engineering facilitates development of entire families of software products with systematic reuse. Model driven SPLs use models in the design and development process. In the safety critical domain, validation of models and testing of code increases the quality of the products altogether. However, to maintain this trustworthiness it is necessary to know that the SPL tools, which manipulate models and code to derive concrete product variants, do not introduce errors in the process.

We propose a general technique of checking correctness of product derivation tools through translation validation. We demonstrate it using Featherweight VML—a core language for separate variability modeling relying on a single kind of variation point to define transformations of artifacts seen as object models. We use Featherweight VML with its semantics as a correctness specification for validating outputs of a variant derivation tool. We embed this specification in the theorem proving system Coq and develop an automatic generator of correctness proofs for translation results within Coq. We show that the correctness checking procedure is decidable, which allows the trustworthy proof checker of Coq to automatically verify runs of a variant derivation tool for correctness.

We demonstrate how such a simple validation system can be constructed, by using this to validate variant derivation of a simple variability model implementation based on the Eclipse Modeling Framework. We hope that this presentation will encourage other researchers to use translation validation to validate more complex correctness properties in handling variability, as well as demonstrate to commercial tool vendors that formal verification can be introduced into their tools in a very lightweight manner.

1. Introduction

Variability modeling in software product lines. Model-driven development [1] of software products employ models to represent the product architecture. When several products share a common set of core assets they can be developed as a software product line [2]. Modeling the product line architecture as a single *base model* facilitates the *derivation* of new *product variants* by reusing artifacts from existing ones. Variability models describe how the artifacts can be selected and recombined into new products.

[☆] This article is a full version of the extended abstract presented at the 25th Nordic Workshop on Programming Theory, NWPT 2013, in Tallinn.

[*] Corresponding author.
E-mail addresses: afla@itu.dk (A.F. Iosif-Lazăr), wasowski@itu.dk (A. Wąsowski).

[1] Supported by ARTEMIS JU under grant agreement No. 295397 and by Danish Agency for Science, Technology and Innovation.

http://dx.doi.org/10.1016/j.jlamp.2016.02.001

Separate variability models are independent of the language in which the base model is developed so they can be reused to some extent to handle a system's variability at multiple development phases. The range of distinguishing characteristics which vary among the products of a product line is specified with feature models [3] (or alternatives such as decision models [4]). Each individual product is described by selecting a set of features thus creating a particular configuration. Constraints and dependencies between features are often specified to determine which configurations are valid. Features are realized by implementation artifacts (e.g. formal specifications, object models, source code) contained in a base model. Thus, we need both a mapping from the feature model to the base model and a process called *variant derivation* through which the artifacts can be selected and recombined into new product variants.

The *Orthogonal Variability Model* (OVM) [5], *Delta Modeling* [6] and the *Common Variability Language* (CVL) [7] are examples of separate variability modeling languages.

Trustworthy variant derivation tools. There is a great variety of tools that implement variability modeling languages and facilitate variant derivation. Trustworthy variant derivation is essential to the development of safety critical embedded systems in domains such as automotive or industrial automation [8,9]. Industrial standards such as IEC 61508 [10] mandate the use of state of the art tools and quality assurance techniques. So far, the industry certifies individual products, or even avoids introducing any variability into safety critical parts of the systems.[2]

Trustworthy variant derivation has two requirements. The first is verifying the product line base model, the variability model and the configuration model trying to identify and report errors introduced by the model designers. The second requirement is verifying the product variant derivation tool to ensure that the derivation process is implemented and executed correctly. While there is a need for good verification methods for both input models and tools, most of the research is focused on the former. The tools that implement variability modeling and variant derivation are usually assumed to be correct. This is partly because the tools employ complex algorithms and depend on external libraries which makes it impossible to formally verify them. Nevertheless, qualification is required for code manipulation tools (such as variant derivation) used in producing code influencing functional safety functions. Our goal is to provide a non-intrusive way of verifying that the output of these tools is produced as prescribed by the input models and to enable usable qualification strategies. We achieve this through *translation validation* [11].

Translation validation recognizes that it might be too challenging to verify a translator (originally a compiler; in our case a variant derivation tool). After all, verifying a translator to be correct once for all, means verifying that it will behave correctly on all possible inputs, which is usually an infinite set with complex properties. In practice, a translator will never be run on the entire set, but on a finite subset. Consequently, it seems wasteful to verify its correctness once for all inputs. With translation validation we do not validate the translator itself, but the output of each execution.

The approach is entirely automatic. Usually the translator is extended to generate a formal proof of correctness of the output with respect to the input. This proof is then checked automatically using an independent proof checker. In the usual scenario, where no bug of translation is uncovered, both tools succeed automatically. In the unlikely case of the translator failing to generate the proof (due to a possible bug) or the proof checker reporting that the proof is incorrect, the user can be warned about the error. This is less convenient than eliminating errors altogether, but prevents the use of erroneous output in a critical system, so the harm is avoided.

The main benefit of this validation method is that developing a translation validator is much simpler than verifying the entire translator. Even a simple variant derivation tool, as discussed in this paper, relies on a number of complex frameworks and libraries (Eclipse Modeling Framework, XML libraries, standard libraries of the programming language, the programming language itself, etc.). Building a formal model of all these elements is unbelievably laborious, whereas translation validation only requires providing a semantic based argument that the output structure is correct with respect to the input structure, independently of how complex the frameworks used in the process are. Thus we believe that translation validation is a viable way of increasing trustworthiness of commercial software tools. In the paper we demonstrate how translation validation can be implemented for variant derivation as an add-on, with minimal changes to the implementation of the tool performing variant derivation.

The translation validation approach is independent of the actual implementation. What it does require is: (i) a common semantic framework for both the input and the output; (ii) a formalization of the notion of *correct execution* (iii) a proof method which, based on the input, allows to automatically verify that the output is correct.

Contributions. In this paper we realize the translation validation approach for an abstract variability modeling language, that is able to capture abstractions of executions of many of the above mentioned modeling notations, in particular of those that subscribe to a separate variability perspective (although the translation validation method in itself does not require separate specification of variability, this is how we scope our demonstration). Our contributions are:

- A core language for separate variability modeling, Featherweight VML, along with an abstract semantics, which is as expressive and versatile as other existing variability modeling languages. We will use it to represent abstractions of concrete variability models.

[2] Personal communication with partners in ARTEMIS projects.

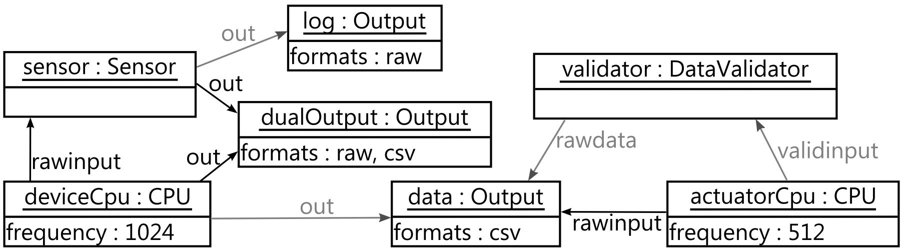

Fig. 1. Example base model for a product line of devices.

- A formal specification of semantics of Featherweight VML, a prerequisite for building a translation validation tool. This captures semantics of relations between features with cardinalities [12] and the base model by copying and flattening the variability model. We also provide a copying semantics for the variant derivation process. We define *two* simple rules for determining which model elements are part of the desired product variant. Compared to in-place model transformations, a copying semantics can more easily be implemented in declarative rule-based model transformation languages and it is easier to reason about using theorem provers. To the best of our knowledge, variability models including both cardinality-based feature modeling and a mapping to implementation artifacts have not been formalized so far.
- A confluence result for our semantics: while other approaches to defining semantics of separate variability modeling languages suggest an implementation by in-place transformations (which makes the order of transformations critical) our rules always produce the same result, independently of the order in which they are applied. Incidentally, this opens for new opportunities to implement variant derivation tools using graph transformations.
- An embedding of the above definitions into the semantic framework of the Coq theorem prover, including a formal mechanically checkable proof of correctness of the embedding.
- A proof of concept translation validation tool for Featherweight VML using the above embedding. Our translation validation strategy is black-box, so it does not require modification of an existing variant derivation tool (in our case a custom in-house tool based on the Eclipse Modeling Framework).

Due to use of the abstraction, our method does not require meticulous formalizations of all the aspects of the variability modeling. The approach allows incremental development. In our demonstration, we instantiate the idea only for connectivity properties of the base model, which keeps the development cost low. If more properties need to be tracked they can be added in subsequent iterations by enriching Featherweight VML and the abstractions.

The paper proceeds as follows. Section 2 provides an analysis of different variability modeling languages in order to determine the core requirements. It also introduces an example of a software product line. Section 3 presents our approach to translation validation. Section 4.1 introduces a minimal representation of base models as graphs. Sections 4.2 and 4.3 describe the formal syntax and semantics of Featherweight VML. Section 5 contains the implementation of Featherweight VML in Coq and describes the validation of a black-box demo tool. We discuss the advantages and limitations of translation validation and also the related work in Sec. 6 and we conclude in Sec. 7.

2. Core requirements for variability modeling

In order to develop a generic method for validating the correctness of variant derivation tools, we require a versatile foundation for variability modeling. To this end, we compare CVL, Delta Modeling and OVM, aiming to find similarities in the way these languages represent and execute variability models over system models. The results help us setting a foundation for defining the syntax and semantics of Featherweight VML. Later we will use Featherweight VML to represent abstractions of variability models during their executions.

2.1. A running example

In Fig. 1 we introduce an example of a product line architecture from which several product variants can be derived. This example will help illustrate the characteristics of OVM, Delta Modeling, CVL and Featherweight VML.

Our system represents a safety critical monitoring device. A minimal product variant is composed of the deviceCpu, the sensor and one of the possible outputs. The deviceCpu receives raw input from the sensor and relays the data as comma-separated values (csv) to the dualOutput to which the sensor also sends the raw data.

Alternatively, sensor may output raw data directly to a log as represented by the gray link out. Yet another possibility is for the deviceCpu to send csv data to a data output which serves as input for an actuatorCpu. As an extra check, the actuatorCpu may compare the raw data with validated input from a validator.

Usually modeling tools require that all elements are contained in an object representing the model root. To keep figures and explanations simple, we ignore the root object in our examples.

2.2. Overview of variability modeling languages

The Orthogonal Variability Model. OVM [5] is designed to handle variability between products. It leaves aside the common parts. It uses *variation points* to specify which characteristics can vary (e.g. color) and *variants* to specify how they vary (e.g. red, blue etc.). Dependency relations between the variation points and variants limit the set of valid configurations. All artifacts are contained in a single model. Both variation points and variants are mapped directly to these artifacts so the solution space does not involve complex transformations. When a configuration is selected (the desired variants are selected for each variation point) the variability model is executed by extracting only the artifacts that the configuration refers to.

In our example, the deviceCpu and the sensor are part of all possible products, thus they do not make the object of the OVM. Instead they are included in all products by default. A variation point VP_1 is needed to specify that the output of the system can vary. This variation point has three variants: V_1 is realized by the dualOutput, V_2 is realized by the log and V_3 is realized by the data and actuatorCpu objects together. Since having both a dualOutput and a log or data output is redundant, we can make the variant pair (V_1, V_2) as well as (V_1, V_3) mutually exclusive.

Another variation point VP_2 specifies that the validator object can vary by having a single optional variant. VP_2 would be dependent of the selection of the variant V_3 for VP_1 since the existence of the validator only makes sense if the data and actuatorCPU are also included in the product. For brevity, we have omitted to show how the variants are referring to the links in the base model (i.e. out, rawinput, rawdata, validinput). Nonetheless, the OVM variants should refer to all artifacts that must be included in the final products.

Delta Modeling. In *Delta Modeling* [6], a product line is represented by a core module and a set of delta modules. The core module provides an implementation of a valid product that can be developed with well-established single application engineering techniques. Delta modules specify changes to be applied to the core module to implement further products by adding, modifying and removing artifacts. Delta Modeling can use any flavor of feature model. Each delta module has an application condition which the configuration must respect in order for the delta to be executed. Delta Modeling can be applied to textual languages, such as the HATS Abstract Behavioral Specification Language [13], or graphical modeling languages, such as Matlab/Simulink [14]. Recently [15], a method has been proposed to systematically deriving a delta language from the grammar of a given base language. Even though Delta Modeling syntax is highly adaptable to the language of the base model, its main concepts remain the same regardless of the implementation.

Listing 1 shows how we can use Delta Modeling to handle the variability in our example. It begins by specifying a list of **features** and a constraint for valid **configurations**. The **core** and **delta** modules add and remove objects and links. The **core** module adds the objects deviceCpu, sensor and dualOutput identified by the object name. It also adds the rawinput link from deviceCpu to sensor and the two out links from deviceCpu and sensor to dualOutput. It is executed when the Dual feature is selected.

The rest of the **delta** modules add and remove objects and links to express the alternative product variants. The **delta** DValidator is executed if both Actuator and Validator features have been selected, but only after the execution of the **delta** DActuator, as specified with the **when** and **after** clauses.

The Common Variability Language. CVL [7] is an industrial attempt to create a generic language that facilitates separate variability modeling for base models specified in any MOF-based language [16]. It employs specialized features called *variability specifications* which can be resolved in particular ways: choices require a yes/no resolution; variables require a value for a specific artifact; classifiers represent features that can be instantiated multiple times in a configuration (similar to features with cardinalities [12]). CVL uses a constraint language to specify constraints over the variability specification tree. Configurations are represented as *resolution models*.

The variability specifications are realized by artifacts which can be manipulated through a wide range of transformations called *variation points.*[3] Among the most common variation points are object/link existence, object substitution (with another object), link-end substitution (substitutes one endpoint with another) or value assignment for object variables. However, most variation points are syntactic sugar as they can be expressed using the fragment substitution. This variation point can replace an entire fragment of the model with another fragment (possibly defined in a separate library).

We illustrate this using our running example. We begin with a base model shown in Fig. 2 where it is considered that the objects deviceCpu, sensor and dualOutput and the links between them are included by default. Fragment substitutions are used to specify additional transformations. A fragment substitution is similar to a delta module. It consists of a *placement* fragment (surrounded by dashed lines) containing the elements that will be removed and a *replacement* fragment (surrounded by solid lines) containing the elements that will be inserted instead. CVL defines fragments by surrounding them with an imaginary closed curve and placing boundary points whenever links cross the curve. Boundary points are elements that fully define all references going in and out of placement/replacement fragments.

To show that dualOutput must be removed we require the boundary points pa and pb pointing to the placement fragment. The boundary point ra marks the entry to the replacement fragment composed of the log object. A binding between pa and ra indicates that the link targeting dualOutput should target log. The actual elements of a fragment are discovered by traversing the model from the entry boundary points. The fragments are fully discovered when all connected elements

[3] CVL and OVM variation points are different concepts.

```
features Dual, Log, Actuator, Validator
configurations Dual ⊕ (Log ∨ (Actuator ∨ (Actuator ∧ Validator)))

core Dual {
  adds objects {deviceCpu, sensor, dualOutput}
  adds links {deviceCpu.rawinput —> sensor, deviceCpu.out —> dualOutput,
    sensor.out —> dualOutput}
}

delta DLog when Log {
  removes objects {dualOutput}
  removes links {deviceCpu.out —> dualOutput, sensor.out —> dualOutput}
  adds objects {log}
  adds links {sensor.out —> log}
}

delta DActuator when Actuator {
  removes objects {dualOutput}
  removes links {deviceCpu.out —> dualOutput, sensor.out —> dualOutput}
  adds objects {data, actuatorCpu}
  adds links {deviceCpu.out —> data, actuatorCpu.rawinput —> data}
}

delta DValidator when (Actuator ∧ Validator) after DActuator {
  adds objects {validator}
  adds links {validator .rawdata —> data, actuatorCpu.validinput —> validator}
}
```

Listing 1. Delta model of the device product line.

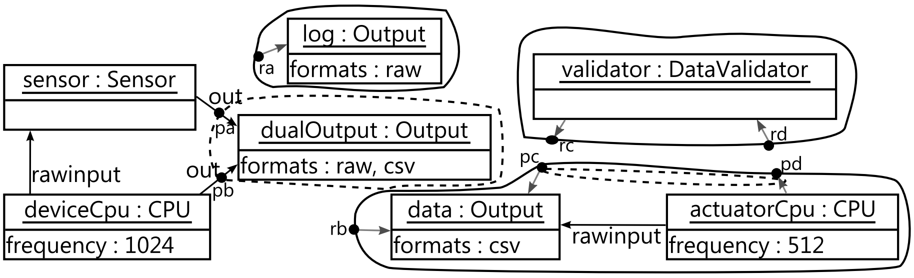

Fig. 2. CVL model of the device product line.

have been reached (the direction of the links is ignored) or when the traversal has been cut off by other boundary points. Traversing the model from rb will retrieve the replacement fragment composed of data, actuatorCpu and the link between them. The following fragment substitutions specify the possible changes to the base model:

$$\mathrm{fs}_1\{\ \textit{placement}\{\mathrm{pa},\mathrm{pb}\}\ \textit{replacement}\{\mathrm{ra}\}\ \textit{binding}\{(\mathrm{pa},\mathrm{ra})\}\ \}$$

$$\mathrm{fs}_2\{\ \textit{placement}\{\mathrm{pa},\mathrm{pb}\}\ \textit{replacement}\{\mathrm{rb}\}\ \textit{binding}\{(\mathrm{pb},\mathrm{rb})\}\ \}$$

$$\mathrm{fs}_3\{\ \textit{placement}\{\mathrm{pc},\mathrm{pd}\}\ \textit{replacement}\{\mathrm{rc},\mathrm{rd}\}\ \textit{binding}\{(\mathrm{pc},\mathrm{rc}),\ (\mathrm{pd},\mathrm{rd})\}\ \}$$

Finally, CVL organizes the features in a variability specification tree and binds the fragment substitutions accordingly as shown in Fig. 3. In this example the *Actuator* is a VClassifier meaning that fs_2 can be executed multiple times to obtain multiple actuators.

2.3. Comparative analysis

Modeling the variations and the configurations is done for all three variability modeling languages using some form of feature models or decision models. OVM is closely related to decision modeling where each variation point is a decision. CVL's variability specification tree is an enhanced feature model with cardinalities [12]. Delta modeling accepts any form of feature or decision model.

Featherweight VML employs feature trees and allows abstract features with no implementation [17]. Also, by employing a constraint language we can define any kind of dependencies between features.

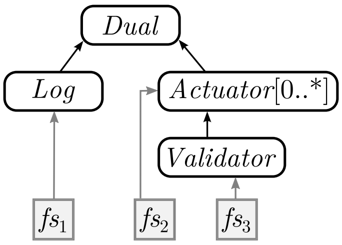

Fig. 3. A variability specification tree.

The realization of features by artifacts is done in multiple ways. OVM uses an annotative approach to mark which artifacts are implementing specific decisions. Delta modeling uses a transformational approach to add, remove and modify artifacts from the model. A delta module's effects can span over the implementation of multiple features so it is not restricted by the structure of a feature tree. CVL variation points, especially the fragment substitution, can define complex transformations. However, they are directly bound to variability specifications so they are constrained by the tree structure. Featherweight VML uses fragment substitutions exclusively. The other CVL variation points, delta modules and of the OVM annotative technique can be reproduced by employing fragment substitutions.

Product derivation requires a clear understanding of how to execute a variability model given a specific configuration. CVL defines how each kind of variation point is executed. The variation points are partially ordered by the resolution tree structure. However, execution is not confluent as two variation points at the same level can have contradictory effects resulting in different variants depending on the order. OVM uses a projection on the model artifacts referenced by the selected variants. Delta Modeling executes each delta module by adding, modifying and removing elements as specified by the modules. The modules also specify a partial order using special clauses. The execution can be made confluent by adding conflict resolving deltas for any pair of conflicting deltas [18].

Orthogonality of variability modeling is the degree to which variability is modeled as a separate concern [19]. CVL defines a clear distinction between feature modeling (via a variability specification tree), and the model transformations over artifacts (via variation points). The variability model is completely separate from the artifacts. OVM design is based on orthogonality. The artifacts can be anything from requirements to model elements or code fragments. Delta Modeling can be applied to any language, textual and graphical alike. Delta modules can use references to artifacts in a separate model to specify what is added, removed and modified. Featherweight VML borrows the layered architecture of CVL as it is general enough to be used with OVM and Delta Modeling.

3. Verifying execution correctness through translation validation

3.1. Correctness properties

Product variant derivation is a process which takes as input *a feature model* over *a base system model* and *a configuration of features*. The output of the derivation is also *a system model* which contains only some of the artifacts of the input base model as specified by the configuration.

The correctness of a tool that implements variant derivation involves several facets (similar to the correctness of a generic model transformation tool [20]):

- termination of the transformation algorithm;
- confluence of the transformation steps (e.g. rule applications, fragment substitutions);
- obtaining the output model prescribed by the input.

The *termination* of the algorithm is essential to obtaining an output model. However, from a functional safety perspective, non-termination of a variant derivation tool is merely a nuisance for the developers of the product, but can't cause unsafe situations for the users of the product (since it is never created). Thus, functional safety standards do not consider termination as a requirement for tool qualification.

Non-*confluent* semantics introduces two problems. First, the product derivation process might be non-deterministic, leading to the possibility of obtaining different outputs on separate executions with the same input. This lowers the efficiency of testing in establishing functional safety. Second, the product derivation tool might be very sensitive to input such that, in spite of a deterministic implementation, the non-confluence is resolved in ad hoc way, not clearly related to input. This makes iterative quality improvement difficult, because small changes to the input model may have unexpected effects on the output.

There are two ways of making a transformation language semantics confluent: either by restricting the legal inputs so that there is no ambiguity (and providing warnings for the users in the case of illegal inputs) or by weakening the semantics in such a way that the non-confluence is hidden (for instance merging all possible outputs). The latter approach can't be used in product derivation, where a single output is needed. Therefore, we follow the former method, ensuring confluence by imposing constraints on the input model, thus eliminating all ambiguity of the specifications and providing a deterministic way of producing a single output, or an error message otherwise.

For product derivation, the confluence of the transformation algorithm can be proven independently of any execution and it holds for all the inputs. It just remains to be checked that the individual output models are consistent with the abstract specification used in this proof.

Additionally to confluence, we also need to verify that the output is as prescribed by the input, which is not a trivial task. This does not imply that the output is correct with respect to any external metamodel. It simply requires that all the specifications of the input model have been respected when producing the output. The problem increases with the complexity of the algorithm, but also with the size of the input. Instead of attempting to prove that the property holds for all executions, we use translation validation to verify the property on each individual resulting output.

Our goal is to verify these correctness properties which attest to the trustworthiness of the derivation tool. They refer to the correctness of the tool itself as opposed to the correctness of the output with respect to other external criteria. To illustrate the difference, we will not verify:

- the conformance of the output model to a language syntax or metamodel;
- the correctness of the output by checking that satisfies any safety properties;
- the introduction of errors resulting from feature interactions.

All the latter properties have been extensively researched, while the correctness of the tool has been largely assumed.

3.2. *Translation validation*

Numerous tools exist for variability modeling with both commercial and academic implementations (e.g. pure::variants,[4] BigLever's Gears,[5] CVL,[6] FeatureMapper,[7] Clafer[8]). These tools are often part of sophisticated integrated development environments. The variant derivation process may use many complex software components, making it very challenging to develop a formal proof of the entire implementation. Even if such a proof could be developed, it would have to be updated whenever the implementation is updated due to bug fixes or addition of new features.

Translation validation [11] is a more pragmatic alternative to verifying translator tools (compilers, code generators). Instead of verifying that the tool *always* produces correct results, we can verify that each result is correct and conforming to the input only for the inputs on which the tool is actually run. The original translation validation method [11] relies on several ingredients:

1. A common semantic framework for the representation of the source code and the generated target code.
2. A formalization of the notion of *correct implementation* as a refinement relation, based on the common semantic framework.
3. A proof method which allows to prove that one model of the semantic framework, representing the produced target code, correctly implements another model which represents the source.
4. Automation of the proof method, to be carried out by an analyzer which, if successful, will also generate a proof script.
5. A rudimentary proof checker that examines the proof script produced by the analyzer and provides the last confirmation for the correctness of the translation.

We adapt the method to SPL variant derivation as illustrated in Fig. 4. The variant derivation tool is considered a black box. It can be an implementation of any variability modeling language. Its input is a variability model accompanied by a particular configuration. Its output is the product variant model.

For the original translation validation method, a *common semantic framework* for the representation of the input and output meant that a new semantic framework would be needed for each different language specification. By using Featherweight VML as the common semantic framework we can reuse the same setup for validating any tool for any language that can be abstracted to Featherweight VML. In Fig. 4 the abstraction is represented by the [lift]ing arrows.

We formalize the notion of *correct implementation* by providing formal semantics for Featherweight VML. We simulate the variant derivation via a formal execution of the abstract input model in Coq. The simulation result is then compared to the abstraction of the actual derivation result. If the two results are equivalent (isomorphic models) then we can say that the product variant model conforms to the input configuration. All this can be done by implementing a simpler tool than the actual derivation tool.

[4] http://www.pure-systems.com/pure_variants.49.0.html.

[5] http://www.biglever.com/solution/product.html.

[6] http://www.omgwiki.org/variability/doku.php.

[7] http://featuremapper.org/.

[8] http://www.clafer.org/.

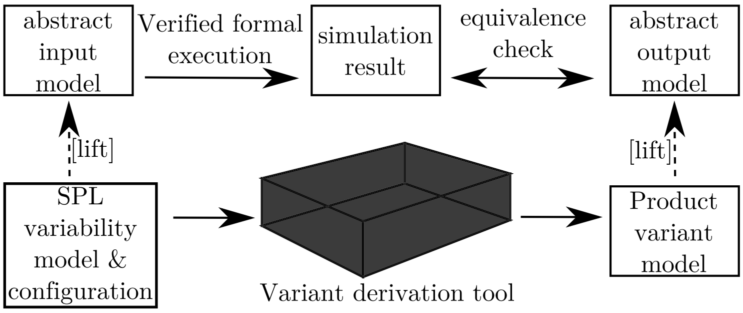

Fig. 4. The translation validation process.

The advantages of simulating the derivation on an abstract model are multiple. The simulation tool can be reused to validate various production tools; the same simulation tool could work for translation validation of both Featherweight VML and CVL. While the actual tools depend on external libraries, the simulation tool can be implemented stand-alone, without dependencies to unverifiable components. Since the simulation is performed on abstractions of the models, the simulation tool can cave a smaller source code and is easier to verify. While most production tools are written in imperative languages, the simulation tool can be written using declarative or functional languages for which it is easier to write proofs of correctness.

We have implemented a simulation tool as a proof of concept (Sec. 5). We used Coq,[9] an interactive theorem proving system implemented on top of a functional language. We implemented Featherweight VML as total functions which are shown to terminate. Then we developed theorems and proved that the simulation executes correctly and is confluent, i.e. the semantics of Featherweight VML is deterministic. Finally we can verify that the result of executing the abstract input model is equivalent to the abstraction of the output model. This approach has the advantage that we do not need to produce proof objects every time we perform the validation. The simulation is verified only once and can be reused on any lifted model. In the following sections we present the different elements of the setup in detail.

4. Featherweight VML

Featherweight VML is designed as a core variability modeling language. It is intended to provide a common framework to which languages such as CVL, Delta Modeling and OVM can be reduced. It is a formally defined language meant to offer a simple, unambiguous view of variability models and variant derivation. Our aim is to use Featherweight VML as a foundation for applying translation validation to variant derivation tools for actual languages (e.g. CVL, Delta Modeling, OVM).

