Enabling white-box reasonings on black-box composition operators in a domain-independent way

Benjamin Benni

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Un modèle de raisonnement pour les opérateurs de composition logicielle décrits en boite noire

Benjamin Benni

Laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (i3s)

Présentée en vue de l'obtention du grade de docteur en Informatique de l'Université Côte d'Azur

Co-dirigée par :
Sébastien Mosser, Professeur, Université du Québec à Montréal
et
Michel Riveill, Professeur des Universités, Université Nice Côte D'azur

Soutenue le : 09/12/2019

Devant le jury composé de :

Claudine Peyrat - Professeure des Universités, Université de Nice Côte d'Azur

Olivier Barais - Professeur des Universités, Université de Rennes 1

Gunter Mussbacher - Professeur associé, Université McGill

Lionel Seinturier - Professeur des Universités, Université de Lille

Un modèle de raisonnement pour les opérateurs de composition logicielle décrits en boite noire

Jury :

Présidente :

Claudine Peyrat - Professeure des Universités, Université Nice Côte D'azur

Rapporteurs :

- Olivier Barais - Professeur des Universités, Université de Rennes 1
- Gunter Mussbacher - Professeur associé, Université McGill
- Lionel Seinturier - Professeur des Universités, Université de Lille

Invités :

- Sébastien Mosser, Professeur à l'Université du Québec À Montréal (UQAM)
- Michel Riveill, Professeur des Universités à l'Université Nice Côte D'azur

Un modèle de raisonnement pour les opérateurs de composition logicielle décrits en boite noire

Résumé

La complexité des systèmes informatiques a rendu nécessaire leur découpage avant de les recomposer. Cette séparation est un défi connu et les développeurs découpent déjà les tâches au préalable. Néanmoins, séparer sans considérer la recomposition finale entraine des réunifications hâtives et chronophages. Cette composition doit mener au bon et meilleur système avec le minimum d'effort humain. Les opérateurs de composition sont souvent ad-hoc et développés par des non-spécialistes. Ils ne respectent pas de formalismes de haut-niveau et deviennent trop complexes ou informels pour pouvoir raisonner. Nous les appelons des "boites-noires" : les techniques nécessitant d'en connaitre l'intérieur ne peuvent être appliquées. Or, ces boites noires doivent garantir des propriétés : d'aucun doit vérifier son idempotence pour l'utiliser dans un contexte distribué ; connaitre son temps d'exécution pour des systèmes réactifs ; vérifier des conflits pour le confronter à des règles d'entreprise. Aucun de ces besoins n'est spécifique à un domaine applicatif. Dans cette thèse, nous présentons une approche indépendante du domaine qui permet, sur des opérateurs existants, (i) de raisonner sur des équations de composition pour (ii) les composer en sécurité, en (iii) proposant une vérification de propriétés similaires à celles de l'état de l'art. Nous avons validé cette approche sur des domaines différents : 19 versions du noyau Linux avec 54 règles de réécriture, réparé 13 « antipatrons » dans 22 applications Android et validé son efficacité sur la composition de 20k images Docker.

Enabling white-box reasonings on black-box composition operators in a domain-independent way

Abstract

The complexity of software systems made it necessary to split them up and reunite them afterward. Separating concerns is a well-studied challenge and teams separate the work to be done beforehand. Still, separating without considering the recomposition leads to rushed, unsafe, and time-consuming recomposition. The composition should create the right and best system with minimal human effort. Composition operators are often ad-hoc solutions developed by non-specialist development teams. They are not developed using high-level formalism and end up being too complicated or too poorly formalized to support proper reasonings. We call them "black-boxes" as existing techniques requiring knowledge of its internals cannot be applied or reused. However, black-box operators, like others, must ensure guarantees: one must assess their idempotency to use them in a distributed context; provide an average execution time to assess usage in a reactive system; check conflicts to validate that the composed artifact conforms to business properties. Despite the black-box aspect, none of these properties are domain-specific. In this thesis, we present a domain-independent approach that enables (i) reasonings on composition equation, (ii) to compose them safely, (iii) by assessing properties similar to the ones from the state-of-the-art. We validated the approach in heterogeneous application domains: 19 versions of Linux kernel with 54 rewriting rules, fixing 13 antipatterns in 22 Android apps, and validating the efficiency of the approach on the composition of 20k Docker images.