In order to cover the entire variant derivation process, Featherweight VML must provide a syntax for abstracting base system models, a syntax for representing the variability specification and a semantics for variant derivation. While other formal specifications for graph-like model representation and transformation exist [21–25], using a specialized language for variant derivation has advantages not only from usability point of view, but also from a formal point of view: it is much easier to do verification (hereunder translation validation) for a tool that implements a concise formal language than for a big transformation language (e.g. ATL [24]).

4.1. Abstract model representation

Featherweight VML is designed to specify variability in models defined using MOF-based metamodels, consisting of objects and relationships between them. We represent models as multi-graphs of attribute-less, untyped objects connected by directed links. We write $\mathbb{O}$ (respectively $\mathbb{L}$) to denote the infinite universe of all objects (resp. links). Both objects and links are discrete identifiable entities. The links are equipped with endpoint mappings indicating source and target objects: $\mathsf{src}\ l$ and $\mathsf{tgt}\ l$, both total functions of type $\mathbb{L} \to \mathbb{O}$. We assume that the universe of links is complete, in the sense that it contains infinitely many links with unique identities between any two objects in $\mathbb{O}$.

Definition 1. A *model* m is a pair of sets of finitely many objects and finitely many links, $m = (m_{\mathsf{Obj}}, m_{\mathsf{Lnk}}), m_{\mathsf{Obj}} \subseteq \mathbb{O}$, $m_{\mathsf{Lnk}} \subseteq \mathbb{L}$. A *model fragment* is a subset of objects and links of a model, so syntactically it is also a pair $f = (f_{\mathsf{Obj}}, f_{\mathsf{Lnk}})$.

While models and model fragments are syntactically identical, semantically they are different. We use the term *model* to refer to complete systems (i.e. the base model and the product variant). *Model fragments* represent components or incomplete pieces of models. We use *model fragments* as interchangeable units in the definition of fragment substitutions.

[9] http://coq.inria.fr/.

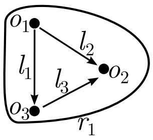

Fig. 5. A fragment.

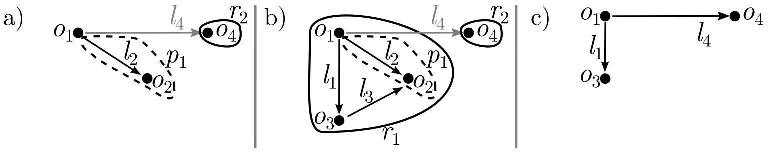

Fig. 6. a) A fragment substitution. b) Fragment interaction. c) The execution result.

We say that a model (or a fragment) m is *closed under links* (or simply *closed*) if for each link $l \in m_{\mathsf{Lnk}}$ its endpoints are contained in the model, so $\mathsf{src}\, l, \mathsf{tgt}\, l \in m_{\mathsf{Obj}}$.

Fig. 5 illustrates a closed model fragment $r_1 = (\{o_1, o_2, o_3\}, \{l_1, l_2, l_3\})$. For the remainder of the paper we lift set operators to fragment operators, e.g. $f_1 \dot{\subseteq} f_2$ means $f_{1\mathsf{Obj}} \subseteq f_{2\mathsf{Obj}} \wedge f_{1\mathsf{Lnk}} \subseteq f_{2\mathsf{Lnk}}$.

4.2. The fragment substitution variation point

We introduce the formal definition of Featherweight VML in two steps: first we explain the execution of fragment substitutions, then we define the entire variability model relating feature models and fragment substitutions.

Syntax of the fragment substitution. Fragment r_1 introduced in Fig. 5 represents a component that can be customized by replacing o_2 with a new object, o_4. In Fig. 6a we define a *placement* fragment, p_1 (enclosed by a dashed line), containing the elements that must be removed from r_1. We also define a new *replacement* fragment, r_2 (enclosed by a solid line), containing the elements that must be added. Finally, we create a new link, l_4 (represented by a gray arrow), that binds r_2 to the rest of the model. The placement and replacement fragments, p_1 and r_2, together with the new link, l_4, constitute a fragment substitution. Fig. 6b shows how the fragment substitution interacts with r_1. After execution we obtain the result shown in Fig. 6c. The link l_3 was removed even though it was not part of the placement fragment, in order to avoid dangling links.

Definition 2. A *fragment substitution fs* is a triple (p, r, b) where p is a placement fragment containing all the elements that must be removed, r is the replacement fragment and b is a set of new links called a *binding*. The placement and replacement fragments are disjoint, $p \dot{\cap} r = (\emptyset, \emptyset)$.

Most variability modeling languages mark a model fragment to be copied by default and form the common base of any product variant (e.g. the core module in Delta Modeling). In order to keep the number of concepts low, in Featherweight VML we use fragment substitutions to represent both the common base and the subsequent changes applied to it. The example in Fig. 7a, b, c, d illustrates a set of fragment substitutions. We assume that we start from an empty model and fs_1 has only a replacement fragment which introduces the common base. The remaining substitutions perform further customization: fs_2 and fs_3 are removing the elements of p_1 and attach two other fragments, r_2 and r_3. The substitution fs_4 attaches a new fragment so its binding links have endpoints in r_3. Fig. 7e represents the interactions between all fragment substitutions in a single base model. Fig. 7f represents the substitutions with the Featherweight VML abstract syntax and Fig. 7g shows the final result.

The placement and replacement fragments are interchangeable units that can be defined independently of any fragment substitution. They can be seen as templates that can be reused. In Fig. 7f the placement fragment p_1 is referenced in both fs_2 and fs_3. Similarly, a replacement fragment can be reused in multiple fragment substitutions. Binding links, on the other hand, are dependent of a particular fragment substitution. They allow the reuse of placement and replacement fragments by changing only the way they connect. This is optimal in the cases where the fragments are large in the number of objects and links, but they connect through a small number of links defined as bindings. In the case when a fragment substitution is required to link objects that have been introduced by previous substitutions, the replacement fragment may be empty and the operation can be performed through binding links exclusively.

We require that for any fragment substitution $fs = (p, r, b)$, the binding links are not incident with placement objects, $\forall l \in b (\mathsf{src}\, l \cup \mathsf{tgt}\, l) \cap p_{\mathsf{Obj}} = \emptyset$. All such links would be removed as dangling since their endpoints belonging to a placement would be removed. Binding links can only be incident with objects from the replacement fragment and boundary objects, defined as follows:

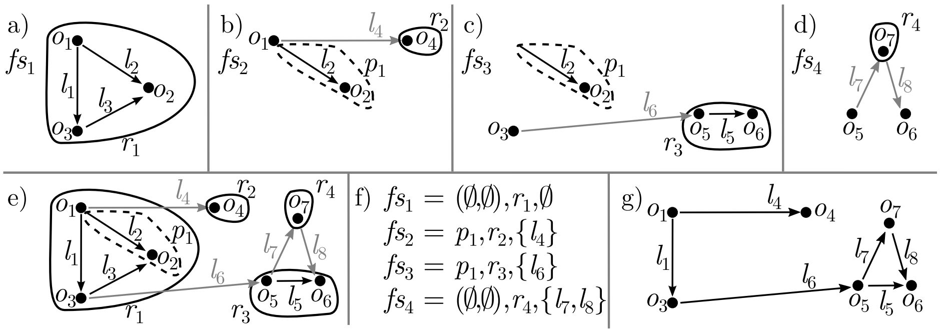

Fig. 7. a, b, c, d) A set of fragment substitutions. e) Interactions between fragment substitutions. f) Syntactic representation. g) The execution result.

Definition 3. The *boundary* of a fragment substitution $fs = (p, r, b)$ is the set of all endpoints of binding links that are not part of the replacement fragment: $\mathsf{boundary}\, fs = \{o \mid o = \mathsf{src}\, l \vee o = \mathsf{tgt}\, l, l \in b\} \setminus r_{\mathsf{Obj}}$.

In Fig. 7 we have $\mathsf{boundary}\, fs_2 = \{o_1\}$, $\mathsf{boundary}\, fs_3 = \{o_3\}$ and $\mathsf{boundary}\, fs_4 = \{o_5, o_6\}$. In Sec. 4.3 we will need to identify all artifacts that a fragment substitution affects outside of its own replacement fragment. These are the artifacts in the placement fragment and the boundary objects used by the binding links.

Definition 4. Given a fragment substitution $fs = (p, r, b)$, the *closure* of the placement fragment p, written $\lceil p \rceil_{fs}$, is defined as all objects of p plus the boundary of the fragment substitution; the set of links remains unchanged: $\lceil p \rceil_{fs} = (p_{\mathsf{Obj}} \cup \mathsf{boundary}\, fs, p_{\mathsf{Lnk}})$.

In Fig. 7, $\lceil p \rceil_{fs_2} = (\{o_1, o_2\}, \{l_2\})$, $\lceil p \rceil_{fs_3} = (\{o_2, o_3\}, \{l_2\})$ and $\lceil p \rceil_{fs_4} = (\{o_5, o_6\}, \emptyset)$. Substitutions fs_2 and fs_3 have different placement closures even if they refer to the same placement fragment. This is because the binding links differ.

Execution semantics of the fragment substitution. The example in Fig. 7 gave the intuition of the fragment substitution execution process. Instead of performing in-place changes to the base model, we propose a copying semantics, meaning that we decide for each object/link whether it should be part of the product variant and we copy only those for which we decide positively.

Given a set of fragment substitutions, *Fs*, we will copy all replacement fragments and all binding links. However, we know that what is contained by placement fragments should be removed and replaced so we will not copy these elements. We will not copy links that are incident with placement fragments either. The result $[\![Fs]\!]$ of executing a set of fragment substitutions *Fs* is called a product variant model; it is a pair of sets of objects/links. The following rules precisely describe which objects and links are copied in $[\![Fs]\!]$:

$$\frac{o \in (\bigcup_{(\_,r,\_)\in Fs} r_{\mathsf{Obj}}) \qquad o \notin (\bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Obj}})}{o \in [\![Fs]\!]_{\mathsf{Obj}}}\ (\text{OBJ-COPY}) \qquad \frac{l \in (\bigcup_{(\_,r,b)\in Fs} r_{\mathsf{Lnk}} \cup b) \qquad l \notin (\bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Lnk}}) \qquad \mathsf{src}\, l, \mathsf{tgt}\, l \notin \bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Obj}}}{l \in [\![Fs]\!]_{\mathsf{Lnk}}}\ (\text{LNK-COPY})$$

The OBJ-COPY rule says that any object contained in a replacement fragment of a fragment substitution in *Fs* will be copied as long as it is not contained in any placement fragment. The LNK-COPY rule says that any link that is contained in a replacement fragment or in a binding set of a fragment substitution in *Fs* will be copied as long as the link or its endpoints are not contained in any placement fragment. The rules are applied exhaustively for all objects and links in all fragments and bindings in the set of fragment substitutions. The complete input model is illustrated in Fig. 7e. Even though individual fragments do not have to be closed under links (see page 1161), the complete input model may be closed. Lemma 1 ensures that applying the rules to a closed input model results in a product variant model without any dangling links.

Lemma 1. *Given a set of fragment substitutions Fs such that the union of all placement fragments, replacement fragments and bindings is a closed model, the product variant model $[\![Fs]\!]$ is also closed under links.*

Proof (Sketch). By assumption, the union of all objects and links is a closed graph, so for every link that might be copied, the graph also contains its endpoints. Then we notice that the premise of (LNK-COPY) is that neither the source or the target of the link being copied are contained in a placement fragment. Thus it is guaranteed that for any link that is being copied, both link ends will also be copied. □

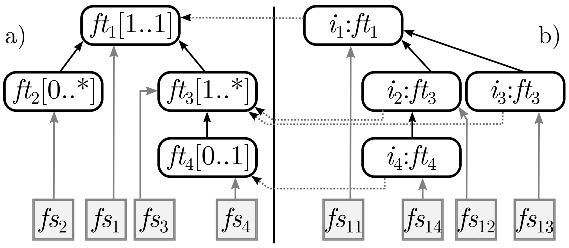

Fig. 8. a) A variability model. b) A configuration and a flattened set of fragment substitutions.

Lemma 2. *Given a set of fragment substitutions, there exists a unique product variant model created by the above rules.*

The lemma holds by construction: objects and links are deterministically selected from a finite set. It follows from the Lemma 2 that the execution of fragment substitution sets is order independent (in other words the semantics is *confluent*), which opens for various implementation strategies.

4.3. The variability model

We have shown how to execute a set of fragment substitutions, *Fs*, to obtain a product variant model. In a normal scenario, we would like *Fs* to describe multiple variants and to be able to select only those fragment substitutions that describe a specific product variant before executing them. We would also like to be able to execute a fragment substitution multiple times and to use a configuration to specify how many copies of the replacement fragment to include in the product variant.

Fig. 8a illustrates a variability model where each fragment substitution, $fs_{1..4}$, is mapped to a feature, $ft_{1..4}$, from a feature tree. Each feature displays a cardinality constraint for how many instances are allowed for that feature under a single parent. In Fig. 8b the features are instantiated in a configuration tree. The root feature has one root instance, feature ft_2 is not instantiated and ft_3 is instantiated twice, meaning that its fragment substitutions should be executed twice. Feature ft_4 is only instantiated as a child of i_2.

Section 4.2 does not handle multiple execution of fragment substitutions. Instead we will show how to flatten the model and the chosen configuration in a set of fragment substitutions that contains as many copies of each fragment substitution as there are instances of its feature. Flattening the model in our example would result in a set containing two copies of fs_3, but no copies for fs_2.

4.3.1. Syntax of the variability model

A feature model defines all characteristics that can be activated in a product variant. Some characteristics may occur multiple times in a product variant (e.g. the number of USB ports on a computer). For this reason, a *feature* in Featherweight VML is similar to a type that can be instantiated multiple times in the product variant so our features have cardinality [12].

Definition 5. A *feature model* is a rooted directed tree of features, $Fm = (Ft, ft_0, \mathsf{parent})$, where Ft is a set of features, $\mathsf{parent} \subseteq Ft \times Ft$ is a connected acyclic parent relation with no sharing (a tree), and $ft_0 \in Ft$ is the root of the tree. We write $\mathsf{parent} ft_2 = ft_1$, if feature ft_1 is a parent node of ft_2 in Fm.

Each feature ft has an associated cardinality constraint $\mathsf{card} ft = (\mathsf{min} ft, \mathsf{max} ft)$, where $\mathsf{min} ft, \mathsf{max} ft \in \mathbb{N} \cup \{*\}$, $\mathsf{min} ft \leq \mathsf{max} ft$ (the symbol $*$ is considered greater than any natural).

A set of fragment substitutions and the feature model that controls which combinations of fragment substitutions can be executed together constitute a complete variability model.

Definition 6. A *variability model* is a triple, $(Fs, Fm, \mathsf{mapping})$, where Fs is a set of fragment substitutions, $Fm = (Ft, ft_0, \mathsf{parent})$ is a feature model and $\mathsf{mapping} : Fs \to Fm$ maps each fragment substitution to a feature.

A configuration represents a combination of features that are active in a product variant.

Definition 7. Given a feature model $Fm = (Ft, ft_0, \mathsf{parent})$, a *configuration* is a rooted tree $Cfg = (I, i_0, \mathsf{parent}, \mathsf{ty})$, where I is a finite set of *instances*, $i_0 \in I$ is the root of the tree, $\mathsf{parent} \subseteq I \times I$ is a connected acyclic parent relation with no sharing (a tree). The typing mapping $\mathsf{ty} : I \to Fm$ maps every instance to its feature, in a manner preserving the parent relations:

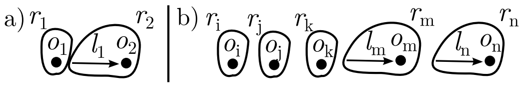

Fig. 9. The replacement fragment problem.

i. The root instance is typed by the root feature: $\mathsf{ty}\, i_0 = ft_0$.
ii. The children of an instance are typed by children of its type: for instances i, j, if $\mathsf{parent}\, i = j$ then $\mathsf{parent}\,(\mathsf{ty}\, i) = \mathsf{ty}\, j$.
iii. The feature cardinality constraints are satisfied, so for each instance $j \in I$ and feature $ft \in Ft$, if $\mathsf{parent} ft = \mathsf{ty}\, j$ then

$$\mathsf{min} ft \leq |\{i \in I \mid \mathsf{parent}\, i = j \text{ and } \mathsf{ty}\, i = ft\}| \leq \mathsf{max} ft$$

Before moving on to the execution semantics we give a set of well-formedness constraints that guarantee that the flattening of variability models produces unique sets of fragment substitutions that can be executed with the rules introduced in Sec. 4.2.

C 1. *The mapping of fragment substitutions to features is injective. Any two fragment substitutions, $fs_i = (p_i, r_i, b_i)$ and $fs_j = (p_j, r_j, b_j)$, that should be mapped to the same feature can be merged into a single fragment substitution, $fs_n = (p_i \dot{\cup} p_j, r_i \dot{\cup} r_j, b_i \cup b_j)$.*

Constraint 1 helps simplifying the following constraints and the semantics. It does not limit the expressive power of Featherweight VML. If fs_i and fs_j should anyway be mapped to the same feature then they should be executed together for each instance of that feature. Thus, requiring that they should be combined into one fragment substitution does not change their effect.

It is not required that every feature has a substitution mapped to it. The inverse $\mathsf{mapping}^{-1} : Ft \to [Fs \cup \{\bot\}]$ returns the fragment substitution mapped to a feature or $\bot$ if such a fragment substitution does not exist.

C 2. *All replacement fragments are closed under links. This constraint enforces that for any link cloned during flattening, its endpoints are also cloned and all the clones will be consistent with the original fragment.*

Fig. 9 illustrates the replacement fragment problem fixed by constraint 2. Assume we have two replacement fragments r_1 and r_2 such that a link form r_2 has an endpoint in r_1. Each fragment is used in a fragment substitution and each substitution is mapped to a different feature. If we instantiate r_1 three times—resulting in fragments r_i, r_j and r_k being inserted in the product variant—and we instantiate r_2 two times—resulting in fragments r_m and r_n—then there is no clear intuition about which of the new objects should be used as endpoints for the new links. In fact, we could even instantiate the links, but not their endpoints.

When a fragment substitution fs_i is instantiated, product line developers intuitively assume that the artifacts of the placement fragment (provided that the fragment is not empty) have already been introduced in the output by the instantiation of another fragment substitution, fs_j. The execution order of these two fragment substitutions is influencing the confluence of the derivation process because fs_i removes and fs_j adds the same artifacts. Featherweight VML is addressing the confluence issue by enforcing constraints on the input variability models.

We recall that the closure of the placement fragment $\lceil p \rceil_{fs_i}$ of a fragment substitution $fs_i = (p_i, r_i, b_i)$ is composed of (i) all the artifacts that will be removed as being part of the placement fragment p_i extended with (ii) all the objects that are endpoints of binding links, but are not newly added by the replacement fragment r_i: $\lceil p \rceil_{fs_i} = (p_{i\mathsf{Obj}} \cup \mathsf{boundary} fs_i, p_{i\mathsf{Lnk}})$. More concisely, the placement closure of a fragment substitution is composed of all the artifacts that are removed or otherwise affected, but are now newly added by the fragment substitution itself. If the placement closure $\lceil p \rceil_{fs_i}$ is not empty and there exists one and only one other fragment substitution $fs_j = (p_j, r_j, b_j)$ such that the artifacts of $\lceil p \rceil_{fs_i}$ are added by fs_j (formally $\lceil p \rceil_{fs_i} \dot{\subseteq} r_j$) we say that fs_i *applies to* fs_j and we write $fs_i \sqsubset fs_j$.

C 3. *All artifacts that are removed or affected in any way by any fragment substitution, must be added by a different fragment substitution. For any fragment substitution $fs_i \in Fs$ for which the placement closure is not empty, $\lceil p \rceil_{fs_i} \dot{\neq} (\emptyset, \emptyset)$, there must exist another fragment substitution $fs_j \in Fs$ such that fs_i applies to fs_j, $fs_i \sqsubset fs_j$.*

Constraint 3 does not in its own enforce the confluence of the semantics. We also need to enforce that if fragment substitution fs_i applies to fs_j, then fs_j is executed first. This is to guarantee that the artifacts are added before removing them or binding to them. Furthermore, in the case that multiple clones are required for both fragment substitutions, the way in which the clones are created must be consistent with the structure of the feature model.

C 4. *The structure enforced by the application, $\sqsubset$, of fragment substitutions is consistent with the feature model: if $fs_i \sqsubset fs_j$ then* $\mathsf{mapping} fs_j \in \mathsf{parent}^*(\mathsf{mapping} fs_i)$*, so if one fragment substitution applies to another, then it is mapped to a feature in the subtree rooted by the feature of the other. Function* parent^* *is the reflexive transitive closure of* parent.

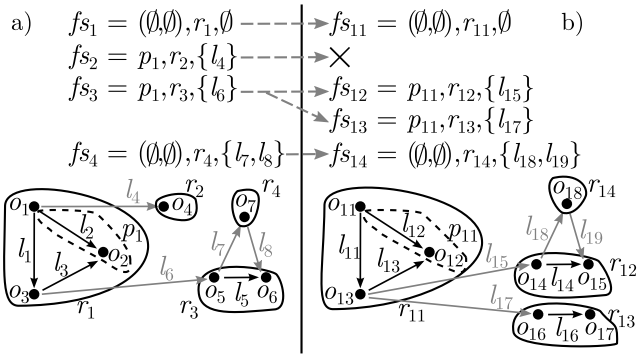
a)
$fs_1 = (\emptyset,\emptyset), r_1, \emptyset$
$fs_2 = p_1, r_2, \{l_4\}$
$fs_3 = p_1, r_3, \{l_6\}$
$fs_4 = (\emptyset,\emptyset), r_4, \{l_7, l_8\}$
b)
$fs_{11} = (\emptyset,\emptyset), r_{11}, \emptyset$
×
$fs_{12} = p_{11}, r_{12}, \{l_{15}\}$
$fs_{13} = p_{11}, r_{13}, \{l_{17}\}$
$fs_{14} = (\emptyset,\emptyset), r_{14}, \{l_{18}, l_{19}\}$
o_1 o_2 o_3 o_4 o_5 o_6 o_7
l_1 l_2 l_3 l_4 l_5 l_6 l_7 l_8
p_1 r_1 r_2 r_3 r_4
o_{11} o_{12} o_{13} o_{14} o_{15} o_{16} o_{17} o_{18}
l_{11} l_{12} l_{13} l_{14} l_{15} l_{16} l_{17} l_{18} l_{19}
p_{11} r_{11} r_{12} r_{13} r_{14}

Fig. 10. Illustration of the flattening process: a) before, b) after.

All constraints must be verified against the input variability model to ensure that an output can be derived deterministically from the variability specification. Once the variability model is flattened, the cloning and renaming of fragment substitutions and artifacts eliminates the dependency on the structure of the feature model. After flattening, the execution order of the fragment substitutions does not affect the output product variant model.

4.3.2. *Execution semantics of the variability model*

In Fig. 10 we recall the fragment substitutions of Fig. 8. On the left side we have the detailed contents of the initial four fragment substitutions. On the right side we have the flattened set. The configuration does not contain an instance for ft_2 so fs_2 is not copied. The substitution fs_3 must be executed two times—once for the instance i_2 and once for i_3. Since the semantics presented in Sec. 4.2 only execute each substitution once, we flatten the model by computing how many times each fragment substitution should be executed and cloning it the appropriate amount of times (carefully updating references).

The semantics is presented as follows: first we introduce functions for copying and renaming basic entities—objects and links. Second, we lift these functions to sets of objects/links, to base model fragments and to fragment substitutions. Third, we explain the flattening of variability models and configurations. We conclude the semantics with a theorem of confluence.

The renaming of objects and links is needed because multiple clones of the same artifact can occur in the same product variant. In the case of a variability model where each feature can only be instantiated once, the renaming is not necessary.

4.3.3. *Preliminaries: copying and renaming basic entities*

Given a variability model, $(Fs, Fm, \mathsf{mapping})$, we use the sets O and L to reference all artifacts contained in this model, $O = \bigcup_{(p,r,b)\in Fs} [p_{\mathsf{Obj}} \cup r_{\mathsf{Obj}}]$ and $L = \bigcup_{(p,r,b)\in Fs} [p_{\mathsf{Lnk}} \cup r_{\mathsf{Lnk}} \cup b]$.

Given a configuration $Cfg = (I, i_0, \mathsf{parent}, \mathsf{ty})$ we use the set I of instances as an index for renaming artifacts. Since the product variant model may end up containing several copies of the same artifacts, we will need to create fresh objects and links, and then be able to refer to them unambiguously. We model this using two injective functions new-obj and new-lnk that create new objects/links for any given feature instance.

$$\mathsf{new\text{-}obj} : I \times O \to \mathbb{O} \setminus O \qquad \mathsf{new\text{-}lnk} : (I \times I) \times L \to \mathbb{L} \setminus L$$

We write the first argument in all renaming functions as an index to make the notation more lightweight. Intuitively, the first argument represents an ordinal index of the copy, whereas the second argument is the entity being copied.

We require that the two functions map to an isomorphic graph structure, so they are injective and for every pair of feature instances i, j (possibly but not necessarily, $i = j$) and any link l we have that: $\mathsf{src}\,(\mathsf{new\text{-}lnk}_{i,j} l) = \mathsf{new\text{-}obj}_i(\mathsf{src}\, l)$ and $\mathsf{tgt}\,(\mathsf{new\text{-}lnk}_{i,j} l) = \mathsf{new\text{-}obj}_j(\mathsf{tgt}\, l)$.

For every instance-object pair we get a different new object, which was not in O. Similarly, for every instance pair (i, j) and a link we get a link, which was not in L, connecting copies of the objects related with $\mathsf{new\text{-}obj}_i$ and $\mathsf{new\text{-}obj}_j$.

We lift the two functions to rename (create) entire sets of objects and links:

$$\mathsf{new\text{-}Obj} : I \times 2^O \to 2^{\mathbb{O}\setminus O}, \text{ where } \mathsf{new\text{-}Obj}_i O' = \{\mathsf{new\text{-}obj}_i o \mid o \in O'\} \text{ and}$$

$$\mathsf{new\text{-}Lnk} : (I \times I) \times 2^L \to 2^{\mathbb{L}\setminus L}, \text{ where } \mathsf{new\text{-}Lnk}_{i,j} L' = \{\mathsf{new\text{-}lnk}_{i,j} l \mid l \in L'\}.$$

Such renaming functions always exist due to our assumption that the universes of objects and links are complete and infinite and we can always obtain a new link between any two objects.

Copying fragments and bindings. We will now explain how to copy a fragment substitution such that all its clones (each clone implementing a different instance) are independent of each other. We lift the simple renaming functions shown above to fragments:

$$\mathsf{new\text{-}frg}_i(O', L') = (\mathsf{new\text{-}Obj}_i O', \mathsf{new\text{-}Lnk}_{i,i} L').$$

In our example we copy the fragment r_3 for the instances i_2 and i_3:

$$\mathsf{new\text{-}frg}_2(\{o_5, o_6\}, \{l_5\}) = (\mathsf{new\text{-}Obj}_2\{o_5, o_6\}, \mathsf{new\text{-}Lnk}_{2,2}\{l_5\}) = (\{o_{14}, o_{15}\}, \{l_{14}\}),$$

$$\mathsf{new\text{-}frg}_3(\{o_5, o_6\}, \{l_5\}) = (\mathsf{new\text{-}Obj}_3\{o_5, o_6\}, \mathsf{new\text{-}Lnk}_{3,3}\{l_5\}) = (\{o_{16}, o_{17}\}, \{l_{16}\}).$$

Renaming bindings is more complex—the endpoints may be renamed differently, according to which fragment they belong to. We formalize binding renaming to take as parameter two disjoint sets of objects. We apply i-renaming if an endpoint is in the first set, and j-renaming if the endpoint is in the other set:

$$\mathsf{new\text{-}bdg}_{i,j}(O_1, O_2, L) = \{\mathsf{new\text{-}lnk}_{\mathsf{ns}(\mathsf{src}\,l),\mathsf{ns}(\mathsf{tgt}\,l)}\,l \mid l \in L\},$$

where ns is a function mapping objects to name spaces (instances), depending on which replacement they belong to; $\mathsf{ns}\,o$ returns i if $o \in O_1$ and it returns j if $o \in O_2$. In our example we want to copy the binding links l_7 and l_8. The ns function allows us to copy the source of l_7 and target of l_8 with the appropriate instance i_2: $\mathsf{new\text{-}bdg}_{4,2}(\{o_7\}, \{o_5, o_6\}, \{l_7, l_8\}) = \{\mathsf{new\text{-}lnk}_{2,4}l_7, \mathsf{new\text{-}lnk}_{4,2}l_8\} = \{l_{18}, l_{19}\}$.

Finally, we lift the renaming functions to entire fragment substitutions:

$$\mathsf{new\text{-}fs}_{i,j}(p, r, b)\; O_j = \left(\mathsf{new\text{-}frg}_j p, \mathsf{new\text{-}frg}_i r, \mathsf{new\text{-}bdg}_{i,j}\left(r_{\mathsf{Obj}}, O_j, b\right)\right).$$

Intuitively, if objects are in set O_j then they should be renamed using the j-indexed renaming functions. It they are in the replacement of the fragment substitution then the i-indexed renaming functions apply. The set O_j will be provided in the semantics according to the context, and it should always be disjoint from objects of the replacement r_{Obj}.

In our example copying fs_3 for i_2 is done by copying p_1 for i_1, r_3 for i_2 and the binding link has its source copied for i_1 and its target for i_2:

$$\mathsf{new\text{-}fs}_{2,1}(p_1, r_3, \{l_6\})\, r_1 = \left(\mathsf{new\text{-}frg}_1 p_1, \mathsf{new\text{-}frg}_2 r_3, \mathsf{new\text{-}bdg}_{2,1}\left(\{o_5, o_6\}, r_1, \{l_6\}\right)\right).$$

Flattening variability models and configurations. By constraint 1 we know that there can be only one fragment substitution mapped to any feature, but it is not required that every feature has a substitution mapped to it. Each feature can be instantiated multiple times in which case the fragment substitution mapped to it (if it exists) is executed multiple times (once per instance). We compute how many times each substitution should be executed and clone it the appropriate amount of times (carefully updating references). This will produce a flat set of fragment substitutions that can be executed using the rules of Sec. 4.2.

The flattening of a variability model M with respect to a configuration Cfg is a set of fragment substitutions, denoted below as $[\![M, \mathit{Cfg}]\!]$. Flattening moves all the necessary information from the feature model and from the fragment substitutions to a new set of fragment substitutions. After this, the features and their instances can be disregarded.

Given a variability model $M = (\mathit{Fs}, \mathit{Fm}, \mathsf{mapping})$ and a configuration Cfg, $\mathsf{mapping}^{-1}(\mathsf{ty}\,i)$ returns the fragment substitution that has to be executed in the context of an instance i or $\perp$ if there is no such substitution.

There are three cases to consider when flattening the model. In the first case, instances of features that have no substitutions mapped to them are ignored by the semantics. In the second case, instances of features that have substitutions with empty placement closures such that they do not apply to any other substitution are copied with the following rule:

$$\frac{i \in \mathit{Cfg} \quad \mathsf{mapping}^{-1}(\mathsf{ty}\,i) = \mathit{fs}_i \quad \lceil p \rceil_{\mathit{fs}_i} = (\emptyset, \emptyset)}{\mathsf{new\text{-}fs}_{i,\_}\; \mathit{fs}_i\; \emptyset \in [\![M, \mathit{Cfg}]\!]} \quad (\textsc{copy-indep})$$

Since the placement fragment is empty and the binding links endpoints can only be objects of the replacement fragment itself, binding links can be appropriately cloned by using just the instance i, by $\mathsf{new\text{-}fs}$.

In the third case, instances of features that have substitutions which apply to other substitutions are copied with the following rule:

$$\frac{\begin{array}{c} i, j \in \mathit{Cfg} \quad \mathsf{mapping}^{-1}(\mathsf{ty}\,i) = \mathit{fs}_i \quad \mathsf{mapping}^{-1}(\mathsf{ty}\,j) = \mathit{fs}_j \quad \mathit{fs}_i \sqsubset \mathit{fs}_j \\ \mathit{fs}_i = (p_i, r_i, b_i) \quad \mathit{fs}_j = (\_, r_j, \_) \quad j \in \mathsf{parent}^* i \end{array}}{\mathsf{new\text{-}fs}_{i,j}\, \mathit{fs}_i\, r_j \in [\![M, \mathit{Cfg}]\!]} \quad (\textsc{copy})$$

The intended meaning of $\textsc{copy}$ is that we copy the replacement fragment using the instance i, the placement with the instance j and the binding links with a combination of the two. We use r_j, the replacement fragment of fs_j to state that a binding link endpoint can either be in the r_i or r_j. By constraint 1 we know that for any pair of instances i and j, $\mathsf{mapping}^{-1}(\mathsf{ty}\,i)$ and $\mathsf{mapping}^{-1}(\mathsf{ty}\,j)$ are uniquely determined (if they exist), thus the rule can be applied deterministically.

In our example we know that $i_2, i_1 \in \mathit{Cfg}$, $\mathsf{mapping}^{-1}(\mathsf{ty}\,i_2) = \mathit{fs}_3$ and $\mathsf{mapping}^{-1}(\mathsf{ty}\,i_1) = \mathit{fs}_1$, $\mathit{fs}_3 \sqsubset \mathit{fs}_1$ and $i_1 \in \mathsf{parent}^* i_2$, therefore we copy fs_3 in the flattened set: $\mathsf{new\text{-}fs}_{2,1}(p_1, r_3, \{l_6\})\, r_1 \in [\![M, \mathit{Cfg}]\!]$.

Lemma 3. *For a well-formed variability model M and a valid configuration Cfg, the above rules define a unique well-formed set of fragment substitutions $[\![M, \mathit{Cfg}]\!]$.*

```
1   Definition Object := nat.
2   Definition ObjectSet := list Object.
3
4   Inductive Link := link : nat -> Object -> Object -> Link.
5   Definition LinkSet := list Link.
6   Notation "id ¤ src --> tgt" := (link id src tgt) (at level 66).
7
8   Inductive Graph := graph : ObjectSet -> LinkSet -> Graph.
9   Notation "gObj ** gLnk" := (graph gObj gLnk) (at level 64).
10  Definition Model := Graph.
11  Definition Fragment := Graph.
12
13  Inductive FragSubst :=
14    fragsubst : Fragment -> Fragment -> LinkSet -> FragSubst.
15  Notation "p ,. r ., b" := (fragsubst p r b) (at level 65).
16  Definition FragSubstSet := list FragSubst.
```

Listing 2. Abstract syntax of Featherweight VML in Coq.

The well-formedness of the output follows from the isomorphism of all renaming operations (all functions are injective and preserve links)—all non-overlapping conditions of well-formedness are thus transferred from the input set of fragment substitutions.

Theorem 1. *Given a well-formed variability model M and a valid configuration Cfg the result of executing the model is unique, and given by* $[\![M, Cfg]\!]$*, and consequently the above formulation of the semantics is confluent.*

The well-formedness constraints (C 1, 2, 3, 4) ensure that the flattening input set of fragment substitutions form a closed union of fragments. Lemma 3 ensures that the output of the flattening is a unique set of substitutions forming a closed union of fragments. Lemma 3 ensures that copying process results in a closed product variant model and Lemma 2 ensures that the result is unique regardless of the ordering of the input objects and links.

5. Translation validation for Featherweight VML

As proof of concept we implemented the translation validation mechanism for the execution of a fragment substitution set.[10] This covers the syntax and semantics presented in Sec. 4.2.

As we explained in Sec. 3, translation validation has several requirements. We present the common semantic framework for the input and output as the implementation of Featherweight VML in Coq in Sec. 5.1. A formalization of the notion of *correct implementation* and the simulation of product derivation in Sec. 5.2. An *automatic proof generator* and a *proof checker* are also required to perform the validation. These are fundamental features of Coq and we explain how we use them in Sec. 5.3. Then we describe Micro CVL, a variant derivation tool developed using the Eclipse Modeling Framework (EMF)[11] that works on Ecore models in Sec. 5.4. Finally we describe how to lift the input and output of Micro CVL to Coq abstractions and validate the derivation in Sec. 5.5.

5.1. Fragment substitution syntax in Coq

Listing 2 shows the core syntax of abstract models and fragment substitutions. In Featherweight VML objects and links are discrete identifiable entities. We use the predefined set of natural numbers (nat) as identifiers. To define new types we can either use the *Definition* keyword to rename existing types or the *Inductive* keyword to specify how new types can be created by composing existing ones. Line 1 defines objects to be simple identifiers without any additional properties. Line 2 defines a type for sets of objects by reusing the predefined parametrized type list. This definition does not enforce that ObjectSets are actually sets, as lists are ordered and can contain the same element multiple times. We will later define properties to enforce this. Line 4 defines the Link type. The constructor, link, takes a nat identifier and two Objects, representing the source and the target of a Link. Line 6 sets up a concise infix notation for links. Similarly a Graph is a pair of sets of objects and links (line 8). For the Model and Fragment types we simply reuse the abstract type Graph, reflecting the syntax defined in Sec. 4.1. The FragSubst type of fragment substitutions combines two fragments and a set of links (line 13), the *placement*, *replacement* and *binding* as in Definition 2.

Featherweight VML relies heavily on set theory. When we encoded Featherweight VML in Coq we also implemented the basic properties, relations and operations of set theory. All the properties and relations are decidable and Coq can automat-

[10] https://github.com/afla/FeatherweightVML_Coq.

[11] http://www.eclipse.org/modeling/emf/.

```
1  Inductive SetContainsObject : ObjectSet −> Object −> Prop :=
2    | SetContainsObject_h : forall o t, SetContainsObject (o::t) o
3    | SetContainsObject_t : forall o h t,
4                               SetContainsObject t o −> SetContainsObject (h::t) o.
5
6  Definition SetContainsObject_dec (s : ObjectSet) (o : Object) :=
7    { SetContainsObject s o } + { ∼SetContainsObject s o }.
8
9  Definition setContainsObject: forall s o, SetContainsObject_dec s o.
10 refine (fix setContainsObject s o: SetContainsObject_dec s o :=
11   match s with
12   | [ ] => right _
13   | h:: t => if eq_nat_dec h o
14             then left _
15             else if setContainsObject t o then left _ else right _
16   end).
17 Proof.
18   unfold not. intro . inversion H. rewrite _H. apply SetContainsObject_h.
19   apply SetContainsObject_t. apply _H0. unfold not. intro . inversion H.
20   apply _H. apply H0. unfold not in _H0. apply _H0. apply H3.
21 Defined.
```

Listing 3. A set membership property along with a function that computes the property for any ObjectSet and Object.

ically compute whether they hold for any concrete sets or set elements. For example, Listing 3 shows the SetContainsObject property which tests whether a set contains a particular Object. In Coq, properties are a special kind of inductive types. The constructor SetContainsObject_h specifies the base case when the Object is accessible as the head of the list, while the constructor SetContainsObject_t specifies the inductive case when the Object is contained in the tail of the list.

In order to prove that an actual ObjectSet contains a specific Object, so that property SetContainsObject holds for them, we would have to manually build a proof object, establishing SetContainsObject, by repeatedly applying the two constructors. Instead, we demonstrate that the property is decidable and provide Coq with the means to automatically check the property for any input. Lines 6 and 7 define SetContainsObject_dec, a decidable type that holds both the fact that an ObjectSet contains a specific Object or not and the proof object. The notation on line 7 indicates a **left** constructor for the positive evaluation of the property and a **right** constructor for the negative evaluation.

At this point we can use Coq as an automatic proof generator. Lines 9 to 16 define setContainsObject, a theorem refined as a fixpoint that calculates the proof object. The fixpoint iterates through the ObjectSet and if it manages to build a complete proof object it returns the **left** constructor of the decidable type SetContainsObject_dec. Otherwise it calls the **right** constructor. Lines 17 to 21 represent the proof that the fixpoint is correct. Executing setContainsObject on an actual ObjectSet and an actual Object allows Coq to generate the proof for that particular case. This approach is much easier and scales very well compared to generating proofs for large models from outside Coq. It is also more reliable and can be performed automatically on any input. In a similar way we have proven that all our other properties are decidable and computable by Coq.

The equality property differs for most types. In Featherweight VML two objects are equal if their identifiers are equal. The same is true for links, so in Coq we use the property LinkEqualId to test that 3¤6–>8 and 3¤1–>4 are equal and cannot be contained in the same LinkSet. However, the same link identifier can occur in multiple link sets, or model fragments, or fragment substitutions. We must check that the source and target objects are the same in all occurrences of the link. We call this property *link consistency*. We do this by defining a series of link consistency properties which can be checked for any two structures that contain links.

Sets of any type are equal if they contain the same elements. For graphs (i.e. models and fragments) equality is verified pair-wise on the object and link sets. Fragment substitutions are equal if the placement/replacement fragments and bindings are equal respectively.

5.2. Fragment substitution semantics in Coq

Before we could encode the two copying rules, OBJ-COPY and LNK-COPY (Sec. 4.2), we had to implement some helper functions that, given a set of fragment substitutions, extract the following sets: all placement objects, all placement links, all replacement objects/links, all binding links and all objects/links throughout the entire fragment substitution set. For example, Listing 4 shows a function that computes the set of all placement objects from all placement fragments of the fragment substitution set fss. We then verify that it executes correctly by proving the theorem PlacementObjectsExecutes which states that for any fragment substitution (pObj**pLnk,.r.,b) contained by fss, it is implied that pObj will be a subset of the complete set of placement objects. Theorem PlacementObjectsOfEqualFss states that any two fragment substitution sets that are equal (i.e. contain the same elements but possibly in different orders) contain the same placement objects. The

```
Fixpoint placementObjects (fss : FragSubstSet) : ObjectSet :=
match fss with
  | [ ] => [ ]
  | (pObj**pLnk,.r.,bdg)::t => objectSetUnion pObj (placementObjects t)
end.

Theorem PlacementObjectsExecutes : forall fss pObj pLnk r b,
  SetContainsFragSubst fss (pObj**pLnk,.r.,b)
  −> ObjectSubset pObj (placementObjects fss).

Theorem PlacementObjectsOfEqualFss : forall fss1 fss2,
  FragSubstSetEqual fss1 fss2 −>
  ObjectSetEqual(placementObjects fss1)(placementObjects fss2).
```

Listing 4. A function that computes all the placement objects from a set of fragment substitutions along with two theorems stating the correctness of this computation.

```
Fixpoint objCopy (obj : ObjectSet) (fss : FragSubstSet) : ObjectSet :=
match obj with
  | [ ] => [ ]
  | h::t =>
    if bSetContainsObject (replacementObjects fss) h
      && !(bSetContainsObject (placementObjects fss) h)
    then h::(objCopy t fss)
    else (objCopy t fss)
end.
```

Listing 5. Implementing the OBJ-COPY inference rule as a function.

```
Function executeFss (fss : FragSubstSet) : Model :=
    (objCopy (allObjects fss) fss)**(lnkCopy (allLinks fss) fss).
```

Listing 6. Implementation of the execution function as the combined application of objCopy and lnkCopy.

theorems have a double role: first, they are improving the quality of the translation validator by having extra checks of the execution and second, they are used in proving larger theorems. We provide similar theorems for all executable functions.

$$\frac{o \in (\bigcup_{(\_,r,\_)\in Fs} r_{\mathsf{Obj}}) \qquad o \notin (\bigcup_{(p,\_,\_)\in Fs} p_{\mathsf{Obj}})}{o \in [\![Fs]\!]_{\mathsf{Obj}}} \ (\text{OBJ-COPY})$$

Finally we have implemented the two copy rules. Listing 5 shows the implementation of OBJ-COPY. Given a set of objects obj and a set of fragment substitutions fss, objCopy will check if the head of obj is a replacement object in fss and in the same time is not a placement object in fss. If the conditions are met, then the head object is copied and the function is called recursively on the tail, otherwise the object is not copied and the tail is processed. The implementation of lnkCopy is analogous to objCopy.

The execution of a fragment substitution set, presented in Listing 6, is simply composing a variant model by applying objCopy to all objects in fss and lnkCopy to all links in fss.

5.3. *Proof of correctness and confluence*

Independently, we also wanted to ensure that the Coq implementation of Featherweight VML is of good quality. Listing 7 shows theorem objCopyExecutes which states that the function objCopy correctly implements the copying rule OBJ-COPY. An interesting aspect of this theorem is that it checks a bidirectional implication in the sense that the resulting set contains *all* and *only* the required elements. There is also an analogous theorem for links, lnkCopyExecutes.

To prove the confluence of the execution function we separately prove the confluence of objCopy and lnkCopy. Listing 8 shows the confluence theorems for objCopy. Theorem objCopyOnEqualFss states that the copying function is confluent with respect to the set of fragment substitutions. Theorem objCopyOfEqualSets states that the function is confluent with respect to the sets of object/links from which they copy the elements of the variant. Together, the two theorems along with two similar others for lnkCopy verify that executing two fragment substitution sets produces isomorphic variant models, thus execution is confluent.

```
Theorem objCopyExecutes : forall o fObj fss,
  SetContainsObject fObj o
  ∧ SetContainsObject (replacementObjects fss) o
  ∧ ∼SetContainsObject (placementObjects fss) o
  <-> SetContainsObject (objCopy fObj fss) o.
```

Listing 7. Theorem stating the correctness of objCopy using bidirectional implication.

```
Theorem objCopyOnEqualFss : forall obj fss1 fss2,
  FragSubstSetEqual fss1 fss2
  -> ObjectSetEqual (objCopy obj fss1) (objCopy obj fss2).

Theorem objCopyOfEqualSets : forall obj1 obj2 fss,
  ObjectSetEqual obj1 obj2
  -> ObjectSetEqual (objCopy obj1 fss) (objCopy obj2 fss).
```

Listing 8. Confluence theorems for objCopy.

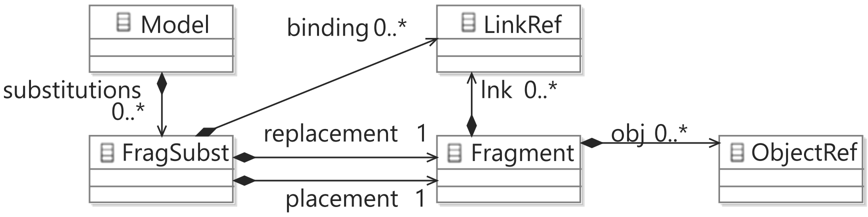

Fig. 11. The metamodel of Micro CVL specified in the EMF Ecore language.

5.4. Micro CVL—a variant derivation tool

To demonstrate the verification of variant models through translation validation we implemented Micro CVL—a *small* variant derivation tool designed as a subset of CVL. In this demonstration, Micro CVL will stand for any actual variant derivation tool.

EMF facilitates the creation of Domain Specific Languages (DSL) by providing a set of tools and a meta-language, Ecore. We implemented Micro CVL to handle variability over any Ecore-based model. The metamodel of Micro CVL is presented in Fig. 11. The language is a set of fragment substitutions (FragSubst). Each fragment substitution contains one placement and one replacement fragment and a set of binding link references. The ObjectRef and LinkRef are references pointing to a base model that is an instance of an arbitrary Ecore-based domain-specific language. We can also write Micro CVL models in textual form (see Listing 9).

A few notes on implementing fragment substitutions of Ecore models:

- EMF does not provide a way to uniquely identify objects. This is a problem because we need to know if the objects of the variant model are the same objects from the subject model. We require that all objects of the subject model inherit from an abstract class with an integer *id* field that is unique throughout the model. All classes in the Device metamodel inherit from the Component abstract class.
- Similarly, links cannot be uniquely identified in Ecore models. In order to reference links we identify them with a combination of the name of the metamodel relation that the link implements and the *ids* of the source and target objects. We also require that there are no multiple links between the same source and target, implementing the same relation.

5.5. Lifting and validating Ecore models

Lifting Ecore models to Coq abstract models is considerably simpler than the actual variant derivation process. After parsing the XML files and obtaining the abstract syntax trees we traverse the trees in pre-order and encode them in Coq models. It is important to notice that the Ecore models and their Coq abstractions are isomorphic, thus it is easily verifiable that the lifting is correct.

In Fig. 12 we recall the Device model example from Fig. 2. We assume that the Device model has an Ecore metamodel which we do not show. We also recall the fragment substitution fs_2 of the CVL model:

$$fs_2\{\ placement\{\mathsf{pa}, \mathsf{pb}\}\ replacement\{\mathsf{rb}\}\ binding\{(\mathsf{pb}, \mathsf{rb})\}\ \}$$

We represent the fragment substitution fs_2 in Micro CVL as shown in Lst. 9.

```
FragSubst
  placement Fragment
    obj [ ObjectRef(dualOutput) ]
    lnk [ LinkRef(sensor.out -> dualOutput) ]
  replacement Fragment
    obj [ ObjectRef(data), ObjectRef(actuatorCpu) ]
    lnk [ LinkRef(actuatorCpu.rawinput -> data) ]
  binding [ LinkRef(deviceCpu.out -> data) ]
```

Listing 9. A Micro CVL representation of fs_2.

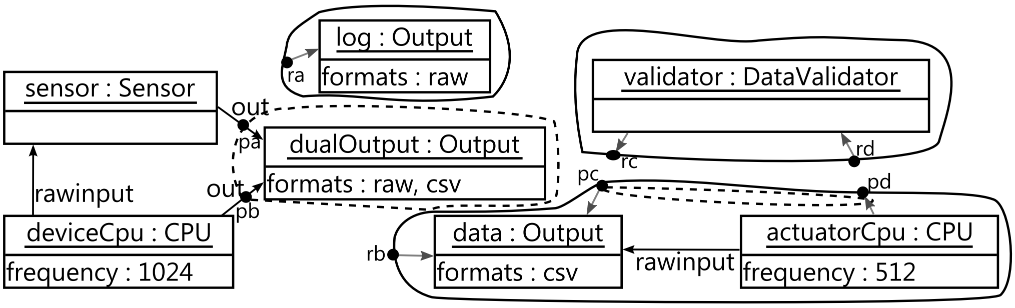

Fig. 12. CVL model.

The lifting process is implemented as follows:

1. We traverse and lift the Micro CVL model to Coq:
   - whenever we encounter an ObjectRef, we search the base model for the integer *id* of the referred object;
   - whenever we encounter a LinkRef composed of the relation name and references to the source and target objects, we assign to it a new unique integer *id* and store it in a HashMap; we use this newly assigned *id* and the source and target *ids* from the subject model to lift the link.
2. We traverse and lift the output product variant Ecore model to Coq:
   - whenever we encounter an object we use the integer *id* as the abstraction;
   - whenever we encounter a link, composed of the relation name and source and target objects, we look up its assigned *id* in the HashMap created in the previous step and the source and target object *ids* as the abstraction.

Considering that the base model in Fig. 1 can be abstracted to the model in Fig. 7a, that the Micro CVL model can be abstracted to the fragment substitutions in Fig. 7b and that the derived variant model can be abstracted to the model in Fig. 7c, the lifting of the Ecore models will produce the Coq representation presented in Listing 10. Lines 1 to 3 show the complete original base model. Lines 5 to 11 show the fragment substitutions, where line 7 represents the default artifacts being copied and the other three represent the changes. Line 9 in particular represents the FragSubst from Listing 9. Lines 13 to 15 show the variant model obtained by lifting the output of the back box derivation tool.

Finally, line 17 verifies that the set of fragment substitutions is consistent with the original model. Line 19 computes the simulation result by executing the Coq fragment substitution set. Line 20 evaluates whether the simulation result and the variant model are equivalent and displays the result as a Boolean value. An an extra check, line 22 checks that the variant model is consistent with the base model meaning that all links have maintained their original source and target.

6. Discussion and related work

Featherweight VML is closely related to CVL as it is able to express CVL models with great accuracy. Most CVL variability specifications can be reduced to features with cardinalities and the variation points are all specific cases of the fragment substitution. Featherweight VML can be seen as a generalization of OVM. We can use abstract features to group variation points together, giving OVM a tree structure while retaining the same meaning. Delta modules are almost identical to fragment substitutions. The only difference is that a delta module is guarded by an application condition over a set of features while Featherweight VML fragment substitutions are each mapped to a single feature. In order to express a Delta model without adding extra concepts we would have to change Featherweight VML's mapping function to a more general expression.

Aside from OVM, CVL and Delta Modeling, there are numerous other approaches to formalizing the elements of a variability model. Many of the variability abstractions used for software product lines, such as feature models [3] or decision models [4], are a subset of configuration modeling and knowledge-based configuration ontologies and approaches [26–29]. We chose to use feature models in defining Featherweight VML simply because they are the preferred option in the industry. The mapping from features to artifacts has been specified by using feature modules [30] to wrap the artifacts pertinent to

```
Definition base : Model :=
  [1; 2; 3; 4; 5; 6; 7]**[1¤3——>1; 2¤1——>2; 3¤3——>2; 4¤1——>4;
    5¤6——>5; 6¤3——>5; 7¤7——>5; 8¤6——>7].

Definition fss : FragSubstSet :=
  [
     (([]**[])   ,.  ([1; 2; 3]**[1¤3——>1; 2¤1——>2; 3¤3——>2]) ., []);
    (([2]**[2¤1——>2]) ,. ([4]**[])  .,  [4¤1——>4]);
    (([2]**[2¤1——>2]) ,. ([5; 6]**[5¤6——>5]) ., [6¤3——>5]);
     (([]**[])   ,.  ([7]**[])   .,  [7¤7——>5; 8¤6——>7])
  ].

Definition variant : Model :=
  [1; 3; 4; 5; 6; 7]**[1¤3——>1; 4¤1——>4; 5¤6——>5; 6¤3——>5; 7¤7——>5;
    8¤6——>7].

Eval compute in bFragSubstSetIsConsistentWithModel fss base.

Eval compute in executeFss fss.
Eval compute in bModelEqual (EXE.executeFss fss) variant.

Eval compute in bModelConsistentWithModel variant base.
```

Listing 10. Coq representation of models.

each feature and applying module composition to derive new variants; alternatively [31,32], by annotating the artifacts with presence conditions thus specifying when each artifact must be present in a product variant. Other approaches [33] involve model transformations where each feature can both remove and add artifacts to existing models. While some formalisms are richer than feature models (e.g. Koala [34] employs a topology of components that is not a tree and interfaces between components), a lot of these formalisms provide a fixed representation of base models (e.g. component models) and do not relate to base models specified in customized domain-specific languages or to concrete implementation artifacts such as source code [32,35]. All these approaches bring different advantages and challenges to the domain of variability modeling, usually compromising between expressive power and simplicity and which influences the possibility of validating actual derivation tools.

So far, most work on variability was dedicated to analyzing feature models [36,37]. Recent work has provided valuable insight such as formalizing feature models represented in a textual language [38] or even providing full proofs in the PVS proof assistant [39]. However, the formalization is limited to feature models and do not touch on the subject of variant derivation. Czarnecki et al. [40] show how to represent the three layers of variability modeling within the single Clafer syntax. However no actual mapping to implementation artifacts is considered, just a Boolean abstraction of dependency. Such a formalization cannot directly be used as a specification of correctness for a variant derivation tool. Other works consider analyzing variability models as a whole, including checking for consistency (for instance [41–45]). All these methods assume correctness of the variant derivation implementation. In this work we make the first step to allow fulfilling this assumption by setting the foundation of analyzing the implementation of variability realization tools.

A crucial feature of our semantics is that it is confluent. We achieve this by identifying sufficient conditions for confluence, and adopting copying style for definition of semantics, to minimize dependencies between executions of individual variation points. Oldevik et al. [46] take a dual route and attempt to detect lack of confluence. As such they belong well to the group of works that are more interested in ensuring that models are correct than that the model manipulation tools are correct.

Since its introduction [11] translation validation has been successfully applied to compilers [47,48], finite state machine transformations [49] or system abstractions [50]. Also, a translation validation for the LLVM compiler by abstracting the input and output to value-graphs [51] and a Coq verified translation validation by symbolic execution [52] have been proposed.

To the best of our knowledge, this is the first application of translation validation in the domain of software product lines. It is also the first implementation that is completely independent from the modeling language in which the input and output are specified. The only requirement is that the variability modeling language can be lifted to Featherweight VML. However, the lifting is considerably simpler and easier to verify than the actual variant derivation process and no extra work is required for new models and metamodels.

Coq supports automatic generation of formally verified implementations of systems (in Haskell and OCaml) out of type and function definitions. Since the lifting is an abstraction, some information from the input and output may be lost (such as attribute values of objects) which means that no concrete execution can be created automatically from an abstract execution. We believe that this use of abstraction is crucial to the success of the method. It allows implementing and growing validators incrementally, without falling into a trap of diminishing returns.

Parts of this work have been presented before. The syntax and semantics of fragment substitutions have been introduced as an extended abstract presented at the Nordic Workshop on Programming Theory, NWPT 2013, in Tallinn. The specification of Featherweight VML (sections 4.1–4.3) was described in [53]. This paper is a long version of the above mentioned works. It adds the entire translation validation framework around the formal semantics of Featherweight VML, including formalizations in Coq, the mechanized proofs in Coq, and implementation of translation validation on top of a home grown variability modeling tool for the Eclipse Modeling Framework.

While Micro CVL, the variability language we developed, only has demonstrative value, we also provided a proof of concept within the VARIES[12] research project. For this, we have looked at the Base Variability Resolution (BVR) language [54]. BVR is a derivative of CVL and, for the most part, it follows the same architecture and employs the same concepts. There is also a prototype implementation of a BVR tool as an Eclipse plug-in.[13] The implementation project has *over one thousand Java classes* where the product derivation code is mixed with Eclipse plug-in code and tests. By comparison, we implemented the lifting operation with just *six Java classes* with a time cost of roughly *150 man–hours* of research and development. Even though this is a rough estimate based on our own experience, it indicates that the cost of implementing translation validation for an actual tool is very small in comparison with the cost of producing the tool itself.

Once the lifting operation is implemented for a variability language, the validation technique can be applied to all projects in which the variability language is used. The nature of the product line or even of the architectural language does not influence the validation technique, meaning that the abstraction to Featherweight VML is a one-time cost. Maintenance of the abstraction is required only in case the variability language itself changes which, in our experience, does not happen very often. In case the changes to the variability language do not influence the derivation algorithm (e.g. it expands to work with a new architectural language) there are, again, no costs to the abstraction. If the changes affect the derivation process (e.g. a new feature is added) then the maintenance implies abstracting the new constructs, which can be defined as syntactic sugar in terms of fragment substitutions. In such cases it is difficult to give clear estimates of the cost, but we believe it is safe to assume that the changes to the abstraction are directly proportional to the changes of the variability language.

The translation validation technique is especially useful when creating trustworthy tools for developing safety critical systems. However, it can be applied with no extra costs to any kind of project, as long as the same variability modeling language is used. One issue that requires further investigation is the performance of the validation. Since non-safety-critical projects tend to have larger and less optimized code-bases, we expect a time increase for the verified formal execution and for the equivalence check between the simulation result and the actual output model.

7. Conclusion

We propose a translation validation of product derivation in software product lines. This technique can be applied to any variant derivation tool, but it is especially useful when creating trustworthy tools for developing safety critical systems. We formally define Featherweight VML, a compact variability modeling language, which retains the expressiveness of CVL on which it is based, but at the same time it has much simpler syntax and semantics. To our best knowledge this is the first attempt to fully formalize an entire variability model. Featherweight VML can be used as an abstraction of CVL, OVM, Delta Modeling and any other variability modeling language that satisfies the same core requirements. Our semantics processes the model in an order-agnostic manner. It is the first confluent formalization of a CVL-like language. We implemented Featherweight VML in Coq and provided proofs of the correctness and confluence of the semantics. We also demonstrated the translation validation of a black box tool, including the lifting of the input and output to Coq representations.

Acknowledgements

This work was supported by ARTEMIS JU under grant agreement No. 295397 and by Danish Agency for Science, Technology and Innovation. We thank Ina Schaefer for help with defining the semantics of Featherweight VML.

References

[1] T. Stahl, M. Völter, J. Bettin, A. Haase, S. Helsen, Model-Driven Software Development: Technology, Engineering, Management, John Wiley & Sons, 2006.
[2] P. Clements, L.M. Northrop, Software Product Lines: Practices and Patterns, Addison–Wesley, 2002.
[3] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, Feature-Oriented Domain Analysis (FODA) feasibility study, Tech. rep., CMU SEI, 1990.
[4] K. Schmid, R. Rabiser, P. Grünbacher, A comparison of decision modeling approaches in product lines, in: Proc. Fifth International Workshop on Variability Modelling of Software-Intensive Systems, Namur, Belgium, January 27–29, 2011, in: ACM International Conference Proc. Series, ACM, 2011, pp. 119–126.
[5] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering—Foundations, Principles, and Techniques, Springer, 2005.
[6] I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella, Delta-oriented programming of software product lines, in: J. Bosch, J. Lee (Eds.), Software Product Lines: Going Beyond – Proc. 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13–17, 2010, in: Lecture Notes in Computer Science, vol. 6287, Springer, 2010, pp. 77–91.

[12] http://www.varies.eu/.
[13] GitHub repository (requires authentication): https://github.com/SINTEF-9012/bvr.

[7] CVL Joint Submission Team, Common Variability Language (CVL), OMG revised submission, OMG document: ad/2012-08-05, 2012.
[8] T. Berger, R. Rublack, D. Nair, J.M. Atlee, M. Becker, K. Czarnecki, A. Wasowski, A survey of variability modeling in industrial practice, in: The Seventh International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '13, Pisa, Italy, January 23–25, 2013, ACM, 2013, p. 7.
[9] J. Hutchinson, M. Rouncefield, J. Whittle, Model-driven engineering practices in industry, in: Proc. 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21–28, 2011, ACM, 2011, pp. 633–642.
[10] H. Gall, Functional safety IEC 61508/IEC 61511 the impact to certification and the user, in: The 6th ACS/IEEE International Conference on Computer Systems and Applications, AICCSA 2008, Doha, Qatar, March 31–April 4, 2008, IEEE Computer Society, 2008, pp. 1027–1031.
[11] A. Pnueli, M. Siegel, E. Singerman, Translation validation, in: B. Steffen (Ed.), Tools and Algorithms for Construction and Analysis of Systems, Proc. 4th International Conference, TACAS '98, Lisbon, Portugal, March 28–April 4, 1998, in: Lecture Notes in Computer Science, vol. 1384, Springer, 1998, pp. 151–166.
[12] K. Czarnecki, S. Helsen, U.W. Eisenecker, Formalizing cardinality-based feature models and their specialization, Softw. Process Improv. Pract. 10 (1) (2005) 7–29, http://dx.doi.org/10.1002/spip.213.
[13] D. Clarke, N. Diakov, R. Hähnle, E.B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, P.Y.H. Wong, Modeling spatial and temporal variability with the HATS abstract behavioral modeling language, in: M. Bernardo, V. Issarny (Eds.), Formal Methods for Eternal Networked Software Systems—11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, Advanced Lectures, SFM 2011, Bertinoro, Italy, June 13–18, 2011, in: Lecture Notes in Computer Science, vol. 6659, Springer, 2011, pp. 417–457.
[14] A. Haber, C. Kolassa, P. Manhart, P.M.S. Nazari, B. Rumpe, I. Schaefer, First-class variability modeling in Matlab/Simulink, in: The Seventh International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '13, Pisa, Italy, January 23–25, 2013, ACM, 2013, p. 4.
[15] A. Haber, K. Hölldobler, C. Kolassa, M. Look, B. Rumpe, K. Müller, I. Schaefer, Engineering delta modeling languages, in: 17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan, August 26–30, 2013, ACM, 2013, pp. 22–31.
[16] Object Management Group, Meta Object Facility (MOF) Core specification version 2.0, OMG document: formal/06-01-01, 2006.
[17] T. Thüm, C. Kästner, S. Erdweg, N. Siegmund, Abstract features in feature modeling, in: Software Product Lines—15th International Conference, SPLC 2011, Munich, Germany, August 22–26, 2011, IEEE, 2011, pp. 191–200.
[18] D. Clarke, M. Helvensteijn, I. Schaefer, Abstract delta modeling, in: Generative Programming and Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10–13, 2010, ACM, 2010, pp. 13–22.
[19] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, A. Wasowski, Cool features and tough decisions: a comparison of variability modeling approaches, in: Proc. Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25–27, 2012, ACM, 2012, pp. 173–182.
[20] A. Narayanan, G. Karsai, Specifying the correctness properties of model transformations, in: Proceedings of the Third International Workshop on Graph and Model Transformations, GRaMoT '08, ACM, 2008, pp. 45–52.
[21] T. Mens, On the use of graph transformations for model refactoring, in: R. Lämmel, J. Saraiva, J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering, International Summer School, Revised Papers, GTTSE 2005, Braga, Portugal, July 4–8, 2005, in: Lecture Notes in Computer Science, vol. 4143, Springer, 2005, pp. 219–257.
[22] T. Mens, G. Taentzer, O. Runge, Analysing refactoring dependencies using graph transformation, Softw. Syst. Model. 6 (3) (2007) 269–285, http://dx.doi.org/10.1007/s10270-006-0044-6.
[23] G. Taentzer, AGG: a tool environment for algebraic graph transformation, in: M. Nagl, A. Schürr, M. Münch (Eds.), Applications of Graph Transformations with Industrial Relevance, Proceedings of International Workshop, AGTIVE'99, Kerkrade, The Netherlands, September 1–3, 1999, in: Lecture Notes in Computer Science, vol. 1779, Springer, 1999, pp. 481–488.
[24] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: a model transformation tool, Sci. Comput. Program. 72 (1–2) (2008) 31–39, http://dx.doi.org/10.1016/j.scico.2007.08.002.
[25] A. Schürr, Specification of graph translators with triple graph grammars, in: E.W. Mayr, G. Schmidt, G. Tinhofer (Eds.), Graph-Theoretic Concepts in Computer Science, Proc. 20th International Workshop, WG '94, Herrsching, Germany, June 16–18, 1994, in: Lecture Notes in Computer Science, vol. 903, Springer, 1994, pp. 151–163.
[26] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, M. Zanker, Transforming UML domain descriptions into configuration knowledge bases, in: Knowledge Transformation for the Semantic Web, IOS Press, 2003, pp. 154–168.
[27] A. Hubaux, D. Jannach, C. Drescher, L. Murta, T. Männistö, K. Czarnecki, P. Heymans, T. Nguyen, M. Zanker, Unifying software, product configuration: a research roadmap, in: W. Mayer, P. Albert (Eds.), Proceedings of the Workshop on Configuration at ECAI 2012, Montpellier, France, August 27, 2012, in: CEUR Workshop Proceedings, vol. 958, CEUR-WS.org, 2012, pp. 31–35, http://ceur-ws.org/Vol-958/paper6.pdf.
[28] D. Benavides, A. Felfernig, J.A. Galindo, F. Reinfrank, Automated analysis in feature modelling and product configuration, in: J.M. Favaro, M. Morisio (Eds.), Safe and Secure Software Reuse—Proceedings 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18–20, in: Lecture Notes in Computer Science, vol. 7925, Springer, 2013, pp. 160–175.
[29] K. Czarnecki, A. Hubaux, E. Jackson, D. Jannach, T. Männistö, Unifying product and software configuration, in: Dagstuhl Seminar 14172, Dagstuhl Rep. 4 (4) (2014) 20–35, http://dx.doi.org/10.4230/DagRep.4.4.20.
[30] C. Prehofer, Feature-oriented programming: a new way of object composition, Concurr. Comput., Pract. Exp 13 (6) (2001) 465–501, http://dx.doi.org/10.1002/cpe.583.
[31] K. Czarnecki, M. Antkiewicz, Mapping features to models: a template approach based on superimposed variants, in: R. Glück, M.R. Lowry (Eds.), Generative Programming and Component Engineering, Proc. 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29–October 1, 2005, in: Lecture Notes in Computer Science, vol. 3676, Springer, 2005, pp. 422–437.
[32] M. Janota, G. Botterweck, Formal approach to integrating feature and architecture models, in: J.L. Fiadeiro, P. Inverardi (Eds.), Fundamental Approaches to Software Engineering, Proc. 11th International Conference, FASE 2008, Budapest, Hungary, March 29–April 6, 2008, in: Lecture Notes in Computer Science, vol. 4961, Springer, 2008, pp. 31–45.
[33] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Syst. J. 45 (3) (2006) 621–646, http://dx.doi.org/10.1147/sj.453.0621.
[34] T. Asikainen, T. Soininen, T. Männistö, A Koala-based ontology for configurable software product families, in: IJCAI 2003 Configuration Workshop, 2003, pp. 45–52, http://www.soberit.hut.fi/pdmg/papers/ASIK03KOA.pdf.
[35] M. Acher, P. Collet, P. Lahire, S. Moisan, J. Rigault, Modeling variability from requirements to runtime, in: 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, Las Vegas, Nevada, USA, 27–29 April 2011, IEEE Computer Society, 2011, pp. 77–86.
[36] D. Benavides, A.R. Cortés, P. Trinidad, S. Segura, A survey on the automated analyses of feature models, in: J.C.R. Santos, P. Botella (Eds.), XI Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2006, Octubre 3–6, 2006, Sitges, Spain, 2006, pp. 367–376.
[37] P. Schobbens, P. Heymans, J. Trigaux, Y. Bontemps, Generic semantics of feature diagrams, Comput. Netw. 51 (2) (2007) 456–479, http://dx.doi.org/10.1016/j.comnet.2006.08.008.
[38] A. Classen, Q. Boucher, P. Heymans, A text-based approach to feature modelling: syntax and semantics of TVL, Sci. Comput. Program. 76 (12) (2011) 1130–1143, http://dx.doi.org/10.1016/j.scico.2010.10.005.

[39] M. Janota, J. Kiniry, Reasoning about feature models in higher-order logic, in: Software Product Lines, Proc. 11th International Conference, SPLC 2007, Kyoto, Japan, September 10–14, 2007, IEEE Computer Society, 2007, pp. 13–22.

[40] K. Bak, K. Czarnecki, A. Wasowski, Feature and meta-models in Clafer: mixed, specialized, and coupled, in: B.A. Malloy, S. Staab, M. van den Brand (Eds.), Software Language Engineering—Third International Conference, Revised Selected Papers, SLE 2010, Eindhoven, The Netherlands, October 12–13, 2010, in: Lecture Notes in Computer Science, vol. 6563, Springer, 2010, pp. 102–122.

[41] T. Berger, S. She, R. Lotufo, K. Czarnecki, A. Wasowski, Feature-to-Code mapping in two large product lines, in: J. Bosch, J. Lee (Eds.), Software Product Lines: Going Beyond—Proc. 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13–17, 2010, in: Lecture Notes in Computer Science, vol. 6287, Springer, 2010, pp. 498–499.

[42] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, M. Mezini, Spl$^{\text{lift}}$: statically analyzing software product lines in minutes instead of years, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16–19, 2013, ACM, 2013, pp. 355–364.

[43] K. Czarnecki, K. Pietroszek, Verifying feature-based model templates against well-formedness OCL constraints, in: Generative Programming and Component Engineering, Proc. 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22–26, 2006, ACM, 2006, pp. 211–220.

[44] Ø. Haugen, CVL: common variability language or chaos, vanity and limitations?, in: The Seventh International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '13, Pisa, Italy, January 23–25, 2013, ACM, 2013, p. 1.

[45] F. Heidenreich, J. Kopcsek, C. Wende, FeatureMapper: mapping features to models, in: 30th International Conference on Software Engineering, Companion Volume, ICSE 2008, Leipzig, Germany, May 10–18, 2008, ACM, 2008, pp. 943–944.

[46] J. Oldevik, Ø. Haugen, B. Møller-Pedersen, Confluence in domain-independent product line transformations, in: M. Chechik, M. Wirsing (Eds.), Fundamental Approaches to Software Engineering, Proc. 12th International Conference, FASE 2009, York, UK, March 22–29, 2009, in: Lecture Notes in Computer Science, vol. 5503, Springer, 2009, pp. 34–48.

[47] G.C. Necula, Translation validation for an optimizing compiler, in: Proc. of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Vancouver, BC, Canada, June 18–21, 2000, ACM, 2000, pp. 83–94.

[48] T.A.L. Sewell, M.O. Myreen, G. Klein, Translation validation for a verified OS kernel, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16–19, 2013, ACM, 2013, pp. 471–482.

[49] T. Li, Y. Guo, W. Liu, M. Tang, Translation validation of scheduling in high level synthesis, in: Great Lakes Symposium on VLSI 2013 (Part of ECRC), GLSVLSI'13, Paris, France, May 2–4, 2013, ACM, 2013, pp. 101–106.

[50] J.O. Blech, I. Schaefer, A. Poetzsch-Heffter, Translation validation of system abstractions, in: O. Sokolsky, S. Tasiran (Eds.), Runtime Verification, 7th International Workshop, Revised Selected Papers, RV 2007, Vancouver, Canada, March 13, 2007, in: Lecture Notes in Computer Science, vol. 4839, Springer, 2007, pp. 139–150.

[51] J. Tristan, P. Govereau, G. Morrisett, Evaluating value-graph translation validation for LLVM, in: Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4–8, 2011, ACM, 2011, pp. 295–305.

[52] J. Tristan, X. Leroy, Formal verification of translation validators: a case study on instruction scheduling optimizations, in: Proc. of the 35th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7–12, 2008, ACM, 2008, pp. 17–27.

[53] A.F. Iosif-Lazar, I. Schaefer, A. Wasowski, A core language for separate variability modeling, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods I: Verification and Validation. Technologies for Mastering Change—Proc. 6th International Symposium, Part I, ISoLA 2014, Imperial, Corfu, Greece, October 8–11, 2014, in: Lecture Notes in Computer Science, vol. 8802, Springer, 2014, pp. 257–272.

[54] Ø. Haugen, O. Øgård, BVR—better variability results, in: D. Amyot, P.F. i Casas, G. Mussbacher (Eds.), System Analysis and Modeling: Models and Reusability—Proc. 8th International Conference, SAM 2014, Valencia, Spain, September 29–30, 2014, in: Lecture Notes in Computer Science, vol. 8769, Springer, 2014, pp. 1–15.

Paper B

Experiences from Designing and Validating a Software Modernization Transformation

Alexandru F. Iosif-Lazăr
IT University of Copenhagen
afla@itu.dk

Ahmad Salim Al-Sibahi
IT University of Copenhagen
asal@itu.dk

Aleksandar S. Dimovski
IT University of Copenhagen
adim@itu.dk

Juha Erik Savolainen
Danfoss Power Electronics
juhaerik.savolainen@danfoss.com

Krzysztof Sierszecki
Danfoss Power Electronics
ksi@danfoss.com

Andrzej Wąsowski
IT University of Copenhagen
wasowski@itu.dk

***Abstract*—Software modernization often involves complex code transformations that convert legacy code to new architectures or platforms, while preserving the semantics of the original programs.**

We present the lessons learnt from an industrial software modernization project of considerable size. This includes collecting requirements for a code-to-model transformation, designing and implementing the transformation algorithm, and then validating correctness of this transformation for the code-base at hand. Our transformation is implemented in the TXL rewriting language and assumes specifically structured C++ code as input, which it translates to a declarative configuration model.

The correctness criterion for the transformation is that the produced model admits the same configurations as the input code. The transformation converts C++ functions specifying around a thousand configuration parameters. We verify the correctness for each run individually, using translation validation and symbolic execution. The technique is formally specified and is applicable automatically for most of the code-base.

***Keywords*—Experience Report, Functional Equivalence, Program Transformation, Symbolic Execution.**

I. Introduction

The panelists of the ASE 2013 conference [1] listed the growing size and complexity of legacy code as one of the key challenges of the software industry. Software modernization is an important contender among the measures to address this challenge. However, relatively little literature is available about software modernization projects, especially in the safety critical domain. We present experiences from a research partnership around an industrial software modernization project, between IT University of Copenhagen and Danfoss Power Electronics[1], a global producer of components and solutions for controlling electric motors driving various machinery. This particular modernization project involves a *configuration* tool used to adapt Danfoss frequency converters to a particular application. The configuration tool consists of thousands of C++ functions for accessing and validating the configuration parameters.

The configurator code base is undergoing a modernization process where each parameter function must be converted from imperative C++ code to a declarative specification. All the functions compute a result (either the value of the parameter or a Boolean value checking its accessibility and validity). The declarative form of a function is an expression that must compute the same result as the original imperative code. The code modernization is done automatically by applying a transformation—implemented in the rule-based transformation language TXL [2]—on each function individually. It performs a sequence of syntactic replacements, gradually eliminating C++ preprocessor directives, local variables, C++ control statements and leaving behind a pure (i.e. side-effect free) expression.

The execution of a TXL transformation can only catch trivial errors revealed by a syntactic analysis of the code (enforcing constraints through pattern matching). This does not guarantee that the semantics of the source program is preserved. In the Danfoss case, determining the semantic equivalence is a decidable problem due to certain properties of the programs—the execution always terminates, the code fragments do not contain inline assembler code and they generally employ a small subset of C++.

We assessed that symbolic execution [3] is mature enough to handle this task efficiently. We implemented a *lightweight* wrapper which compiles both the input and the output of the transformation and employs the symbolic executor KLEE [4] to assert program equivalence. KLEE produces a set of *path conditions* that represent distinct execution paths of the two programs along with their return values as symbolic formulas. If it fails to establish the equivalence of the return values of any execution path then it outputs the path condition and a counter-example.

Our contributions are:

- A synthesis of experiences from the design of a non-trivial modernization transformation for a real project.
- Designing and implementing a transformation validation technique for this case study, which produces counter-examples if two programs are not equivalent.
- Lessons learnt from using the validation technique in the case study, including an analysis of the kind of errors that have been identified.

To the best of our knowledge this is the first analysis of transformation errors extracted from an industrial project of this

[1] http://www.danfoss.com

```
Configuration config = selectedConfigParameter;
Option opt = selectedOptionParameter;
bool result = false;
switch (config) {
  case config1:
    if (opt == option1) result = true;
    break;
  default:
    result = true;
    break;
}
return result;
```

Fig. 1: An example input program.

```
(selectedConfigParameter == config1
   && selectedOptionParameter == option1)
? true
: false
```

Fig. 2: The output program after the transformation.

scale and complexity (4119 functions or 14502 lines of input code of which: 105 error cases were found at transformation time, 104 error cases were found at verification time and 3910 were successfully verified).

II. Example

We illustrate our technique for modernizing a function and validating its correctness by using a simple example. The example is a simplified and anonymized version of a real function from the Danfoss code base.

The input program (given in Fig. 1) is a constraint-enforcing function. It checks that the input variables `selectedConfigParameter` and `selectedOptionParameter` have correct values with respect to each-other and to some constants `config1` and `option1`. We need to turn this program in a declarative constraint that must preserve the names of the input variables and must compute the same truth value as the imperative program.

Syntactically, the input program consists of a `switch` statement with two cases where one of them contains an inner `if`-statement. The program returns the value stored in the variable `result`. The corresponding declarative C++ expression (Fig. 2) obtained by running the TXL transformation is produced in several steps:

- The `switch`-statement is replaced with a nested `if-else` conditional statement.
- A series of simplifications are performed on the conditional statements, so we end up with a straightforward control flow.
- The conditional statement is finally reduced to a ternary expression of the form `e1 ? e2 : e3`.

Even for such a small example, checking that the transformation has preserved the semantics of the input program is non-trivial. We use KLEE to symbolically execute both the input and transformed output program to obtain all possible symbolic execution paths. Then the input and output programs

```
      (selectedConfigParameter == config1)
      ? (selectedOptionParameter == option1)
      : true
```

Fig. 3: The expected output.

are compared for equivalence, by checking whether there are corresponding matches between the obtained path conditions. If there exists a path condition in one program for which no matching path condition is generated in the other, then KLEE reports the corresponding path as a counter-example, i.e. as a witness of an execution that is possible in one program but not in the other.

For example, the path condition $\mathit{selectedConfigParmeter} \neq \mathit{config}_1 \wedge \mathit{selectedOptionParameter} = \mathit{option}_1 \wedge \mathit{result} = \mathsf{true}$ is generated for the input program in Fig. 1 but not for the transformed program in Fig. 2, which shows that these two programs are not semantically equivalent. By investigating the counter-example we are able to determine the expected output program shown in Fig. 3.

We were also able to trace the transformation rules that were applied to the input program and successfully diagnosed which rule produced erroneous output. The erroneous rule was an `if`-statement simplification rule which tried to simplify nested `if`-statements correctly, but forgot to take the `else`-branch into consideration. The verification technique was therefore very useful in accurately tracking the places in which bugs occur and an experienced transformation specialist would be able to fix this issue relatively swiftly.

III. Research Methodology

We followed the general framework of action research [5]. The study can also be seen as an exploratory case study, whose objective is to establish the feasibility of solving a concrete industrial problem.

A. Study Description

Objective: To establish the feasibility of transforming a large body of configuration code from an imperative implementation in C++ to a declarative model, in a manner that is *automatic*, *trustworthy* and *cost effective*.

The cost effectiveness is understood as being cheaper than reimplementing the code from scratch. The study is exploratory, as only the problem statement is known initially. The researchers have access to the input C++ code and to three Danfoss engineers/architects knowledgeable about the code, the context and the use case. The study combines engineering (transformation development) and research (designing semantically sound transformations and validating them). The case has been initiated as a pilot project for larger modernization activities in the same organization.

The first study proposition is to establish that freely available transformation and validation tools are sufficiently mature to execute this modernization process. The second study proposition—of greater interest from a research point of view—is to explain what kind of errors might appear in transformation projects involving complex code, even if implemented by experienced model transformation developers and language specialists.

The case: The input code consists of 4119 C++ functions from a configuration application. These functions encode dependencies, visibility constraints, and default values of approximately one thousand configuration parameters of a frequency converter. The use of C++, as opposed to C, is modest. No object-oriented aspects are used, except for member access and limited encapsulation. Most functions have straightforward control flow, without `goto` statements, loops and recursion. The most common statements are conditional branching and `switch` statements. The rare `for` loops all have a constant number of iterations. Other constructs include variable declarations and usage of local variables in arithmetic and comparison expressions, calls to pure functions (for instance for converting physical units), and casts between different types (both C-style casts and `static_casts`). There are few references to static and singleton member variables and functions, which act similarly to other function calls.

There are 14502 source lines of code (SLOC) in total that need to be modernized in the pilot project, and more similarly-looking configurators for other products waiting for modernization afterwards. As many as 3348 of the 4119 functions are already in expression form; these do not need to be modernized, but should not be broken by the modernization. The remaining 771 functions have 14.47 SLOC of code on average.

Goal of the modernization project: It has been decided that this code is suboptimal from a maintenance point of view and that it needs to be replaced by an off-the-shelf verifier using a declarative configuration model [6]. The model has to be trustworthy since the configuration code contains crucial domain information; missing configuration constraints could lead to creation of unsafe configurations by customers.

The code base in question is being actively worked on, so a manual transformation that puts the normal development on hold for a longer period is not possible. The modernization has to be automatic. Automation allows to limit the time when the code base is inaccessible for developers in two ways. First, the development of the modernization is done by implementing a transformation, which can be done in parallel with the evolution of the transformed code. Secondly, the transformation itself can be executed efficiently within minutes as opposed to weeks if it had to be executed manually.

Theory: In principle, it is not possible to know in advance whether refactoring the code in question preserves semantics. Validation of the transformation itself is generally undecidable. However, in recent years progress has been made by recognizing that many actual analysis problems appearing in engineering can be handled using incomplete and (partly) unsound [7] methods. This is true in particular for bug finding, which is our goal here. The effectiveness of the transformation is discussed in Sect. IV, and the effectiveness of the validation is discussed in Sect. V.

Existing model transformation validation technology (see Sect. IX) is insufficient for the needs of this case, as it focuses on preservation of structural properties of the transformed artifacts. In here we need to reason about equivalence of semantics of the transformed programs.

a) Research questions:

RQ1 Is it feasible to design the aforementioned transformation using off-the-shelf technology in a limited time?

RQ2 What are the main obstacles and challenges in designing and implementing the transformation? Are the transformation tools sufficiently efficient for the task?

RQ3 Can high assurance methods be used at acceptable costs to validate the transformation?

RQ4 What kind of errors are found in a transformation implemented by experts?

Questions RQ1–3 are interesting for companies looking into technology transfer in modernization projects, and for research stakeholders looking into setting up industry collaborations on software modernization. The last research question is more relevant for researchers in rewriting and model transformation. We are not aware of any studies of errors in realistic software transformations. Thus this work can be used to guide further research in quality assurance and verification of model and code transformations by formulating a hypothesis on what kind of problems are worth addressing.

Methods: We decided to address RQ1 and RQ2 by searching for the most effective way to implement a transformation. We elicited the requirements from the industry partner and evaluated a number of approaches and supporting technologies. A similar process was executed for RQ3. We recorded experiences during the process and report them in this paper. For RQ2 we have collected statistics about the effectiveness and efficiency of the transformation. For RQ3 we have measured the ratio of false positives and explained why they appear when following our validation methods. Regarding RQ4, we collected counterexamples from the validation process, classified them, and qualitatively analyzed them to understand what kind of errors arise in the transformation.

Study Participants: Three teams have been involved in the project: i) The industrial partner presented the software modernization problem, requirements and artifacts. Two engineers and one architect participated in the team. ii) The transformation team consisted of a transformation expert, a language semantics expert and a project leader. The transformation expert designed and implemented the transformation in dialog with the other members of the team. iii) The validation team consisted of a junior applied verification researcher (with 2 years of experience in verification), a junior PhD student in programming languages, and the same language semantics expert and project leader that were involved in designing the transformation.

B. Threats to Validity

Construct Validity: The industrial partner had selected the case and the problem they wanted to be solved, and the research team only had access to the above mentioned parts of the configurator. It may be possible that the researchers misunderstood some aspects of the software architecture, which would influence the validity of the results reported here; however, we believe that the impact of this would only be limited. First, the transformation is of substantial complexity, so even if it was misaligned with the requirements, it still sheds a lot of light on pragmatics of such transformations. More importantly, the validation method finds actual errors that are easy to confirm manually.

Internal Validity: Since the validation procedure has been developed in the same study in which it is evaluated, there is certainly a risk that it had overlooked some important errors. The transformation had been designed with a focus on effectiveness, with no thought of later verification in mind. The validation project is largely independent of the transformation project, and the key developers of the two parts have not communicated in any significant manner at all; several months have passed between the end of the transformation implementation project and the beginning of the validation project. The validation team mapped between the classes of bugs and the programming errors causing them, so there is a slight risk of misinterpretation. To minimize this risk, we have cross checked the result with other team members.

External Validity: Limited external validity is in the very nature of an individual study. For this reason we describe the properties of the case in detail. Simple C-like code with bounded for-loops is common in safety critical software, and we thus believe that the findings can generalize to additional modernization projects.

Reliability: The analyzed transformation errors can be biased towards the weaknesses of the particular development team. Note that the involved developers were experts in model transformations (more than 6 years of applied research experience) and verification (more than 6 years of experience with program and model verification), thus it is unlikely that the errors they made were trivial and would not happen to other developers.

Nevertheless, we report these findings only existentially (without quantifying them), providing a single data point for the research space where very little evidence is available so far. More studies are definitely needed in order to understand the nature of design errors in transformative and generative programming.

IV. Designing the Transformation

A. Design Principles Established for the Project

We found that implementing an automatic transformation is superior to a manual refactoring, as it allows to minimize down-time on the main development of the code base. The code to be modernized is only locked for the time it takes to run the transformation and it can be evolved freely while the transformation is being implemented and tested. This is somewhat different from the standard use case for transformations which is automating repetitive tasks in conversions of data, code or models. This transformation was meant to be executed only once, but automation was key to minimize disruptions in the regular development process.

Observation 1. Automatic transformations allowed us to decouple and parallelize the regular development and the modernization activities.

Translation of imperative C++ code into a declarative form in general may be very complex. However, our objective was much more modest: we only needed to handle the code at hand (and similar). Thus we settled on saving resources whenever possible by sacrificing generality. In fact, we even gave up transforming the entire code at hand, agreeing that a small number of complex fragments (involving loops) were simpler to modernize manually rather than designing rules that would handle them correctly from first principles. The manual migrations can be hard-coded into a transformation as special cases, so that the transformation execution remains automatic.

Observation 2. Since modernization is a one-off transformation it was economically beneficial to sacrifice generality, and instead focus on the code at hand whenever it simplifies things.

To control the lack of generality, the implementation should follow a *fail-fast* programming style [8], [9]. It should succeed on the inputs that it was designed for, but it should fail as early as possible on inputs that violate the assumptions made when sacrificing generality (e.g. arbitrary nesting of constructs in C++, or use of unexpected language elements). This was achieved by making preconditions for rules as precise as possible (so that rules are not applied when failing) and writing explicit assertions when possible (when a rule is in principle applicable, but it does not cover all the cases, for simplicity of development). Without the fail-fast programming we would have very limited trust that the transformation works correctly on hundreds of code fragments that we were not able to inspect manually.

Observation 3. The fail-fast programming approach helped to avoid implementing anything that is not strictly required for the modernization project to succeed, while retaining quality on the expected inputs.

Initially, we considered using type analysis and other semantic mechanisms (such as static single assignment form) to solve the task. However, it soon became clear that this would raise the complexity of the implementation considerably, and likely also lead to polluting the output with identifiers generated in the process (unfamiliar to developers). Full understanding of static semantics might only be required if one implements a general program rewriter. For an incomplete transformation of a known code base it seems much easier to work with syntactic transformations. Even typing and simple data flow information can be captured with a finite number of patterns if we only need to work with limited code base.

Observation 4. We found working with syntax much more effective for an ad hoc transformation task, than when using semantic data and semantically informed rewrites.

B. Tool Selection

Since the input language (C++) is rather complex, we understood early on that the transformation tooling should be driven not by our personal preferences, but by the availability of a C++ grammar. Thus we considered using existing open source compiler front-ends (for instance GCC[2]), or language tools (such as Eclipse CDT[3]). However, we found them challenging to use. Then we turned to transformation tools and found that both Spoofax [10] and TXL [2] have C++ grammars, though the

[2] https://gcc.gnu.org

[3] http://eclipse.org/cdt/

```
1 rule convert_simple_sel_stmt
2   replace [selection_statement]
3     'if '(EXP [expression] ')STMT [statement]
4   where not STMT [contains_selection_stmt]
5   where not STMT [is_compound_stmt]
6   construct TRUE_STMT [true_case_statement]
7     'TRUE ';
8   by '(EXP ')'? '(STMT ')': '(TRUE_STMT ')
9 end rule
```

Fig. 4: An example TXL rewrite from our project: a translation of a C++ conditional statement into a conditional expression.

grammar of the former was unmaintained and hard to migrate to modern versions of the tool. We ended up using TXL which has a good C++ grammar, handles ambiguity quite well and provides mechanisms for relaxing the grammar, making it easy to adapt to our needs. TXL is a standalone command-line tool, with few dependencies, so it took virtually no time to get it to run. In fact, simple proof of concept rewrites were working after only 4 hours of experiments with TXL.

Observation 5. Simplicity and the integration with the languages to be transformed influenced the selection of tools stronger than the properties of the transformation language or a rewriting paradigm.

C. Transformation Implementation

TXL is a rewrite tool that transforms syntax trees into syntax trees. It accepts two kinds of definitions: *grammars* and *transformations*. Grammars serve for parsing and unparsing. TXL works with one grammar at a time—in other words, the input and output grammars must be the same. To overcome this we selected a subset of C++ expression language as our target language (C++ expression language is sufficiently good to express declarative constraints over finite domain variables). To handle this format we needed to relax the C++ grammar only slightly to allow top-level expressions in C++. We also wrote a simple rule that validates whether the output program is indeed an expression in the subset of interest.

Transformation definitions specify how to rewrite a particular input syntax to output syntax. Figure 4 shows an example transformation rule, `convert_simple_sel_stmt`, from our project. All caps identifiers refer to syntax trees. The rule matches a conditional statement without an else clause (line 3) and translates it into a conditional expression (line 8). Lines 4–5 specify that the rule should fail if the guarded statement is either a compound statement or another conditional statement. Due to the grammar construction and the interaction with other rules, this means that the rewrite will only apply to conditional statements that themselves already guard a simple expression. In lines 6–7 a constant `true` expression is constructed, which fills in for the missing branch in the conditional expression.

The overall algorithm applied by the transformation is:

1) The program fragment is checked for format assumptions: all branches return a value, there are no loops and goto jumps, no calls to non-pure methods, etc. The transformation does not establish the purity of functions itself, but consults a white-list of names of pure functions provided as a parameter.
2) All preprocessor `#ifdef` directives in the program are cleaned up, and converted to ordinary `if` statements.
3) Local variable assignments are inlined in the following expressions in the right order (i.e. going from the last assignment to the first). When all references to the local variables are eliminated, their declarations are also removed.
4) All `switch` statements are converted to a series of `if` statements.
5) Series of sequential `if` statements are simplified into nested `if` statements, such that the fragments are reduced to a single root statement, all additional functionality being implemented through substatements of the root.
6) `if` statements are converted to ternary expressions and `return` statements are replaced by the expression they return.

Observation 6. We succeeded to implement a flow-aware syntactic transformation, including constant folding and variable inlining, which enabled us to produce code that uses the same identifiers, and reminiscent structure of the input programs.

Readability of the ultimate output is important in modernization projects, as it is expected that the developers will further evolve the generated code.

D. Basic metrics

The entire transformation (including grammar definitions, excluding white space and comments) spans 6515 lines of code. The core C++ grammar, which is provided as a resource from the TXL website, has 137 nonterminal rules (595 lines of code). In addition, 98 grammar rules are defined or redefined in the transformation to adapt the grammar to our needs.

The transformation has 468 definitions (rules or functions) in total, where 171 of the definitions are function definitions and the remaining 297 are rule definitions (as opposed to functions, rules are called repeatedly until a fixed point is reached).

It took 3 months full time work of experienced software developer to implement this transformation (including learning TXL, domain understanding, unit testing and meetings with the industrial partner). The cost is deemed acceptable, especially given that the company has several more products to modernize, that include configuration code, for which the transformation would be largely reusable.

The transformation execution lasts 30 minutes on the 4119 functions out of which 105 functions are not handled—the transformation reported errors, marking that these functions contain special cases and should be migrated manually to keep the whole process cost-effective.

V. Validating the Transformation

A. Verification challenges

TXL is a very powerful transformation language that can express arbitrary computation using deep pattern matching, complex side conditions, global state and rewriting. Individual rules and functions of a transformation are non-modular and might intermediately break syntactic and semantic correctness properties. This makes it hard to verify individual rules. Instead, it is important that the transformation is validated as a whole.

Observation 7. Our transformation was written in a non-modular fashion (as most transformations we have seen). This made it hard to verify the transformation rules individually.

Our transformation makes heavy use of the generic programming and dynamic reparsing features which allow serialization of abstract syntax trees to textual syntax and then reinterpretation as other syntactic structures. An external script controls the sequence of transformation steps, adding even more complexity to the algorithm. Furthermore, the rules are designed in an ad hoc fashion specifically to handle patterns present in the use case. For example, we use the transformation rule $\texttt{if}\,(E)\,\texttt{return}\,\mathit{true} \rightarrow \texttt{return}\,E$, which is not correct in general, but it is correct under assumption that the `if` statement occurs last in the `main` function. To combat this complexity, our verification method treats the transformation as a black-box and checks that the input and corresponding transformed output agree semantically. This works since the size of input is manageable and we do not expect the transformation to be generally applicable in other unrelated settings.

Observation 8. We were able to treat a complex transformation as a black-box, reducing the validation to checking whether the provided input and transformed output agree according to specific semantic properties.

B. Approach

Ordinary TXL rules and functions are constrained to replacing well formed syntactic trees with newly built ones, effectively making it impossible for TXL programs to produce syntactically incorrect output. The correctness criterion of the transformation was to produce *semantically* equivalent C++ programs. Therefore our validation technique must be able to reason about the semantics of C++ programs.

We considered several abstract interpretation and analysis techniques and we found that symbolic execution [3] is able to build a very precise semantic model. We decided on using the precise symbolic executor KLEE [4] which handles the majority of features used in the code at hand, and integrating it in the automation process proved to be cost-effective.

One of the challenges of using KLEE is that it requires the input code be compiled to LLVM [11] intermediate representation (LLVM−IR), including all external libraries (otherwise the symbolic execution may not terminate or provide incomplete results). However, our sample code-base consisted of individual functions which called external functions with unknown implementations. Therefore, we had to close the code with suitable instrumentation:

1) We created stubs for unknown functions—ordinary, static member and singleton member functions alike—such that a set of arguments is matched to the same symbolic result on every call.
2) We created stubs for the data structures with straightforward constructors (that initialize all members) and structural equality comparison operations.

```
bool defined(int p) {
  static node<int, bool> *results;
  static int* counter = new int(0);
  bool* val = new bool;
  if(!getResult(&results, &results, p, counter, val)) {
    char symbolicname[40];
    sprintf(symbolicname, "defined%d", *counter);
    *val = klee_range(0, 2, symbolicname);
  }
  return *val;
}
```

Fig. 5: A stub for an unknown Boolean function.

Function stubs are created in the *symbolic function* [12] style. For each stub an execution table which matches input arguments with symbolic return values is allocated. When the function is called, it looks up the arguments in the execution table: if they are found, it returns the same symbolic result as before; otherwise, a new record that stores the arguments along with a fresh symbolic result variable is created in the table.

Figure 5 shows a stub for an unknown Boolean function. The static execution table `results` and call counter are reused for all calls to the function. The function `getResults` looks up the input argument `p` in the execution table. If found, the existing symbolic variable is returned through the pointer `val`. Otherwise, the call counter is incremented and `val` points to a new memory address which is made symbolic with `klee_range` and returned.

Observation 9. We were able to use off-the-shelf tools to perform semantic verification of programs, with a moderate amount of effort required to pre-process the input to the tool.

C. Research questions

In the experiment we answer the following detailed questions, refining RQ3:

RQ3.1 How large a part of the transformed code base can be verified automatically?
RQ3.2 How much additional effort would it require to verify the rest of the code base?
RQ3.3 How can our verification effort be generalized to other similar modernization projects?

D. Method

We address RQ3.1 in Section VI by running the verification procedure on the input and transformed output, and reporting and classifying the results. Section VIII discusses the challenges (and solutions to these challenges) that appeared during verification (RQ3.2) and what parts of our verification procedure is generalizable to other transformation tools and projects (RQ3.3).

The validation execution lasts 7 minutes on all the transformed functions out of which 3348 are evaluated trivially to pass verification–the output is identical to the input.

VI. Bug Analysis

A. Bugs in Numbers

Out of the 4119 functions in the code base there were 771 which could not be validated trivially (the output was not identical to the input). We present the statistics in Table I.

Observation 10. It was possible to analyze a substantial amount of the modernized code automatically, and only 20 corner cases were left to be handled manually.

We analyze the bugs found by verification (3 & 4b) below. In Section VIII we will discuss the types of spurious counter-examples (4c), the unhandled cases due to design limitations (5), and propose solutions for these issues.

B. Analysis of Bugs Found by Verification

Our technique had identified seven bugs present in the transformation. While these bugs varied in nature, they had one important thing in common: they were all related to execution semantics and would have been hard to find with a syntactic check or simpler static semantics check (like a type system).

Observation 11. When the code base of our modernization project reached a certain complexity it became infeasible to find all bugs through expertise and unit testing. Validation of semantics was essential to ensure that the output code worked correctly.

Bug 1: *Function call is dropped in some paths.* Perhaps the most widespread output errors were missing function calls—absent in the output expression but present in the original code. This happened with a variety of calls to different functions, and could happen multiple places in the output expression; furthermore, calls to the same function might still be present in other branches of the output expression.

The bug was caused by an *incomplete rewrite rule*. When a rule matches a functional call in a return statement and forgets to reinsert the function call on replacement.

Bug 2: *Structure replaced by a constant integer.* Another simple bug is the one where the input declares a variable of a class type, calls its object initializer with multiple arguments and returns it. Here, the transformation seems to return the first argument given to the variable—which is often an integer and therefore having incompatible type—instead of the whole object.

This bug also happens due to *misuse of deep pattern search* and a *broken rule assumption*. It happens when trying to inline a variable with its initial value, but expecting the variable to be of a simple type. Since there is a use of a deep pattern search to extract an expression from the initialiser, it will simply pick the first one (ignoring the others) and the rest of the transformation would continue without noticing the bug.

TABLE I: Erroneous transformation cases caught by each step of the validation process.

	Step	#Cases
1	Failing transformation precondition (not handled, requiring manual inspection)	105
2	Failing silently due to unhandled syntactic structures (caught statically during preliminary steps of verification)	3
3	Caught by C++ compiler	3
4	Checked for equivalence using KLEE	640
4a	Validated being equivalent	562
4b	Concrete bug cases with provided counter-examples	50
4c	False positives with spurious counter-examples (due to over-approximation of functions, and representation mismatch)	28
5	Unhandled cases containing assertions (intentional, due to design limitations of the validation technique)	20

Bug 3: *Conditional branches are dropped.* This bug caused the transformation to ignore all branches following a nested `if`-statement (also referred to in Section II). It was caused by *incomplete rewrite rules*. The rewrite rule matches a nested conditional followed by other branches, and then rewrites the conditional correctly but forgets to handle the other branches.

Bug 4: *The unexpected exceptions.* This bug is surprising and happens mostly in very large functions with complex nesting of conditional control flow. While the input function seems total and returns a correct result on all paths, the transformation produces an output which contains a branch that throws an exception stating that the branch should be invalid.

This bug happens due to *overconstrained pattern matching* and *broken rule assumption*. When a sequential composition of nested conditional followed by a return statement is matched by the transformation, it tries to put the final return statement inside the previous conditionals. However, in this case the pattern was overconstrained and so it did not match the form of input it was given; later, when the transformation tries to convert the statement to an expression it finds a branch with no `return` statements and replaces it with a `throw` statement because it did not expect this case to be possible (a part of the fail-fast approach).

Bug 5: *Use of undeclared variables.* This bug is the only one that appears during compilation. The original input contained declarations to local variables that were not inlined correctly and the transformation removed all local declarations—but retained references to their respective identifiers—leaving compiler errors.

This bug occurs due to a combination of *dynamic reparsing capabilities* and *wrong target type* in expression. To control the number of iterations of inlining substitution, the transformation replaces variable nodes with the string representation of their assigned expressions, using the textual output capabilities of TXL. This ensures that the substitution terminates, but might also be incorrect if the transformation has not finished migrating the serialised subtrees. In general, it is caused by challenges in implementing a semantic operation (i.e. inlining) syntactically.

Bug 6: *Negation dropped in result.* The simplest bug found by the KLEE-based verifier is where the transformation had transformed the whole input correctly except a negation operation which was missing in the output. This bug occurs due to *misuse of deep pattern search* of TXL. The rewrite rule searches for a more specific object type than necessary, making it ignore more complex objects that do not fit to the expected pattern.

Bug 7: *Conditional with error code assignment dropped.* One interesting bug is where the input has a function which contains a conditional statement that assigns a value to an error code pointer variable, in addition to returning a value (both in the conditional and outside). In this case, the transformation will produce output that will completely remove the conditional branch and only keep the final return value, which makes the function produce the wrong result.

This bug happens due to the *dynamic reparsing capabilities* and *eager removal of source data*. It originates in the inlining phase where some abstract syntax is broken by wrongly inserted textual syntax, and subsequently a rule that removed empty conditional branches was applied.

Bug 8: ***Variable declarations without assignment not handled.*** This bug was caught statically in the cases when the transformation finished, but the output was empty. Similar to Bug 2, a combination of a *broken rule assumption* and *misuse of deep pattern search* was the cause of this bug. In ordinary circumstances, the transformation tries to inline all locally declared variables with their assigned expressions and then remove the declarations. However, in these cases the declaration and initialization of the local variables were situated in separate statements. The declaration removal rule used deep search to identify statements which contained local variables and since program consistent of one large if-statement containing the assignments, it was completely removed.

Classification summary: Most cases were affected by bug 1 where there were 23 cases in total, and followed by bug 2 which had 15 cases in total; both of which were simple in nature. This is perhaps unsurprising since function calls and object initialisations are common constructs in C++, and a simple mistake in the transformation of these features will therefore affect a large number of analyzed functions. The more interesting (complex) bugs 3 and 4 had 5 cases in total each. This type of bugs often appeared in larger files with a complex nesting of conditionals, and would therefore have been hard to immediately spot manually or with simpler unit tests. Finally, the remaining bugs (5, 6, 7, 8) had 3, 1, 1 and 3 cases in total, respectively. These errors represent issues that appear to be corner cases that were either not caught by the preconditions of the transformation, or occurred where an intermediate assumption of the transformation was wrong.

Observation 12. *Simple bugs hit wide, complex bugs hit deep.* Simple semantic errors affected a large number of functions while complex errors were found in a few but bigger functions.

VII. Formal Justification of the Procedure

A. Concrete execution

The transformation translates many functions individually. Each of them needs to be translated in a semantics preserving manner. We view functions (or programs in general) as input–output relations.

Definition 1. A *program* P is a set of imperative instructions that state how to calculate the designated output variable ret from a set of input variables $Var_{\text{in}} = \{i_1, \dots, i_k\}$.

Definition 2. A *concrete state* (store) σ is a *function* mapping program variables Var into values Val, i.e. $\sigma : Var \to Val$. The values Val are constants from ordinary C++ types: Boolean, bounded integer, float, etc.

A concrete execution starts with an *initial state*, σ_{in}, where all input variables are assigned some initial values. During the execution of the program, the effect of executing each statement s in a state σ produces a successor state σ', written as $\sigma \xrightarrow{s} \sigma'$. When there are no statements left to execute, the program reaches a final state σ_{out}, in which the value of the output variable ret is well-defined.

Definition 3. A *concrete execution path* $\pi = \sigma_{\text{in}}, \sigma_1, \dots \sigma_{\text{out}}$ of the program P is a sequence of states, such that σ_{in} is an initial state, every next state in the sequence is obtained by sequentially executing statements from P one by one, and σ_{out} is the final state.

Definition 4 (Concrete program path semantics)**.** The *path semantics* of program P—called $[\![P]\!]_{trace}$—is defined to be the set of all valid concrete execution paths π of P.

Definition 5 (Denotational program semantics)**.** The *denotational semantics* of program P is a partial function $[\![P]\!] : Val^k \rightharpoonup Val$ defined by: $[\![P]\!](\sigma_{\text{in}}(i_1), \dots, \sigma_{\text{in}}(i_k)) = \sigma_{\text{out}}(ret)$, for any concrete execution path $\pi \in [\![P]\!]_{trace} = (\sigma_{\text{in}}, \dots, \sigma_{\text{out}})$.

Definition 6 (Semantic equivalence)**.** Two programs P and P' are *semantically equivalent*, written $P \sim P'$, if for any collection of values $v_1, \dots, v_k$ it holds: $[\![P]\!](v_1, \dots, v_k) = [\![P']\!](v_1, \dots, v_k)$.

Determining the semantic equivalence of programs using concrete path semantics is infeasible due to the immense range of input values. Instead, we use symbolic execution to cluster the input values using a set of constraints called path conditions.

B. Symbolic execution

In *symbolic execution* the program does not assign values to its variables; instead, it assigns symbolic expressions containing uninterpreted symbols abstractly representing user-assignable values in a concrete execution of the program. For ease of notation, we will use capital letters $X, Y, \dots$ to represent uninterpreted symbols, and we use *Sym* to represent the set of all of these symbols. In the initial execution state, each possible input variable i is usually assigned a corresponding unique symbol I.

For example, let x and y be input integer variables. The concrete semantics of $ret = x{+}y$ is the set $\{([x \mapsto 0, y \mapsto 0], [x \mapsto 0, y \mapsto 0, ret \mapsto 0]), ([x \mapsto 0, y \mapsto 1], [x \mapsto 0, y \mapsto 1, ret \mapsto 1]), \dots\}$, where initial states are all possible assignments of integer values to x and y. However, the symbolic path semantics of $ret = x + y$ will contain only one symbolic execution path $([x \mapsto X, y \mapsto Y], [x \mapsto X, y \mapsto Y, ret \mapsto X + Y])$, where X and Y are symbols.

Symbolic execution confounds a set of different concrete paths into one by following all branches whenever a branching or looping statement is encountered. In the same time, for each branch it maintains a set of constraints called the *path condition*, which must hold on the execution of that path.

Definition 7. A *symbolic expression se* from the set *SExp* can be built out of constant values from Val, symbolic values from Sym, and arithmetic-logic operations.

Definition 8. A *symbolic state* $\sigma^{\#}$ is a function mapping program variables Var into symbolic expressions $SExp$, i.e. $\sigma^{\#} : Var \to SExp$. The initial symbolic state $\sigma^{\#}_{\text{in}}$ maps all input variables $i \in Var_{\text{in}}$ into a fresh symbolic value $I \in Sym$.

Definition 9 (Constrained symbolic state)**.** A *constraint* is a Boolean symbolic expression. A *constrained symbolic state* is a pair $\langle \sigma^{\#}, sb \rangle$, which constraints the symbolic expressions in $\sigma^{\#}$ with a Boolean symbolic expression sb.

Definition 10 (Symbolic execution path)**.** A *symbolic execution path* of the program P is a sequence of constrained symbolic states $(\langle\sigma^{\#}_{\text{in}}, \text{true}\rangle, \langle\sigma^{\#}_{1}, sb_1\rangle, \ldots \langle\sigma^{\#}_{\text{out}}, sb_{\text{out}}\rangle)$, where the initial state $\sigma^{\#}_{\text{in}}$ is unconstrained, and the constraint produced for the final state, sb_{out}, represents the path condition.

It is notable that the resulting set of symbolic execution paths partitions the set of concrete execution paths. For the program that computes the absolute value of an integer variable i, there are two different paths returned by symbolic execution: $\big(\langle[i \mapsto I], \text{true}\rangle, \langle[i \mapsto I, ret \mapsto I], I \geq 0\rangle\big)$ and $\big(\langle[i \mapsto I], \text{true}\rangle, \langle[i \mapsto I, ret \mapsto -I], I < 0\rangle\big)$. If the initial value of i is non-negative, then the return value is the symbolic expression I; otherwise, the return value is $-I$. Hence, the set of all concrete execution paths (determined by the input values of i) has been partitioned in two sets: those for which $i \geq 0$ holds and those for which $i < 0$ holds.

Proposition 1. For each concrete execution path $\pi = (\sigma_{\text{in}}, \sigma_1, \ldots \sigma_{\text{out}})$ of the program P, there exists the corresponding symbolic execution path $\pi^{\#} = (\langle\sigma^{\#}_{\text{in}}, \text{true}\rangle, \ldots \langle\sigma^{\#}_{\text{out}}, sb_{\text{out}}\rangle)$, such that $\sigma^{\#}_{\text{in}} = [i_1 \mapsto I_1, \ldots, i_k \mapsto I_k]$, $\sigma_{\text{out}}(ret) = \sigma^{\#}_{\text{out}}(ret)[I_0 \mapsto \sigma_{\text{in}}(i_0), \ldots, I_k \mapsto \sigma_{\text{in}}(i_k)]$, and $sb_{\text{out}}[I_0 \mapsto \sigma_{\text{in}}(i_0), \ldots, I_k \mapsto \sigma_{\text{in}}(i_k)]$ is true.

Proof: See online appendix[4]. ■

Theorem 1. *Two programs P and P' are semantically equivalent $P \sim P'$ iff for each valuation $V \in Val$ it holds:*

$$\Big(\bigvee_{1..m}^{j} sb^{j}_{\text{out}} \wedge \sigma^{\#,j}_{\text{out}}(ret) = V\Big) \iff \Big(\bigvee_{1..m'}^{i} sb'^{i}_{\text{out}} \wedge \sigma'^{\#,i}_{\text{out}}(ret) = V\Big)$$

where $(\langle\sigma^{\#,1}_{\text{in}}, \text{true}\rangle, \ldots \langle\sigma^{\#,1}_{\text{out}}, sb^{1}_{\text{out}}\rangle)$, $\ldots$, $(\langle\sigma^{\#,m}_{\text{in}}, \text{true}\rangle, \ldots \langle\sigma^{\#,m}_{\text{out}}, sb^{m}_{\text{out}}\rangle)$ *are symbolic paths of P, and* $(\langle\sigma'^{\#,1}_{\text{in}}, \text{true}\rangle, \ldots \langle\sigma'^{\#,1}_{\text{out}}, sb'^{1}_{\text{out}}\rangle)$, $\ldots$, $(\langle\sigma'^{\#,m'}_{\text{in}}, \text{true}\rangle, \ldots \langle\sigma'^{\#,m'}_{\text{out}}, sb'^{m'}_{\text{out}}\rangle)$ *are symbolic paths of P'.*

Proof: It follows from Prop. 1 and Def. 6. ■

VIII. Discussion

A known challenge of action research in software engineering is unreliability of academic tools. Tools developed by academic partners are rarely adopted in companies due to a lack of reliable support service (unless the industrial partner can reasonably take over the tool maintenance itself). This is apparently a much smaller problem in software modernization projects, as the tools are only used for a short period. We note that modernization is a very good domain for research-based tools, where the actual adoption is likely easier than elsewhere.

The applied design and validation principles translate easily to other program and model-transformation languages. All such languages work at the level of (abstract) syntax, so designing the rewriter syntactically is achievable. Since we validate the output of the transformation against the input, but the transformation itself is treated as a black-box, the method is oblivious to the choice of the transformation language. Because of that, it could work even for manual transformations, for instance for manual refactoring. However it is unclear, whether the identified bugs are specific to this case, this input and output languages, and TXL.

We have met the following technical verification challenges:

1) Representation of Boolean expressions: In C++ any integer valued expression can be used as a logical condition (inside `if`-statements etc.), and so any non-zero value would count as true and zero would count as false. If an integer variable `a` is used only as a logical condition both in the input and output programs it would be pragmatically fine. However, our transformation contains simplification rules which convert statements of form `if (a) return true; else return false;` to `return a;` which clearly has different semantics. In cases where we are certain that specific integer variables are only used as conditionals we instruct KLEE to assume that these variables have values lying in range $[0, 2)$.

2) Over-approximation of Function Semantics: Because we do not know the implementation of all external functions, we use an over-approximated function call representation with stubs that simply map equal parameters to equal unique symbolic results. However, if any of these functions actually had equivalent implementations and we used a different stub for each one, calling the two stubs with the same parameters would result in distinct return values. This led to a number of false positives where KLEE decided that the input and output programs were not equivalent. This was solved by using the same stub for functions which were known to be identical *a priori*.

3) Assertions: When KLEE meets a C++ assertion that has a failing condition on a possible path, it will immediately halt execution for that particular path. This concretely means that it will never check whether the input and output functions have the same results, or in this case rather both fail. Instead of using the default assertion function, one could instead use a stub that throws a recoverable error (rather than halt) on failing conditions; thereafter, one could check whether both the input and output programs failed on the same paths and if they did one could consider the paths to be equal. Our code base contains 20 cases affected by this limitation, but implementing the suggested solution was not feasible in the allocated time.

IX. Related Work

Translation validation [13] is a verification technique for translator tools (compilers, code generators). It requires a common semantic framework for the representation of the source code and the generated target code, a formalization of the notion of *correct implementation* and a method which allows to prove that one model of the semantic framework, representing the produced target code, correctly implements another model which represents the source. Our approach is aligned with translation validation in the sense that it validates concrete translations instead of the transformation tool or algorithm. The path conditions produced by KLEE are a semantic framework of the compared C++ programs. KLEE uses an SMT solver to prove the equivalence of the path conditions and provides a counter example when they are not equivalent.

Other KLEE extensions aim to solve the same problem. UC–KLEE [14] can save and load program states so it can execute two programs in the exact same memory context. By comparing memory states at program termination UC–KLEE offers a more precise equivalence test which also covers programs that terminate unexpectedly (crash). The task would

[4] http://www.itu.dk/people/afla/files/ase-2015-appendix/prop1-proof.html

have been easier to solve with UC−KLEE, but we used regular KLEE because there was no release of UC−KLEE available.

KLEE−FP [15] proves the equivalence of two symbolic floating point expressions by first applying a series of expression canonicalization rules, and then syntactically matching the two expressions, whereas regular KLEE silently concretizes floating point values to zero. However, when we used it to validate our transformations it failed to identify some of the bugs that regular KLEE had previously found and did not reveal any new bugs. We did not investigate this further.

Proof of program transformation correctness is often studied in conjunction with optimization. Parametrized equivalence checking (PEC) [16] uses a form of translation validation which tries to find a bisimulation between the control flow graphs of the original and optimized programs in a way that parameterizes over some program components (such as expressions, statements and declarations). Other work [17] specifies the optimisation rewrite rules as temporal logic formulas, and proves the correctness of the transformation manually. Both techniques lack tools that support them.

Currently, there exist various robust techniques for verifying preservation of properties by model transformations [18]–[21]. These techniques use (bounded) model finders and SMT solvers to verify the preservation of structural properties of model transformation rules. However, our interest lies in checking behavioral equivalence between the input and output, which is significantly more complex to check than structural properties, and thus not supported by the presented techniques.

A similar automation-based modernization effort is presented in a joint project by Semantic Designs (SD)[5] and Boeing[6] [22] which uses Semantic Designs's commercially-available transformation and analysis tool DMS [23] to convert an old component-based C++ codebase to a standardized CORBA [24] architecture. Our case study shows that it is possible to do automation-based modernization relying solely on freely available tools (TXL, etc.), and we were able to present additional useful observations. More importantly, our evaluation effort is significantly larger and includes validation in addition to rigorous testing and code reviews, which we have shown to be useful since it caught many subtle bugs and corner cases that were missed earlier in the process.

A series of papers [25]–[27] discuss a similar case study that aims to design and verify a model transformation for modernizing an existing collection of proprietary models such that they conform to the standardised AUTOSAR [28] format. The transformation [25] was initially encoded in a (non-Turing complete) subset of the ATL model transformation language [29] and then verified for structural properties [26]. The same verification effort was then repeated [27] more efficiently by by symbolically executing a version of the transformation re-encoded in DSLTrans [30]. While these verification tools and the presented case study have significant contributions to the model transformations community, they were not applicable in our study due to the difference in expressiveness between TXL and the verified non-Turing-complete subset of ATL/DSLTrans, and the complexity of the property we wanted to check (behavioral equivalence versus structural properties).

X. Conclusion

We have reported experiences from an industrial software modernization project, including requirements elicitation for a code-to-model transformation, designing and implementing the transformation, and verifying the correctness of the transformation against semantic properties using symbolic execution. The project allowed us to derive observations regarding automation in software modernization as well as choices and challenges in design and validation of modernization transformations. Probably, the biggest technical challenge seen in the transformation is that it seems impossible to reason about it inductively (rule-by-rule), because the intermediate transformation results are incorrect by design. Moreover, our method is oblivious to complexity of the transformation language, but since it is property driven, it depends strongly on the properties of the *transformed* languages. Observe though that, a white-box method would be vulnerable to both kinds of complexity.

Our validator finds many semantic bugs that have been missed by unit tests of an experienced transformation developer. These errors would be very difficult to find without verification. For each bug, the tool provides a counter-example consisting of execution paths on which the input and transformed programs differ. This way, we have obtained helpful debugging information, which can be used to improve and correct the transformation. We group the identified bugs into seven classes. Our paper is, to the best our knowledge, the first ever study reporting errors from a realistic transformation project, including validation using real bugs (as opposed to planted bugs) and operational semantic properties of input and output (as opposed to syntactic and typing properties).

Acknowledgements We thank Karl Potratz for introducing us to the subject system, Rolf Helge-Pfeiffer for implementing the transformation and Claus Brabrand for discussions on its design. The project was supported by The Danish Council for Independent Research under a Sapere Aude project, VARIETE, and by ARTEMIS JU under grant agreement n°295397 together with Danish Agency for Science, Technology and Innovation.

References

[1] J. Penix, "Big problems in industry (panel)," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, p. 3. [Online]. Available: http://dx.doi.org/10.1109/ASE.2013.6693060

[2] J. R. Cordy, "The TXL source transformation language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, 2006. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2006.04.002

[3] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[4] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[5] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering*. Springer, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29044-2

[6] A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, *Knowledge-based Configuration: From Research to Business Cases*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014.

[5] https://www.semanticdesigns.com

[6] http://www.boeing.com

[7] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundness: a manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015. [Online]. Available: http://doi.acm.org/10.1145/2644805

[8] J. Gray, "Why do computers stop and what can be done about it?" in *Fifth Symposium on Reliability in Distributed Software and Database Systems, SRDS 1986, Los Angeles, California, USA, January 13-15, 1986, Proceedings*. IEEE Computer Society, 1986, pp. 3–12.

[9] J. Shore, "Fail fast," *IEEE Software*, vol. 21, no. 5, pp. 21–25, 2004. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MS.2004.1331296

[10] L. C. L. Kats and E. Visser, "The spoofax language workbench: rules for declarative specification of languages and ides," in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, W. R. Cook, S. Clarke, and M. C. Rinard, Eds. ACM, 2010, pp. 444–463. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869497

[11] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. [Online]. Available: http://dx.doi.org/10.1109/CGO.2004.1281665

[12] R. Corin and F. A. Manzano, "Efficient symbolic execution for analysing cryptographic protocol implementations," in *Engineering Secure Software and Systems - Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings*, ser. Lecture Notes in Computer Science, Ú. Erlingsson, R. Wieringa, and N. Zannone, Eds., vol. 6542. Springer, 2011, pp. 58–72. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19125-1_5

[13] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, ser. Lecture Notes in Computer Science, B. Steffen, Ed., vol. 1384. Springer, 1998, pp. 151–166. [Online]. Available: http://dx.doi.org/10.1007/BFb0054170

[14] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 669–685. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_55

[15] P. Collingbourne, C. Cadar, and P. H. J. Kelly, "Symbolic crosschecking of floating-point and SIMD code," in *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*, C. M. Kirsch and G. Heiser, Eds. ACM, 2011, pp. 315–328. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966475

[16] S. Kundu, Z. Tatlock, and S. Lerner, "Proving optimizations correct using parameterized program equivalence," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, M. Hind and A. Diwan, Eds. ACM, 2009, pp. 327–337. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542513

[17] D. Lacey, N. D. Jones, E. V. Wyk, and C. C. Frederiksen, "Compiler optimization correctness by temporal logic," *Higher-Order and Symbolic Computation*, vol. 17, no. 3, pp. 173–206, 2004. [Online]. Available: http://dx.doi.org/10.1023/B:LISP.0000029444.99264.c0

[18] F. Büttner, M. Egea, and J. Cabot, "On verifying ATL transformations using 'off-the-shelf' SMT solvers," in *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds., vol. 7590. Springer, 2012, pp. 432–448. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33666-9_28

[19] F. Büttner, M. Egea, J. Cabot, and M. Gogolla, "Verification of ATL transformations using transformation models and model finders," in *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, ser. Lecture Notes in Computer Science, T. Aoki and K. Taguchi, Eds., vol. 7635. Springer, 2012, pp. 198–213. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34281-3_16

[20] F. Büttner, M. Egea, E. Guerra, and J. de Lara, "Checking model transformation refinement," in *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, K. Duddy and G. Kappel, Eds., vol. 7909. Springer, 2013, pp. 158–173. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38883-5_15

[21] X. Wang, F. Büttner, and Y. Lamo, "Verification of graph-based model transformations using alloy," *ECEASST*, vol. 67, 2014. [Online]. Available: http://journal.ub.tu-berlin.de/eceasst/article/view/943

[22] R. L. Akers, I. D. Baxter, M. Mehlich, B. J. Ellis, and K. R. Luecke, "Case study: Re-engineering C++ component models via automatic program transformation," *Information & Software Technology*, vol. 49, no. 3, pp. 275–291, 2007. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2006.10.012

[23] I. Baxter, C. Pidgeon, and M. Mehlich, "DMS ®: program transformations for practical scalable software evolution," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, May 2004, pp. 625–634.

[24] J. Siegel, Ed., *CORBA 3 fundamentals and programming*, 2nd ed. New York: OMG Press, John Wiley & Sons, 2000.

[25] G. M. K. Selim, S. Wang, J. R. Cordy, and J. Dingel, "Model transformations for migrating legacy deployment models in the automotive industry," *Software and System Modeling*, vol. 14, no. 1, pp. 365–381, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0365-1

[26] G. M. K. Selim, F. Büttner, J. R. Cordy, J. Dingel, and S. Wang, "Automated verification of model transformations in the automotive industry," in *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, ser. Lecture Notes in Computer Science, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. J. Clarke, Eds., vol. 8107. Springer, 2013, pp. 690–706. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41533-3_42

[27] G. M. K. Selim, L. Lucio, J. R. Cordy, J. Dingel, and B. J. Oakes, "Specification and verification of graph-based model transformation properties," in *Graph Transformation - 7th International Conference, ICGT 2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings*, ser. Lecture Notes in Computer Science, H. Giese and B. König, Eds., vol. 8571. Springer, 2014, pp. 113–129. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09108-2_8

[28] S. Bunzel, "AUTOSAR - the standardized software architecture," *Informatik Spektrum*, vol. 34, no. 1, pp. 79–83, 2011. [Online]. Available: http://dx.doi.org/10.1007/s00287-010-0506-7

[29] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 31–39, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.08.002

[30] B. Barroca, L. Lucio, V. Amaral, R. Félix, and V. Sousa, "Dsltrans: A turing incomplete transformation language," in *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, B. A. Malloy, S. Staab, and M. van den Brand, Eds., vol. 6563. Springer, 2010, pp. 296–305. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19440-5_19

Paper C

Effective Analysis of C Programs by Rewriting Variability

Alexandru F. Iosif-Lazar[a], Jean Melo[a], Aleksandar S. Dimovski[a], Claus Brabrand[a], and Andrzej Wąsowski[a]

a IT University of Copenhagen, Denmark

Abstract Context. Variability-intensive programs (program families) appear in many application areas and for many reasons today. Different family members, called variants, are derived by switching statically configurable options (features) on and off, while reuse of the common code is maximized.

Inquiry. Verification of program families is challenging since the number of variants is exponential in the number of features. Existing single-program analysis and verification tools cannot be applied directly to program families, and designing and implementing the corresponding variability-aware versions is tedious and laborious.

Approach. In this work, we propose a range of variability-related transformations for translating program families into single programs by replacing compile-time variability with run-time variability (non-determinism). The obtained transformed programs can be subsequently analyzed using the conventional off-the-shelf single-program analysis tools such as type checkers, symbolic executors, model checkers, and static analyzers.

Knowledge. Our variability-related transformations are *outcome-preserving*, which means that the relation between the outcomes in the transformed single program and the union of outcomes of all variants derived from the original program family is *equality*.

Grounding. We present our transformation rules and their correctness with respect to a minimal core imperative language IMP. Then, we discuss our experience of implementing and using the transformations for efficient and effective analysis and verification of real-world C program families.

Importance. We report some interesting variability-related bugs that we discovered using various state-of-the-art single-program C verification tools, such as Frama-C, Clang, LLBMC.

ACM CCS 2012

- *Software and its engineering → Software creation and management Software verification and validation; Software notations and tools Formal language definitions;*

Keywords Program Families, Variability-related Transformations, Verification, Static Analysis

The Art, Science, and Engineering of Programming

Submitted September 1, 2016

Published January 27, 2017

DOI 10.22152/programming-journal.org/2017/1/1

In *The Art, Science, and Engineering of Programming*, vol. 1, no. 1, 2017, article 1; 25 pages.

1 Introduction

Many software systems today are variability intensive. They permit users to derive a custom variant by choosing suitable configuration options (features) depending on their requirements. There are different strategies for implementing variational systems (program families) [11]. Still, many popular industrial program families from system software (e.g. Linux kernel) and embedded software (e.g. cars, phones, avionics) domains are implemented using annotative approaches such as conditional compilation. For example, #ifdef annotations from the C-preprocessor are used to specify under which conditions, parts of the code should be included or excluded from a variant.

Due to the increasing popularity of program families, formal verification techniques for proving their correctness are widely studied (see [35] for a survey). Analyzing program families is challenging [29]. From only a few compile-time configuration options, exponentially many variants can be derived. Thus, for large variability-intensive software systems, any brute-force approach that derives and analyzes all variants individually one by one using existing single-program analysis tools is very inefficient or even infeasible. Recently, many dedicated family-based (variability-aware) analysis tools have been developed, which operate directly on program families. They produce results for all variants at once in a single run by exploiting the similarities between the variants. Examples of successful family-based analysis tools are applied to syntax checking [25, 20], type checking [24, 8], static analysis [7, 6], model checking [10, 14], etc. Although they are more efficient than the brute-force approach, still their design and implementation for each particular analysis and language is tedious and error prone. Often, these family-based tools are research prototypes implemented from scratch. So it is very difficult to re-implement all optimization algorithms in them that already exist for their single-program industrial-strength counterparts, which have been under development for several decades.

Another approach for efficient variability-aware verification would be to replace compile-time variability with run-time variability (or non-determinism) [37]. In particular, in this work we consider a class of variability-related transformations that transform a program family into a single program, whose outcomes are equal to the union of all outcomes of individual variants. We call the corresponding transformations outcome-preserving. Subsequently, existing single-program analysis tools (verification oracles) that can handle non-determinism (run-time variability) can be used to analyze the generated single program. Finally, the obtained results are interpreted back on the individual variants. The overview of this approach is given in Figure 1. Instead of using specialized variability-aware tools to analyze program families (which would be tedious and labor intensive), our transformation-based approach allows us to use the standard off-the-shelf single-program analysis tools to achieve the same goal. Nevertheless, the limitation of our approach is that we may not obtain the most precise conclusive results for all individual variants. Of course, this depends on the particular analysis and tool that we use.

To demonstrate correctness of our transformation-based approach, we define the transformations formally using IMP, a small imperative language. To model compile-

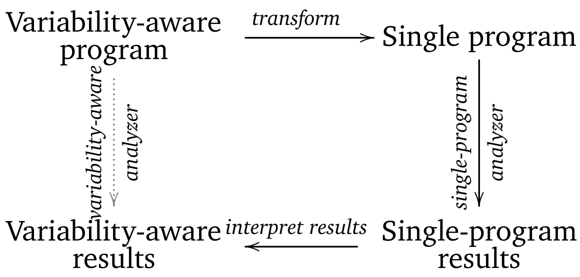

Figure 1 The overview of our transformation-based approach for verification of program families. The single-program analyzer can be any verification oracle for single programs, such as: symbolic executor, type checker, static analyzer, model checker.

time variability, we extend IMP with an "#ifdef" construct for encoding multiple variants, which we call $\overline{\text{IMP}}$ language. To encode run-time variability, we extend IMP with an "or" construct for encoding non-determinism, which we call IMPor language. We define transformations that translate any given $\overline{\text{IMP}}$ program into a corresponding IMPor program. Furthermore, for each transformation we show the relation between the semantics of the input and output programs.

Finally, we report on our experience with implementing and applying our transformations for a full-fledged language, C. The tool, called C Reconfigurator, uses variability-aware parser SuperC [20] for parsing C code with preprocessor annotations, then applies our variability rewrites thus producing a single C program as output. We evaluate our approach on real-world variability intensive C programs with real bugs. We show how some known off-the-shelf single-program analysis tools can be used for efficient and effective verification of such programs.

In summary, this work makes the following contributions:

- A stand-alone variability-related transformation, which transforms a program family into a single program by replacing compile-time variability with non-determinism.
- Correctness of the proposed transformation, which shows that the set of outcomes of the transformed program is equal to the union of sets of outcomes of variants from the input family.
- A prototype tool, C Reconfigurator, which implements the above variability-related transformation for the C language.
- An evaluation of the effectiveness of our transformation-based approach for finding real variability bugs in large variability intensive C software systems.

2 Motivating Example

We begin by showing how our variability transformations work on C program families. Consider a preprocessor-based family of C programs shown in Figure 2 (left column), which uses two (Boolean) features A and B. Our two features give rise to a family of four variants defined by the set of configurations $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$.

```
int foo() {
   int x:= 1;
   #if (A) x:= x+1 #endif;
   #if (B) x:= x-1 #endif;
   return 2/x;
}
```

```
int A := rand()%2;
int B := rand()%2;
int foo() {
   int x := 1;
   if (A) x:= x+1;
   if (B) x:= x-1;
   return 2/x; }
```

Figure 2 Before (left column) and after (right column) our transformations

For each configuration a different variant (single program) can be generated by appropriately resolving #if statements. For example, the variant for $A \land B$ will have both features A and B enabled (set to true), thus yielding the following body of $foo()$: int x := 1; x := x+1; x := x-1; return $2/x$. The variant for $\neg A \land \neg B$ is: int x := 1; return $2/x$. In such program families, errors (also known as *variability bugs* [1]) can occur in some variants (configurations) but not in others. In our example program family in Figure 2, the variant $\neg A \land B$ will crash at the return statement when we attempt to divide by zero. On the other hand, the other variants do not contain the division-by-zero error since the value of x at the return statement is: 1 for variants $A \land B$ and $\neg A \land \neg B$, and 2 for $A \land \neg B$.

In Figure 2, we show a single program (right column) obtained by applying our variability-related transformation on the family shown in the left column. All features are first declared as ordinary global variables and non-deterministically initialized to 0 or 1, then all #if statements are transformed into ordinary if-s with the same conditions. Thus, the division-by-zero error is present in this single program and corresponds to a trace when A is initialized to 0 and B to 1. The set of outcomes of the transformed program (Figure 2, right column) is equal to the union of outcomes of all individual variants from the family (Figure 2, left column). Therefore, the division-by-zero error is present in the transformed program.

In general, the transformed program that we obtain from the original program family can be analyzed by various single-program verification tools, in order to find variability errors or to confirm the absence of errors in the given program family.

3 A Formal Model for Transformations

We now introduce the IMP language that we use to demonstrate our transformations and their proofs of correctness. We describe two extensions of IMP: IMPor used to represent run-time variability (non-determinism), and $\overline{\text{IMP}}$ used to represent compile-time variability.

3.1 IMP

We use a simple imperative language, called IMP [32, 34], which represents a regular general-purpose programming language, aimed at the development of single programs. IMP is a well established minimal language, which is used only for presentational purposes here.

Syntax. IMP is an imperative language with two syntactic categories: expressions and statements. Expressions include integer constants, variables, and binary operations. Statements include a "do-nothing" statement skip, assignments, statement sequences, conditional statements, while loops, and local variable declarations. Its abstract syntax is summarized using the following grammar:

$$\begin{array}{lll} e & ::= & n \mid \mathsf{x} \mid e_0 \oplus e_1 \\ s & ::= & \mathsf{skip} \mid \mathsf{x} := e \mid s_0 \; ; \; s_1 \mid \mathsf{if}\ e\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid \mathsf{var}\ \mathsf{x}{:=}e\ \mathsf{in}\ s \end{array}$$

In the above, n stands for an integer constant, x stands for a variable name, and $\oplus$ stands for any binary arithmetic operator. We denote by *Stm* and *Exp* the set of all statements, s, and expressions, e, generated by the above grammar.

Semantics. A state of a program is a *store* mapping variables to values (integer numbers), $Val = \mathbb{Z}$. We write $Store = Var \rightarrow Val$ to denote the set of all possible stores. Expressions are computed in a given store, denoted by σ. A function $\mathcal{E} : Exp \times Store \rightarrow Val$ defined below by induction on e, maps an expression and a store to a single value, thereby formalizing evaluation of expressions.

$$\mathcal{E}(n,\sigma) = n, \qquad \mathcal{E}(\mathsf{x},\sigma) = \sigma(\mathsf{x}), \qquad \mathcal{E}(e_0 \oplus e_1,\sigma) = \mathcal{E}(e_0,\sigma) \oplus \mathcal{E}(e_1,\sigma)$$

Figure 3 presents the inference rules for a small-step operational semantics for IMP [32, 34]. The notation $\sigma[\mathsf{x} \mapsto n]$ denotes the state that maps x into n and all other variables y into $\sigma(\mathsf{y})$. Following the convention popularized by C, we model Boolean values as integers, with zero interpreted as false and everything else as true (see rules If2 and Wh2, respectively, If1 and Wh1). Note that for variable declarations (see rules Var1 or Var2) we need to restore the declared variable, x, to its earlier global value assigned to x before the declaration, when the scope of declaration has completed. That is why the statement s' in intermediate configurations (rule Var1) is prefixed with variable declarations whose initializations store the local values of x. We can use the inference rules in Figure 3 to define the transition relation: $\langle s,\sigma\rangle \rightarrow \gamma$, where γ is either of the form $\langle s',\sigma'\rangle$ or of the form σ'. If γ is of the form $\langle s',\sigma'\rangle$ then the execution of s is not completed and the complex statement s is rewritten into simpler one s', possibly updating the store σ into σ' (for instance, Seq1 or Seq2). If γ is of the form σ' then the execution of s from σ has terminated and the final state is σ' (for instance, Skip or Wh2).

A *derivation sequence* of s starting in store σ can be either a finite sequence $\langle s,\sigma\rangle \rightarrow \langle s_1,\sigma_1\rangle \rightarrow \ldots \rightarrow \sigma'$ (means: s is run in σ and terminates successfully transforming σ to σ' in the process), or an infinite sequence $\langle s,\sigma\rangle \rightarrow \langle s_1,\sigma_1\rangle \rightarrow \ldots$ (means: s diverges

$$\text{Skip}\,\frac{}{\langle \mathsf{skip}, \sigma\rangle \to \sigma} \qquad \text{Asgn}\,\frac{n = \mathscr{E}(e,\sigma)}{\langle \mathsf{x} := e, \sigma\rangle \to \sigma[\mathsf{x} \mapsto n]} \qquad \text{Sq1}\,\frac{\langle s_0,\sigma\rangle \to \langle s_0', \sigma'\rangle}{\langle s_0 ; s_1,\sigma\rangle \to \langle s_0' ; s_1,\sigma'\rangle}$$

$$\text{Sq2}\,\frac{\langle s_0,\sigma\rangle \to \sigma'}{\langle s_0 ; s_1,\sigma\rangle \to \langle s_1,\sigma'\rangle} \qquad \text{If1}\,\frac{\mathscr{E}(e,\sigma) \neq 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1,\sigma\rangle \to \langle s_0,\sigma\rangle}$$

$$\text{If2}\,\frac{\mathscr{E}(e,\sigma) = 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1,\sigma\rangle \to \langle s_1,\sigma\rangle} \qquad \text{Wh1}\,\frac{\mathscr{E}(e,\sigma) \neq 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle \to \langle s ; \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle}$$

$$\text{Wh2}\,\frac{\mathscr{E}(e,\sigma) = 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle \to \sigma} \qquad \text{Var1}\,\frac{n = \mathscr{E}(e,\sigma) \quad \langle s,\sigma[\mathsf{x} \mapsto n]\rangle \to \langle s',\sigma'\rangle}{\langle \mathsf{var\ x}{:=}e\ \mathsf{in}\ s,\sigma\rangle {\to} \langle \mathsf{var\ x}{:=}\sigma'(\mathsf{x})\ \mathsf{in}\ s',\sigma'[\mathsf{x}{\mapsto}\sigma(\mathsf{x})]\rangle}$$

$$\text{Var2}\,\frac{n = \mathscr{E}(e,\sigma) \quad \langle s,\sigma[\mathsf{x} \mapsto n]\rangle \to \sigma'}{\langle \mathsf{var\ x}{:=}e\ \mathsf{in}\ s,\sigma\rangle \to \sigma'[\mathsf{x} \mapsto \sigma(\mathsf{x})]}$$

Figure 3 Small-step operational semantics for IMP

when run in σ). We write $[\![s]\!]\sigma$ for the final store σ' that can be derived from $\langle s,\sigma\rangle$ (if the derivation is finite), i.e. $\langle s,\sigma\rangle \to^* \sigma'$, otherwise if the derivation is infinite $[\![s]\!]\sigma$ is undefined (empty). In general, we define:

$$[\![s]\!] = \bigcup_{\sigma \in Store^{\mathsf{Init}}} [\![s]\!]\sigma$$

where $Store^{\mathsf{Init}}$ denotes the set of initial input stores on which s is executed.

3.2 IMPor

Syntax The language IMPor is obtained by extending IMP with a non-deterministic choice operator 'or' which can non-deterministically choose to evaluate either of its arguments.

$$e \quad ::= \quad \dots \mid e_0\ \mathsf{or}\ e_1$$

Semantics. Since we have non-deterministic construct 'or', it is possible for an expression to evaluate to a set of different values in a given store. Therefore, now we have $\mathscr{E} : Exp \times Store \to \mathscr{P}(Val)$ defined as follows:

$$\mathscr{E}(n,\sigma) = \{n\}, \qquad \mathscr{E}(\mathsf{x},\sigma) = \{\sigma(\mathsf{x})\}, \qquad \mathscr{E}(e_0\ \mathsf{or}\ e_1,\sigma) = \mathscr{E}(e_0,\sigma) \cup \mathscr{E}(e_1,\sigma)$$
$$\mathscr{E}(e_0 \oplus e_1,\sigma) = \{v_0 \oplus v_1 \mid v_0 \in \mathscr{E}(e_0,\sigma), v_1 \in \mathscr{E}(e_1,\sigma)\}$$

The small-step operational semantics rules for IMPor are those of IMP given in Figure 3, but now they take into account the non-determinism of $\mathscr{E}(e,\sigma)$. For example, we have:

$$\text{Wh1}\,\frac{n \in \mathscr{E}(e,\sigma) \quad n \neq 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle \to \langle s ; \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle} \qquad \text{Wh2}\,\frac{0 \in \mathscr{E}(e,\sigma)}{\langle \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle \to \sigma}$$

For IMPor, we write $[\![s]\!]\sigma$ for the *set* of final stores σ' that can be derived from $\langle s,\sigma\rangle$, i.e. $\langle s,\sigma\rangle \to^* \sigma'$. Note that since IMPor is a non-deterministic language $[\![s]\!]\sigma$ may contain more than one final store. Finally, $[\![s]\!] = \bigcup_{\sigma \in Store^{\mathsf{Init}}} [\![s]\!]\sigma$.

3.3 $\overline{\mathbf{IMP}}$

A finite set of Boolean variables $\mathbb{F} = \{A_1, \dots, A_n\}$ describes the set of available *features* in the program family. Each feature may be *enabled* or *disabled* in a particular variant.

A *configuration* k is a truth assignment or a valuation which gives a truth value to each feature, i.e. k is a mapping from $\mathbb{F}$ to $\{\text{true}, \text{false}\}$. If a feature $A \in \mathbb{F}$ is enabled for the configuration k then $k(A) = \text{true}$, otherwise $k(A) = \text{false}$. Any configuration k can also be encoded as a conjunction of literals: $k(A_1) \cdot A_1 \wedge \cdots \wedge k(A_n) \cdot A_n$, where $\text{true} \cdot A = A$ and $\text{false} \cdot A = \neg A$. We write $\mathbb{K}$ for the set of all *valid* configurations defined over $\mathbb{F}$ for a family. The set of valid configurations is typically described by a feature model [23], but in this work we disregard syntactic representations of the set $\mathbb{K}$. Note that $|\mathbb{K}| \leq 2^{|\mathbb{F}|}$, since, in general, not every combination of features yields a *valid* configuration. We define *feature expressions*, denoted *FeatExp*, as the set of well-formed propositional logic formulas over $\mathbb{F}$ generated using the grammar: $\phi ::= \text{true} \,|\, A \in \mathbb{F} \,|\, \neg\phi \,|\, \phi_1 \wedge \phi_2 \,|\, \phi_1 \vee \phi_2$.

Syntax. The programming language $\overline{\text{IMP}}$ is our two-stage extension of IMP (thus, $\overline{\text{IMP}}$ does not contain the 'or' construct). Its abstract syntax includes the same expression and statement productions as IMP, but we add the new compile-time conditional statements for encoding multiple variants of a program. The new statements "#if (ϕ) s #endif" and "#if (ϕ) var x:=n in #endif s" contain a feature expression $\phi \in$ *FeatExp* as a presence condition, such that only if ϕ is satisfied by a configuration $k \in \mathbb{K}$ then the code between #if and #endif will be included in the variant for k.

s ::= ... | #if (ϕ) s #endif | #if (ϕ) var x:=n in #endif s

Note that only statements and local variable declarations can be compile-time conditionally defined in $\overline{\text{IMP}}$. However, in general "#if" constructs defined on arbitrary language elements could be translated into constructs that respect the appropriate syntactic structure of the language by code duplication [19]. Also note that the C preprocessor uses the following keywords: #if, #ifdef, and #ifndef to start a conditional construct; #elif and #else to create additional branches; and #endif to end a construct. Any of such preprocessor conditional constructs can be desugared and represented only by #if construct we use in this work, e.g. #ifdef (ϕ) s_0 #else s_1 #endif is translated into #if (ϕ) s_0 #endif ; #if ($\neg\phi$) s_1 #endif.

Semantics. The semantics of $\overline{\text{IMP}}$ has two stages: first, given a configuration $k \in \mathbb{K}$ compute an IMP single program without #if-s; second, evaluate the obtained variant using the standard IMP semantics. The first stage is a simple *preprocessor* specified by the projection function π_k mapping an $\overline{\text{IMP}}$ program family into an IMP single program corresponding to the configuration $k \in \mathbb{K}$. The projection π_k copies all basic statements of $\overline{\text{IMP}}$ that are also in IMP, and recursively pre-processes all sub-statements of compound statements. For example, $\pi_k(\text{skip}) = \text{skip}$ and $\pi_k(s_0;s_1) = \pi_k(s_0);\pi_k(s_1)$. The interesting case is "#if (ϕ) s #endif" (resp., #if (ϕ) var x:=n in #endif s) statement, where the statement s (resp., the local variable declaration var x:=n in) is included in the resulting variant iff $k \models \phi$;[1] otherwise it is removed. We have:

$$\pi_k(\texttt{\#if }(\phi)\ s\ \texttt{\#endif}) = \begin{cases} \pi_k(s) & \text{if } k \models \phi \\ \texttt{skip} & \text{if } k \not\models \phi \end{cases}$$

$$\pi_k(\texttt{\#if }(\phi)\ \texttt{var x:=}n\ \texttt{in}\ \ \texttt{\#endif}\ s) = \begin{cases} \pi_k(\texttt{var x:=}n\ \texttt{in}\ s) & \text{if } k \models \phi \\ \pi_k(s) & \text{if } k \not\models \phi \end{cases}$$

Note that since any configuration $k \in \mathbb{K}$ has only one satisfying truth assignment (values of all features are fixed in k), either $k \models \phi$ or $k \not\models \phi$ for any $\phi \in$ *FeatExp*.

4 Variability-related Transformations

Our aim is to transform an input $\overline{\text{IMP}}$ program family $\bar{s}$ with sets of features $\mathbb{F}$ and configurations $\mathbb{K}$ into an output IMPor program $\overline{s'}$.

In a pre-transformation phase, we first convert each feature $A \in \mathbb{F}$ into the variable A, which is non-deterministically initialized to 0 or 1 (meaning to false or true). Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be the set of available features in the family $\bar{s}$, then we have the following initialization fragment in the resulting pre-transformed program pre-t($\bar{s}$):

$$\texttt{pre-t}(\bar{s}) = \texttt{var } A_1 := 0 \texttt{ or } 1 \texttt{ in } \ldots \texttt{var } A_n := 0 \texttt{ or } 1 \texttt{ in } \bar{s}$$

Note that in the initialization we consider all possible combination of values for features (in total $2^{|\mathbb{F}|}$). We will take into account the specific set of configurations $\mathbb{K}$ ($|\mathbb{K}| \leq 2^{|\mathbb{F}|}$) later on, in the transformation phase.

In the following, rewrite rules have the form:

$$\psi \ \vdash \ s \ \rightsquigarrow \ s'$$

meaning that: if the current program family being transformed matches any abstract syntax tree (AST) node of the shape s nested under #if-s with the resulting presence condition that implies $\psi \in$ *FeatExp* (i.e. in context ψ) then *replace s by s'*. Formally, if we apply the rule $\psi \ \vdash \ s \rightsquigarrow s'$ to a family:

$$\ldots\texttt{\#if }(\phi_1)\ \ldots\texttt{\#if }(\phi_n)\ \ldots; s; \ldots\ \texttt{\#endif}\ldots\ \texttt{\#endif}\ldots$$

where $\phi_1 \land \ldots \land \phi_n \implies \psi$, then we obtain the transformed program:

$$\ldots\texttt{\#if }(\phi_1)\ \ldots\texttt{\#if }(\phi_n)\ \ldots; s'; \ldots\ \texttt{\#endif}\ldots\ \texttt{\#endif}\ldots$$

We write $Rewrite(\bar{s}, \psi \ \vdash \ s \rightsquigarrow s')$ for the final transformed program $\overline{s'}$ obtained by repeatedly applying the rule $\psi \ \vdash \ s \rightsquigarrow s'$ on $\bar{s}$ and its transformed versions until we reach a point where this rule can not be applied (a fixed point of the rule). Note

[1] Here $\models$ denotes the standard satisfaction relation of propositional logic.

that rules of the form: true $\vdash s \rightsquigarrow s'$, are the most general and can be applied to any statement s no matter whether s is a top-level statement not nested within some #if or s is nested somewhere deep within #if-s. This is due to the fact that any "s" can be written as: "#if (true) s #endif" in the earlier case when s is a top-level statement, and $\phi \implies$ true for any $\phi \in FeatExp$ in the latter case when s is nested within #if-s with presence condition ϕ.

We start with three rules for eliminating configurable variable declarations. They involve duplicating code and variable renaming. The most straightforward way to handle renaming of variables in different contexts is by adding an *environment* δ as a parameter to the statements being transformed. We define an environment $\delta : \mathit{Var} \times \mathit{FeatExp} \to \mathit{Var}$ as a function mapping a given pair of a variable and a feature expression to a variable name. We write $\delta^{\mathsf{fe}}(\mathsf{x}) \subseteq \mathit{FeatExp}$ for the set of all feature expressions ϕ such that $\delta(\mathsf{x},\phi)$ is defined, i.e. $\delta^{\mathsf{fe}}(\mathsf{x}) = \{\phi \in \mathit{FeatExp} \mid (\mathsf{x},\phi) \in dom(\delta)\}$. We write (s,δ) to denote the result of simultaneously substituting $\delta(\mathsf{x},\phi)$ for each occurrence of any variable x in s in the context (presence condition) that implies ϕ.

Conditional variable declaration. This rule transforms a local variable conditionally declared within a given context $\psi \in \mathit{FeatExp}$:

$$\psi \vdash (\mathsf{\#if}\ (\phi)\ \mathsf{var\ x}{:=}n\ \mathsf{in\ \#endif}\, s,\delta) \rightsquigarrow \mathsf{var\ x}_{new}{:=}n\ \mathsf{in}\ (s,\delta[(\mathsf{x},\phi)\mapsto \mathsf{x}_{new}]) \tag{1}$$

where x_{new} is a fresh variable name that does not occur as a free variable in s and $range(\delta)$.

Conditional variable use. The second rule handles the case when a local variable is used within a context $\psi \in FeatExp$. There are three cases to consider here.

$$\psi \vdash (\mathsf{y}{:=}e[\mathsf{x}],\delta) \rightsquigarrow (\mathsf{y}{:=}e[\delta(\mathsf{x},\phi)],\delta) \tag{2.1}$$

if there exists an unique $\phi \in \delta^{\mathsf{fe}}(\mathsf{x})$, such that $\psi \models \phi$. Here $e[\mathsf{x}]$ means that the variable x occurs free in the expression e. The second case is when there are several $\phi_1,\dots\phi_n \in \delta^{\mathsf{fe}}(\mathsf{x})$, such that $\mathsf{sat}(\phi_1\wedge\psi),\dots,\mathsf{sat}(\phi_n\wedge\psi)$:

$$\psi \vdash (\mathsf{y}{:=}e[\mathsf{x}],\delta) \rightsquigarrow (\mathsf{\#if}\ (\phi_1)\ \mathsf{y}{:=}e[\delta(\mathsf{x},\phi_1)]\ \mathsf{\#endif;}\dots\mathsf{\#if}\ (\phi_n)\ \mathsf{y}{:=}e[\delta(\mathsf{x},\phi_n)]\ \mathsf{\#endif},\delta) \tag{2.2}$$

Otherwise, meaning that for all $\phi \in \delta^{\mathsf{fe}}(\mathsf{x})$ it follows that $\mathsf{unsat}(\phi\wedge\psi)$, we have:

$$\psi \vdash (\mathsf{y}{:=}e[\mathsf{x}],\delta) \rightsquigarrow (\mathsf{y}{:=}e[\mathsf{x}],\delta) \tag{2.3}$$

Conditional variable define. The third rule applies when a local variable is assigned to within a context $\psi \in \mathit{FeatExp}$. There are three cases to consider here as well.

$$\psi \vdash (\mathsf{x}{:=}e,\delta) \rightsquigarrow (\delta(\mathsf{x},\phi){:=}e),\delta \tag{3.1}$$

when there exists an unique $\phi \in \delta^{\mathsf{fe}}(\mathsf{x})$, such that $\psi \models \phi$.

$$\psi \vdash (\mathsf{x}{:=}e,\delta) \rightsquigarrow (\mathsf{\#if}\ (\phi_1)\ \delta(\mathsf{x},\phi_1){:=}e\ \mathsf{\#endif;}\dots\mathsf{\#if}\ (\phi_n)\ \delta(\mathsf{x},\phi_n){:=}e\ \mathsf{\#endif}),\delta \tag{3.2}$$

when there are $\phi_1,\dots\phi_n \in \delta^{\mathsf{fe}}(\mathsf{x})$, such that $\mathsf{sat}(\phi_1 \wedge \psi),\dots,\mathsf{sat}(\phi_n \wedge \psi)$. Otherwise,

$$\psi \vdash (\mathsf{x}{:=}e,\delta) \rightsquigarrow (\mathsf{x}{:=}e,\delta) \tag{3.3}$$

After applying the above three rules, all local variable declarations that are conditionally defined ($\texttt{\#if}\ (\phi)\ \mathsf{var}\ \mathsf{x}{:=}n\ \mathsf{in}\ \texttt{\#endif}s$) are resolved. The transformed program contains only #if-s where statements are conditionally defined.

Conditional statement elimination. The set of valid configurations $\mathbb{K}$ can be equated to a propositional formula [4], say $\kappa \in \mathit{FeatExp}$, such that $\kappa = \vee_{k\in\mathbb{K}} k$. The last rule simply replaces #if-s with ordinary if-s whose guards are strengthen with the feature model κ, thus taking into account only valid configurations $\mathbb{K}$ of a family.

$$\psi \vdash \texttt{\#if}\ (\phi)\ s\ \texttt{\#endif} \rightsquigarrow \mathsf{if}\ (\phi \wedge \kappa)\ \mathsf{then}\ s\ \mathsf{else\ skip} \tag{4}$$

Note that we omit to write the environment δ in rules that do not use it explicitly (e.g. rules (4), (5)). Let $\delta_0 = [\,]$ be the empty environment. Let $\mathit{Rewrite}^{\mathsf{preserve}}(\mathsf{pre\text{-}t}(\bar{s}),\delta_0)$ be the final transformed program $\overline{s'}$ obtained from the pre-transformed program $\mathsf{pre\text{-}t}(\bar{s})$ by applying the rules (1)–(3), and then the rule (4). The following result shows that the set of final answers from terminating computations of $\overline{s'}$ coincides with the union of final answers from terminating computations of all variants from $\bar{s}$.

Theorem 1. *Let $\overline{s'} = \mathit{Rewrite}^{\mathsf{preserve}}(\mathsf{pre\text{-}t}(\bar{s}),\delta_0)$. We have: $[\![\overline{s'}]\!] = \bigcup_{k\in\mathbb{K}} [\![\pi_k(\bar{s})]\!]$.*

Proof. First, we show that $\mathit{Rewrite}^{\mathsf{preserve}}(\mathsf{pre\text{-}t}(\bar{s}),\delta_0)$ terminates. This is due to the fact the number of if-s in pre-t($\bar{s}$ is finite, and by iteratively applying rules (1)–(3) we eliminate all $\texttt{\#if}\ (\phi)\ \mathsf{var}\ \mathsf{x}{:=}n\ \mathsf{in}\ \texttt{\#endif}s$; whereas by applying rule (4) afterwards we eliminate all $\texttt{\#if}\ (\phi)\ s\ \texttt{\#endif}$. Subsequently, for each rule (1)–(3) and (4), the above result can be proved by structural induction. □

We now present an optimization rule, which is applied before the rules (1)–(4) for eliminating if-s. The correctness of our transformation does not depend on it, but we can use it for achieving faster convergence and smaller transformed programs. In our implementation, we use many such optimization rules.

Guard inlining. This rule collapses two sequentially composed #if-s with mutually exclusive presence conditions ϕ_0 and ϕ_1 (i.e. $\phi_0 \wedge \phi_1 \equiv \mathsf{false}$) that conditionally enable the same statement s into one #if that conditionally enables s:

$$\psi \vdash \texttt{\#if}\ (\phi_0)\ s\ \texttt{\#endif};\ \texttt{\#if}\ (\phi_1)\ s\ \texttt{\#endif} \rightsquigarrow \texttt{\#if}\ (\phi_0 \vee \phi_1)\ s\ \texttt{\#endif} \tag{5}$$

Example 2. *We present the transformation rules on a program family with $\mathbb{F} = \{A,B\}$ and $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$.*

$$\begin{aligned}
&\big(\textit{\#if}\ (A)\ \textit{var x:=2 in}\ \textit{\#endif\#if}\ (\neg A)\ \textit{var x:=5 in}\ \textit{\#endif\#if}\ (B)\ \textit{y:=x \#endif}, \delta_0\big)\\
&\overset{(1)}{\rightsquigarrow}\ \textit{var}\ \textit{x}_1\textit{:=2 in}\ \big(\textit{\#if}\ (\neg A)\ \textit{var x:=5 in}\ \textit{\#endif\#if}\ (B)\ \textit{y:=x \#endif}, [(\textit{x},A) \mapsto \textit{x}_1]\big)\\
&\overset{(1)}{\rightsquigarrow}\ \textit{var}\ \textit{x}_1\textit{:=2 in}\ \textit{var}\ \textit{x}_2\textit{:=5 in}\ \big(\textit{\#if}\ (B)\ \textit{y:=x \#endif}, [(\textit{x},A) \mapsto \textit{x}_1, (\textit{x},\neg A) \mapsto \textit{x}_2]\big)\\
&\overset{(2.2)}{\rightsquigarrow}\ \textit{var}\ \textit{x}_1\textit{:=2 in}\ \textit{var}\ \textit{x}_2\textit{:=5 in}\ \textit{\#if}\ (B)\ \textit{\#if}\ (A)\ \textit{y:=x}_1\textit{; \#endif\#if}\ (\neg A)\ \textit{y:=x}_2\ \textit{\#endif \#endif}\\
&\overset{(4)}{\rightsquigarrow}\ \textit{var}\ \textit{x}_1\textit{:=2 in}\ \textit{var}\ \textit{x}_2\textit{:=5 in}\ \textit{if}(B)\textit{then}\,\textit{if}\ (A)\ \textit{then y:=x}_1\ \textit{else skip};\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{if}\ (\neg A)\ \textit{then y:=x}_2\ \textit{else skip};\ \textit{else}\,\textit{skip}
\end{aligned}$$

5 Implementation

We have developed a tool, called C Reconfigurator, which implements variability-related transformations for the C language. All transformations are implemented using Xtend [2] . The C Reconfigurator tool is available from: https://github.com/models-team/c-reconfigurator. It calls variability-aware parser SuperC [20] to parse code with preprocessor annotations, which uses Binary Decision Diagrams (BDD's) for encoding feature expressions and for decisions during the parsing process. SuperC returns an AST with variability, in which variability is reflected with choice nodes over feature expressions. In particular, a choice node is a node with two children, such that the left child of the choice node is included in the result of those configurations for which the given feature expression is satisfied; otherwise the right child of the choice node is included in the parsing result when the feature expression is not satisfied. We apply our variability-related transformation rules as described in Section 4 on AST's with variability obtaining an ordinary AST, which is subsequently translated into a single C program. Since IMP is a subset of C, all rewritings described in Section 4 transfer directly to C. We now discuss how a selection of other interesting C constructs, which are not present in IMP, are handled by our tool.

Variables declared with optional types are very common in C. For example, we have x-bit integers on x-bit machines. We handle them in a similar way as configurable variable declarations in rules (1)–(3). First, we rename and duplicate the variable declaration, then at each point where the variable is used we transform the code such that the used variable refers to the correct configuration name. For example,

#if (A) int #else float #endif x=0;
x = x+1;

will be transformed into:

int $x_1 = 0$; float $x_2 = 0$;
#if (A) $x_1 = x_1$+1; #else $x_2 = x_2$+1; #endif

Note that if optional local variables are initialized by non-constant expressions, then we split their transformation into two parts: declaration which is performed by renaming and duplication, followed by initialization where all optional variables refer to the correct configuration.

Optional (configurable defined) functions are important since all statements in C are inside some function. If conditionally defined code occurs in the function body, then it will be transformed using the corresponding rules. For example,

int f (int x) {return #if (A) x++ #else 0 #endif; }

will be transformed into:

int f (int x) {return A ? x++ : 0; }

[2] http://www.eclipse.org/xtend/.

If the function signature is configurable, then we use renaming plus duplication as in rules (1)–(3) for handling configurable variable declarations. For example, the code:

int f (#if (A) int #else float #endif x) {...}
...f(5)...

will be transformed into:

int f_1(int x) {...}
int f_2(float x) {...}
...#if (A) f_1(5) #else f_2(5) #endif...

Arrays with optional size are also possible in real-world C programs. They usually emerge via constant macros with conditional definitions. For example, the code

int a[#if (A) 10 #else 15 #endif];
a[5] = 0;

will be transformed into:

int a_1[10]; int a_2[15];
#if (A) a_1[5] = 0; #else a_2[5] = 0; #endif

All other variability patterns that we met in our examples, such as configurable fields in struct-s and pointers, are also handled similarly: first by using renaming and duplication, then by modifying all references to the given pattern such that the use always refers to the correct definition. Consider the following code with pointers:

int a = 10; int$*$ p = &a; #if (A) p = $null$; #endif ($*$p)++

will be transformed into:

int a = 10; int$*$ p = &a; if (A) p = $null$; ($*$p)++

Hence, we obtain a variability bug whenever the feature A is enabled.

Remark. We can see that most of the variability patterns are handled using renaming plus duplication. In the worst case, this may cause exponential growth of the transformed program in the number of used features. However, in practice this does not happen often (see Table 3 for some data from real files). Namely, variability patterns usually depend on a few features, so only a few new definitions are used. Also we apply several optimization rules, which eliminate all definitions that do not correspond to a valid configuration. Finally, the evaluation results in Section 6 show that the analysis time for such transformed programs is comparable to single programs. This is due to the fact that transformed programs are not increased significantly and the analysis tools we use (Frama-C, Clang, LLBMC) are very optimized and mature.

6 Evaluation

We evaluate our reconfiguration technique based on variability transformations and single-program verification oracles on several real-world C case studies. The evaluation aims to show that we can use state-of-the-art single-program verification tools to verify realistic C program families using variability-related transformations. To do so, we ask the following research questions:

- How precise is our technique? **(RQ1)**
- How efficient is the verification oracle to identify variability bugs after transforming the code using our technique? **(RQ2)**

In particular, we want to reproduce the variability bugs reported in [1, 28] using various verification oracles applied on transformed programs, which are obtained using our tool. We use Frama-C [27], Clang [9] and LLBMC [30] as our verification oracles. Frama-C is a framework for modular static (dataflow) analysis of C programs. The Clang project includes the Clang compiler front-end and the Clang static analyzer for several programming languages, including C. LLBMC (the low-level bounded model checker) is a software model checking tool for finding bugs in C programs.

6.1 Subject Files and Experimental Setup

All transformations are applied using the C Reconfigurator tool as described in Section 5. We investigate precision and performance in finding real variability bugs extracted from three benchmarks: Linux, BusyBox and Libssh. In particular, we use simplified bugs from the VDBb [3] database that are found in the Linux kernel files [1] and in BusyBox. Abal et al. [1] created a simplified version of a program for each bug they found by capturing the same essential behavior (and the same problem) as in the original bug. Simplified bugs are independent of the kernel code and the corresponding programs were derived systematically from the error trace. In addition, we use real variability bugs from Libssh provided by Medeiros et al. [28].

Table 1 presents the characteristics of the subject files we analyzed in our empirical study. We list: the file id, bug type, number of features ($|\mathbb{F}|$), number of valid configurations ($|\mathbb{K}|$), lines of code, the size in KB of the files before (with `#ifdef`-s) and after (without `#ifdef`-s) our transformations, and commit hash (clickable) for each project. This collection consists of a diverse set of bug types such as null pointer dereferences, buffer overflow, and uninitialized variable. In total, we have 11 distinct kinds of bugs. The number of features per file varies from one to seven. In addition, the number of lines of code ranges from 12 to 165 for the simplified files (from VBDb), and from 1404 to 2959 for real files (from Libssh). After the transformation, the biggest increase in size of almost 8 times can be observed for file id 7. This is due to the fact that this file has seven different features and several variability patterns that depend on them. In most of the other cases the size increase is not very big.

[3] http://VBDb.itu.dk.

File ID	Bug type	$\lvert\mathbb{F}\rvert$	$\lvert\mathbb{K}\rvert$	LOC	Size KB		Hash
					before	after	
VBDb Linux files							
1	null pointer deref.	5	24	165	2.9	4.3	76baeeb
2	null pointer deref.	3	6	112	1.9	2.5	f7ab9b4
3	null pointer deref.	4	8	55	0.9	1.0	ee3f34e
4	null pointer deref.	3	6	34	0.5	0.6	6252547
5	buffer overflow	1	2	58	1.0	1.2	8c82962
6	buffer overflow	1	2	33	0.6	0.7	60e233a
7	read out of bounds	7	63	69	1.1	8.4	0f8f809
8	uninitialized var.	2	4	54	0.8	1.0	7acf6cd
9	uninitialized var.	1	2	54	1.0	1.1	bc8cec0
10	uninitialized var.	1	2	53	0.8	1.0	30e0532
11	uninitialized var.	2	4	38	0.9	1.2	1c17e4d
12	uninitialized var.	2	4	26	0.3	0.5	e39363a
13	undefined symbol	4	14	25	0.4	0.6	7c6048b
14	undefined symbol	2	4	20	0.3	0.5	2f02c15
15	undefined symbol	2	4	20	0.3	0.5	6515e48
16	undefined symbol	2	4	19	0.3	0.5	242f1a3
17	undeclared identifier	3	8	37	0.6	1.0	6651791
18	undeclared identifier	2	4	20	0.3	0.4	f48ec1d
19	wrong # of args	1	2	12	0.2	0.4	e67bc51
20	multiple funct. defs	2	4	21	0.3	0.8	e68bb91
21	dead code	1	2	19	0.2	0.3	809e660
22	incompatible type	2	4	27	0.4	0.7	d6c7e11
23	assertion violation	2	4	79	1.5	1.8	63878ac
24	assertion violation	2	4	75	1.1	1.2	657e964
25	assertion violation	2	4	41	0.6	0.7	0988c4c
VBDb BusyBox files							
26	null pointer deref.	1	2	28	0.4	0.7	199501f
27	null pointer deref.	2	4	24	0.4	0.6	1b487ea
28	uninitialized var.	2	4	28	0.4	0.7	b273d66
29	undefined symbol	1	2	42	0.8	0.9	cf1f2ac
30	undefined symbol	2	4	27	0.4	0.6	ebee301
31	undeclared identifier	1	2	35	0.5	0.8	5275b1e
32	undeclared identifier	1	2	19	0.3	0.4	b7ebc61
33	incompatible type	3	8	46	0.9	1.5	5cd6461
Real Libssh files							
34	null pointer deref.	6	48	1404	34.8	32.6	0a4ea19
35	null pointer deref.	4	4	1428	44.1	31.9	fadbe80
36	uninitialized var.	3	4	2959	72.4	77.6	2a10019

Table 1 Characteristics of the benchmark files.

ID	Frama-C				
	BUGGY VARIANT		RECONFIGURED		ALL
	y/n	time	y/n	time	time
VBDb Linux files					
1	√	218	√	235	5602
2	√	220	√	225	1394
3	√	215	x	236	1918
4	√	218	√	224	1379
5	√	218	√	227	488
6	√	213	√	227	463
7	√	218	√	225	14381
8	√	241	√	250	918
9	√	224	√	230	462
10	√	216	inc	224	460
11	√	234	√	224	917
12	√	216	inc	227	914
13	√	239	√	248	3194
14	√	237	√	244	905
15	√	224	√	248	906
16	√	213	√	222	910
17	√	216	√	230	3823
18	√	210	√	224	901
19	√	210	√	224	452
20	√	213	x	228	907
21	√	239	x	240	458
VBDb BusyBox files					
26	√	230	√	234	484
27	√	224	√	234	959
28	√	237	inc	237	957
29	√	230	√	236	481
30	√	231	√	228	968
31	√	220	√	228	486
32	√	216	√	224	477

(a) VBDb files using Frama-C.

ID	Clang/LLBMC				
	BUGGY VARIANT		RECONFIGURED		ALL
	yes/no	time	yes/no	time	time
VBDb Linux files					
22	√	21	√	23	91
23	√	4	√	10	10
24	√	3	√	7	11
25	√	3	√	5	8
VBDb BusyBox files					
33	√	27	√	31	222

(b) VBDb files using Clang (files 22 and 33) and LLBMC (files 23, 24, and 25).

ID	Clang/LLBMC				
	BUGGY VARIANT		RECONFIGURED		ALL
	yes/no	time	yes/no	time	time
34	√	1526	√	1702	17029
35	√	1591	√	1804	5917
36	√	112	√	144	448

(c) Libssh files using Clang (file 36) and LLBMC (files 34 and 35).

Table 2 Verification results for the benchmark files. Times in milliseconds (ms).

All experiments were executed on a Kubuntu VM (64bit, 4 CPUs), Intel®Core^{TM} i7-3720QM CPU running at 2.6GHz with 12GB RAM memory. The performance numbers reported constitute the median runtime of fifty independent executions.

6.2 Results

We now present the results of our empirical study and discuss the implications. All experiment materials are available online at https://github.com/models-team/c-reconfigurator-test. Before we proceed, we stress that we only evaluate bugs that are detectable by the verification tools on the erroneous variant code.

Simplified files. Table 2a shows the results of verifying our benchmark files which contain known bugs by using Frama-C. The table has three main columns: buggy variant, reconfigured, and all that depict the tool results on the buggy variant code, on the reconfigured program family code, and on all valid variants from $\mathbb{K}$ analyzed one by one (in a brute force fashion), respectively. Each checkmark (✓) means that the same bug was found in both the buggy variant and reconfigured program by the verification tool. Otherwise, the result is either x—bug not found in the reconfigured program, or *inc*—inconclusive which means that Frama-C was able to detect a bug in the reconfigured program that is different from the bug in the product variant. In the case of brute force approach (all), we consider the analyses times of all valid variants regardless of whether they contain a bug or not.

In terms of precision, our C Reconfigurator tool transforms the family code by preserving the erroneous traces from the buggy variant in most cases. For instance, Frama-C could detect 22 (78%) bugs from the simplified benchmark files (28 in total) after reconfiguring the files using our tool. Besides that, the C Reconfigurator preserves a variety of bug types such as buffer overflow and uninitialized variable. Still, for different types of bugs the success rate depends on the tool which may or may not detect them. For example, our technique is able to transform a file containing a memory leak error, but Frama-C does not have any analysis to identify it.

In three specific cases (cf. file ids 10, 12 and 28), Frama-C did not report the original bug as an error, but it did detect that some variable might be uninitialized in some conditions. This happens because Frama-C performs a *may* value analysis for finding uninitialized variables. A *may* analysis describes information that may possibly be true along one path to the given program point and, thus in our case, computes a superset of all uninitialized variables in all variants. So the reported variable may not match with the one in the buggy variant. We marked these three cases as *inc*—inconclusive in the table. Still the verification oracle reports that there might be an error in the reconfigured code.

In addition, the verification tool could not identify the required bug in the reconfigured file in three cases (cf. file ids 3, 20 and 21). For example, file 21 contains dead code, which is a function (do_sect_fault()) that is never called when feature ARM is enabled (see the code snippet in Fig. 4, left column). The C Reconfigurator transforms the code by changing the #ifdef into ordinary if condition, making the function available for the transformed single program (i.e., the function is not dead any more), as shown in the code snippet in Fig. 4 (right column). The other two cases are similar to this one in the sense that the C Reconfigurator makes feature code explicit to the entire program family.

Generally speaking, if one variant does not use a variable/function, but another does, then the reconfigured code will use the variable/function and the error will be hidden (like in the example above). This happens due to the limitations of variability encoding, especially because we cannot preprocess the reconfigured code to filter out the irrelevant features for a particular variant. In a reconfigured code, all variants are encoded as a single program (see Section 6.4 for more discussion).

We now consider the remaining simplified files. We use Clang and LLBMC to analyze only the other types of bugs (incompatible type and assertion violation) that

```
int do_sect_fault(){          | int do_sect_fault(){
   return 0;                  |  return 0;
}                             | }
int main(){                   |
   #ifndef ARM                | int main(){
      do_sect_fault();        |     if (! ARM)
   #endif                     |        do_sect_fault();
   return 0;                  |    return 0;
}                             | }
```

Figure 4 File 21 - Before (left) and after (right) our transformations

Frama-C cannot handle. We treat Clang/LLBMC as one verification oracle, since we first need to compile and emit llvm code with Clang in order to analyze it using LLBMC. So, we do not make difference in reporting whether the bug was found by Clang during the compilation or afterwards by LLBMC.

Table 2b, similarly to Table 2a, shows the results of verifying both the buggy variant and the reconfigured code using Clang and LLBMC. We also report the analysis time of the brute force approach in the column all. As we can see, all bugs were found by Clang/LLBMC in the reconfigured version. We can thus confirm that our C Reconfigurator tool transforms the family code by preserving the erroneous traces from the buggy variant. We are now ready to answer RQ1 on the precision of our technique. Based on analyzing 33 simplified variability bugs from Linux and BusyBox, we find that:

> Answer RQ1 (precision). The C Reconfigurator enables single-program verification tools such as Frama-C, Clang, and LLBMC to **successfully** detect *most* of the simplified variability bugs on the reconfigured code, obtained from the Linux and BusyBox benchmark files.

We now turn to evidence regarding research question RQ2 (performance). We evaluate performance of the verification tools to identify the given variability bugs. Tables 2a and 2b show time needed for the verification tools to analyze the buggy variant code (buggy variant column) and the reconfigured program family code (reconfigured column). We can see that the analysis times in both cases are similar although reconfigured code is bigger in size. In fact, Frama-C takes less than half a second to analyze each file regardless whether it is a variant or a reconfigured file. For instance, Frama-C analyzes file 1 in 218 and 235 milliseconds on the variant code and on the reconfigured program family code, respectively. Recall that file 1 contains a null pointer dereference and has five features. If we apply the brute force approach (all column), which analyzes all variants individually one by one, to this file using Frama-C it takes 5,602 ms, since the number of configurations is 24. In this way, we obtain significant speed-up to verify the program family using our approach. We also obtain similar results in terms of performance using Clang/LLBMC (see Tables 2b and 2c). In general, the performance of analyzing a reconfigured code is similar to analyzing only one variant, which gives us a speed-up proportional to the number of valid variants of a program family. Overall, we answer the second research question (RQ2) by observing that:

> ANSWER RQ2 (PERFORMANCE). The C RECONFIGURATOR speeds-up the family-based analysis via single-program verification tools, so that we can **efficiently** detect simplified variability bugs on the reconfigured code, obtained from the VBDb benchmark.

Real files. We now consider real files to confirm our previous observations with respect to precision and performance. Table 2c presents the results of analyzing three real files from the Libssh project using CLANG and LLBMC.[4] These files contain two types of bugs: null pointer dereference and uninitialized variable. Each file has at least three distinct features.

We can see that our C RECONFIGURATOR transforms the family code by preserving the erroneous traces from the buggy variant even for complex and large files. In fact, the verification tool (CLANG/LLBMC) found the same bug (from the buggy variant code) on the reconfigured code in all three cases. From this preliminary evidence, we thus confirm that our technique enables single-program verification oracles to successfully detect variability bugs on the reconfigured code, obtained from complex and real files.

Regarding performance, we can still see the similarity in verifying a variant code and a reconfigured one. For example, CLANG/LLBMC took 1,5 sec to analyze file 34 in the single variant version, whereas in the reconfigured version, the tool analyzed it in 1,7 sec. We can also observe a speed-up of the family-based analysis using the C RECONFIGURATOR and single-program verification tools by a factor of the number of valid variants compared to the brute force approach. We conclude that:

> SUMMARY. All single-program verification tools (FRAMA-C, CLANG, LLBMC) detect **successfully** and **efficiently** most of the variability bugs on the reconfigured code as well as on the single variant code.

6.3 Threats to Validity

Internal validity. Verifying semantics preservation in a complex transformation is a very hard problem [22, 2]. We manually verified the correctness of the C RECONFIGURATOR on the simplified VBDb files by comparing the original and the reconfigured files side-by-side, which leaves space for human error. For the larger real files we were not able to determine if the C RECONFIGURATOR preserved semantics for all variants on the entire file due to the complex configuration space, but instead we focused on the functions involved in producing/reproducing the bug. We mitigate this threat by relying on the results of our evaluation which show the effectiveness of conventional single-program analysis tools to identify the same bugs in the reconfigured code version as in the buggy single varaints.

[4] We do not report results from FRAMA-C on the real files because FRAMA-C could not handle them.

External validity. From our preliminary evaluation, we show that our technique transforms the program family code by preserving the erroneous traces from the buggy variant. However, we acknowledge that our transformations were not tested under the entirety of the C language, but only on the subset used in the VBDb and Libssh files presented here. The C Reconfigurator though can be extended with extra rules to deal with other cases that we did not face in our benchmark files. Worst case exponential growth of transformed programs can happen, even though we have not observed it in our subject files.

6.4 Discussion

The main limitation of our transformation based approach is that we may not obtain conclusive results for all individual variants, thus losing some precision. This is due to the fact that our transformed program contains all possible paths that may occur in any variant. However, the precision loss depends on the particular analysis we use.

Consider the case of model checking. Since (single-system) model checkers stop once a single counter-example is found in the model, we can use our approach to find a variability bug which occurs in some subset of valid variants but we will not be able to report conclusive results (whether the given property is satisfied or not) for the rest of the valid variants. To overcome this issue, we may repeat our technique on the remaining variants for which no conclusive results were reported in the previous iteration.

Consider the case of *must* dataflow analysis (e.g., available expressions, very busy expressions). In this case, the result in a given program point contains only the common results found on all execution paths to that point. Thus, the analysis result for the transformed program will contain only the results that occur in all variants. For example, for available expressions analysis we may obtain less available expressions than there are in any single variant. The available expression analysis determines which expressions must have already been computed, and not later modified, on all paths to a program point [32]. This information can be used to avoid re-computation of an expression. Consider the program family:

$$\mathsf{x} := a + b; \mathsf{while}\ (\mathsf{y} > a + b)\ \mathsf{do}\ \{\mathsf{\#ifdef}\ (A)\ \mathsf{y} := \mathsf{y} - 1\ \mathsf{\#else}\ a := a + 1\ \mathsf{\#endif}\}$$

The expression $a + b$ is available at the guard of the while loop for variants satisfying A, so it needs not be re-computed for them. However, in the transformed program we have paths from all variants, so the expression $a + b$ is modified by the assignment $a := a + 1$ in a path coming from variants $\neg A$. Therefore, the analyzer will not report this expression as available at the guard of the loop for the transformed program.

Consider the case of *may* dataflow analysis (e.g., reaching definitions, live variables, uninitialized variables). In this case, the result in a given program point contains the results found on at least one execution path to that point. Thus, the analysis result for the transformed program will contain all results that occur in at least one variant. For example, for live variables analysis, we may obtain more live variables than there are in any single variant. The live variables analysis determines which variables may be live at a program point, that is there is a path from the program point to a use of the variable that does not redefine it [32]. This information can be used as a basis for dead code elimination. If a variable is not live at the exit from an assignment to the variable, then that assignment can be eliminated. Consider the program family:

$\mathsf{x} := 5; \mathsf{y} := 1;$ #ifdef (A) $\mathsf{x} := 1$ #else $\mathsf{x} := \mathsf{x} + 1$ #endif

The variable x is not live at the exit from the first assignment x := 5 for variants satisfying *A*. Therefore, the assignment x := 5 is redundant for those variants. However, x is live for ¬*A* variants, so it will be live after the first assignment for the transformed program as well. Thus, we cannot eliminate this assignment in the transformed program. This is also the reason why Frama-C does not identify the variability bug for files 3, 20 and 21.

7 Related work

Recently, formal analysis and verification of program families have been a topic of considerable research. The challenge is to develop efficient techniques that work at the level of program families, rather than the level of single programs. There are two main approaches to address this issue: (1) to develop dedicated variability-aware (family-based) techniques and tools; (2) to use specific simulators and encodings which transform program families into single programs that can be analyzed by the standard single-program verification tools. The two approaches have different strengths and weaknesses. The advantage of (1) is that precise (conclusive) results are reported for every variant, but the disadvantage is that their implementation can be tedious and labor intensive. On the other hand, the approaches based on (2) re-use existing tools from single-program world, but some precision may be lost when interpreting the obtained results.

Specifically designed variability-aware techniques. Various lifted techniques have been proposed which lift existing single-program verification techniques to work on the level of program families. This includes lifted syntax checking [25, 20], lifted type checking [24, 8], lifted static analysis [7, 6, 31], lifted model checking [10, 14], etc. TypeChef [25] and SuperC [20] are variability-aware parsers, which can parse languages with preprocessor annotations. The results are ASTs with variability nodes. The difference between these two approaches is that feature expressions are represented as formulae in TypeChef, and as BDD's in SuperC. TypeChef has also implemented some variability-aware dataflow analyses. Several approaches have been proposed for type checking program families directly. In particular, lifted type checking for Featherweight Java was presented in [24], whereas variational lambda calculus was studied in [8]. Lifted model checking for verifying variability intensive systems has been introduced in [10]. SNIP, a specifically designed family-based model checker, is implemented for efficient verification of temporal properties of such systems. The input language to SNIP is fPromela, which represents a variability-aware extension of the known Promela language for the (single-system) SPIN model checker [21]. fPromela uses an #ifdef-like statement for encoding multiple variants, which represents a nondeterministic "if" statement guarded by features expressions that are used to specify what system parts are included (resp., excluded) for which variants. An approach for lifted software model checking using game semantics has been introduced in [14]. It verifies safety of #ifdef-based second-order program families containing undefined components, which are compactly represented using symbolic game semantics models [13, 12]. Brabrand et al. [7] and Midtgaard et al. [31] show how to lift any single-program dataflow analysis from the monotone framework to work on the level of program families. The obtained lifted dataflow analyses are much faster than ones based on the naive variant-by-variant approach that generates and analyzes all variants one by one. Another efficient implementation of lifted analysis formulated within the IFDS framework for inter-procedural distributive environments has been proposed in SPL$^{\text{LIFT}}$ [6]. In order to speed-up the lifted verification techniques, variability abstractions have been introduced in [17, 18, 15, 16]. They tame the exponential blowup caused by the large number of features and variants in a program family. In this way, variability abstractions enable deliberate trading of precision for speed in the context of lifted (monotone) data-flow analysis [17, 18] and lifted model checking [15, 16].

Lifting by simulation. Variability encoding [37] and configuration lifting [33] are based on generating a product-line *simulator* which simulates the behaviour of all products in the product line. Then, an existing off-the-shelf single-program analyzer is used to verify the generated product-line simulator, which represents a single program. The work in [37] defines variability encoding on the top of TypeChef parser for C and Java program families. They have applied the results of variability encoding to testing [26], model checking [3], and deductive verification [36]. Compared to [37], our approach has the following distinguished characteristics. C Reconfigurator is aimed at transforming C program families and uses SuperC as a back-end tool. We show transformation rules and their correctness with respect to a minimal C-like imperative (state-based) language, whereas in [37] the rules and their correctness is shown with respect to Featherweight Java. C is a language much wider used in industry for variability than (Featherweight)Java. Also, we do not have to rely on object-oriented encodings to make the variability-transformations work. We evaluate our approach with several state-of-the-art single-program verification tools for finding real variability bugs on real-world C programs (both on large and sanitized files). The academic examples (e-mail, elevator, mine-pump) considered by Apel et al. [3] are considerably smaller than those presented here; and they are focussed on verifying specific class of bugs: undesired feature interactions (using CPAchecker [5]), whereas we consider here various types of more severe bugs that occur in practice. In this way, the external validity of our experiments is considerably broader. Yet another difference is that the work in [3] considers product lines implemented using compositional approaches, where all features are modeled as separate and composable units. In contrast, we consider here annotative product lines based on #ifdef-s, which is a common way of implementing variability in industry.

8 Conclusion

We have proposed variability-related transformations to translate program families into single programs without variability. The transformed programs can then be effectively analyzed using various single-program analyzers. The evaluation confirms that some interesting variability bugs can be found in real-world C programs in this way. As a future work, we plan to extend our evaluation and consider more verification oracles as well as different practical case studies. We derive several observations from the attempt to verify, analyze, and find bugs in realistic C programs. We hope that our technique will be useful for future builders of analysis tools.

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: a qualitative analysis. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14*, pages 421–432. ACM, 2014. URL: http://doi.acm.org/10.1145/2642937.2642990, doi:10.1145/2642937.2642990.

[2] Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, and Andrzej Wasowski. Symbolic execution of high-level transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016*, pages 207–220. ACM, 2016.

[3] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *35th Intern. Conference on Software Engineering, ICSE '13*, pages 482–491, 2013.

[4] Don Batory. Feature models, grammars, and propositional formulas. In *9th International Software Product Lines Conference, SPLC '05*, volume 3714 of *LNCS*, pages 7–20. Springer-Verlag, 2005.

[5] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011. Proceedings*, volume 6806 of *LNCS*, pages 184–190, 2011. URL: http://dx.doi.org/10.1007/978-3-642-22110-1_16, doi:10.1007/978-3-642-22110-1_16.

[6] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spl$^{\text{lift}}$: Statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13*, pages 355–364, 2013.

[7] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development*, 10:73–108, 2013.

[8] Sheng Chen, Martin Erwig, and Eric Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*, pages 29–40, 2012. URL: http://doi.acm.org/10.1145/2364527.2364535, doi:10.1145/2364527.2364535.

[9] Clang. Clang static analyzer. Clang: a C language family frontend for LLVM. URL: http://clang-analyzer.llvm.org/.

[10] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Software Eng.*, 39(8):1069–1089, 2013. URL: http://doi.ieeecomputersociety.org/10.1109/TSE.2012.86, doi:10.1109/TSE.2012.86.

[11] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[12] Aleksandar Dimovski and Ranko Lazic. Compositional software verification based on game semantics and process algebra. *STTT*, 9(1):37–51, 2007. URL: http://dx.doi.org/10.1007/s10009-006-0005-y, doi:10.1007/s10009-006-0005-y.

[13] Aleksandar S. Dimovski. Program verification using symbolic game semantics. *Theor. Comput. Sci.*, 560:364–379, 2014. URL: http://dx.doi.org/10.1016/j.tcs.2014.01.016, doi:10.1016/j.tcs.2014.01.016.

[14] Aleksandar S. Dimovski. Symbolic game semantics for model checking program families. In *Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings*, volume 9641 of *LNCS*, pages 19–37. Springer, 2016.

[15] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. Family-based model checking without a family-based model checker. In *22nd International SPIN Workshop on Model Checking of Software, SPIN '15*, volume 9232 of *LNCS*, pages 282–299. Springer, 2015.

[16] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. Efficient family-based model checking via variability abstractions. *STTT*, 2016. doi:10.1007/s10009-016-0425-2.

[17] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses. In *29th European Conference on Object-Oriented Programming, ECOOP '15*, volume 37 of *LIPIcs*, pages 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[18] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Finding suitable variability abstractions for family-based analysis. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings*, volume 9995 of *LNCS*, pages 217–234, 2016. URL: http://dx.doi.org/10.1007/978-3-319-48989-6_14, doi:10.1007/978-3-319-48989-6_14.

[19] Alejandra Garrido and Ralph E. Johnson. Refactoring C with conditional compilation. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 323–326. IEEE Computer Society, 2003. URL: http://doi.ieeecomputersociety.org/10.1109/ASE.2003.1240330, doi:10.1109/ASE.2003.1240330.

[20] Paul Gazzillo and Robert Grimm. Superc: parsing all of C by taming the preprocessor. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, 2012*, pages 323–334, 2012. URL: http://doi.acm.org/10.1145/2254064.2254103, doi:10.1145/2254064.2254103.

[21] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[22] Alexandru F. Iosif-Lazar, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wasowski. Experiences from designing and validating a software modernization transformation (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 597–607, 2015. URL: http://dx.doi.org/10.1109/ASE.2015.84, doi:10.1109/ASE.2015.84.

[23] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[24] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14, 2012.

[25] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA'11*, pages 805–824. ACM, 2011.

[26] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *FOSD '12*, pages 1–8, 2012.

[27] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015. URL: http://dx.doi.org/10.1007/s00165-014-0326-7, doi:10.1007/s00165-014-0326-7.

[28] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Christian Kästner, and Sven Apel. An empirical study on configuration-related bugs. *Submitted for publication at IEEE TSE*, 2016.

[29] Jean Melo, Claus Brabrand, and Andrzej Wasowski. How does the degree of variability affect bug finding? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 679–690, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2884781.2884831, doi:10.1145/2884781.2884831.

[30] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Proceedings*, volume 7152 of *LNCS*, pages 146–161. Springer, 2012. URL: http://dx.doi.org/10.1007/978-3-642-27705-4_12, doi:10.1007/978-3-642-27705-4_12.

[31] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105:145–170, 2015. URL: http://dx.doi:10.1016/j.scico.2015.04.005, doi:10.1016/j.scico.2014.10.002.

[32] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Secaucus, USA, 1999.

[33] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *ASE'08*, pages 347–350, LAquila, Italy, 2008. IEEE Computer Society.

[34] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[35] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6, 2014.

[36] Thomas Thüm, Ina Schaefer, Martin Hentschel, and Sven Apel. Family-based deductive verification of software product lines. In *Generative Programming and Component Engineering, GPCE'12*, pages 11–20. ACM, 2012. URL: http://doi.acm.org/10.1145/2371401.2371404, doi:10.1145/2371401.2371404.

[37] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. Variability encoding: From compile-time to load-time variability. *J. Log. Algebr. Meth. Program.*, 85(1):125–145, 2016. URL: http://dx.doi.org/10.1016/j.jlamp.2015.06.007, doi:10.1016/j.jlamp.2015.06.007.