Thanks.

First, I would like to thank you, reader! Someone once told me that the best place to keep your money safe is your thesis because you'll be the only guy that will open it up. He was wrong! Or maybe you are reading an online version...

First of all, I would like to thank the members of my thesis committee. I believe reading a whole Ph.D. thesis is not an easy task, and I want to thank you for that. I am proud to present this work to you, and I hope you will find it understandable and exciting.

I would like to thank my advisors, Pr. Sébastien Mosser and Pr. Michel Riveill, for their involvement. Thanking advisors for being part of the Ph.D. thesis can seem silly, but I can assure you it is not. I met a lot of Ph.D. students, from various horizons, with different purposes, again, thanks to the kindness of my advisors, just to realize how lucky I was. I do not mean to be a brown-noser, but I think it is rare to find such dedicated advisors, that you can trust and that trust you in return. Special and additional thanks to Sébastien for his indefectible support which has appeared in surprising shapes and forms sometimes. Your repeated warnings about the effects of a Ph.D. did not make me give up, and I am proud of this adventure. Thank you for our calls, for all the opportunities you gave me, for your trust, your friendly advices, and for your couch. It was an amazing experience (the Ph.D., not the couch).

Part of the journey is the end, I guess, and what a journey it was. It is only years later that I realize how the human factor is a huge chunk of it. I was lucky enough to have a great mentor and be part of a team of passionate people. I remember when I first heard about what my advisor Sébastien did during his Ph.D. I remember how mind-blowing it was to discover a whole new universe. Guys working with meta-level for the first time understand what I mean. It is like these times when you connect things remotely located in your mind, things start to make sense, and you discover new meanings thank to this new connection.

Speaking of a team of passionate people, I would like to thank all the Sparks members for their welcoming. Thank you, Mireille, for your open-mind, your joy and our interactions scientific or not. Thank you, Philippe, for your constant providing of chocolates and jokes that helped me go through the tough times. Thank you, Gerald and Franck, for your friendly support at our coffee breaks whose hours are set with *Swiss-precision*. Thank you, Anne-Marie, for our early conversations in the morning that kickstarted my days and often change the way I see things.

During this Ph.D. I hold a teaching assistant position. I would sincerely like to thank Sébastien, Mireille, Philippe, Guilhem, Anne-Marie, Michel, and Erick for all of this. Thanks to your trust and advice, I was able to *build* things, courses, tutorings, projects, and learn a lot from both human and teacher perspectives. Thank you all for your kindness, professionalism, and for having taken the time to make me learn. Thank you for all the responsibilities you gave to me, and thank you for all the responsibilities you did not give to me.

As research is made of meeting new people, collaboration, exchange of ideas and philosophical discussions, I would like to thank all the researchers that also helped me be who I am today. Thank you Jean-Michel, Daniel,

Gunter, Jörg, Xavier, Houari, Eugene, and many others for our great discussions that started as formals during summer schools or conferences, and end up being arguing who is the fastest at skiing or won the last round in a board game.

Finally, I want to thank my family. A not-huge-enough thanks to Pascale, my Mom, who supported me, no matter what, no matter the cost, even in really tough times. Obviously, I would not have been the same without her energy to move forward. Thank you to my brother, Bastien, who has always told me that "things are going to be ok," for our breaks in the Bouchaniere's mountains, for being supportive even if you did not understand why I was doing all of this. Thank you Coline, for your lovely attentions, for preparing coffee at midnight during heavy periods of redaction, for acting as an audience when I rehearsed for a talk, and you tried your best to understand the pros and cons of "finding bad practices in container-based technologies using composition operator", for the balance you brought in all of this. Thank you for all the memories we shared during the last 9 years and a half, and for all those to come. Thank you to my friends Sébastien P., Anais, Nabil, Hugo, Maxime, and Franck for understanding that "I have work to do" was not an excuse, and for their support for the last 8 years now. Thank you to Coline's family, who also has been supportive, always in surprising ways. Thank you Maxime, Jean-Baptiste, Olivier, Thierry, Yaya, Hervé, Sylvie and others. The least I can say is that no one has an in-law family like this one. Thanks for the warm welcome and for the bottle of wine and others we shared that also helped for this Ph.D. to end.

Finally, I would like to thank Alfred for his indefectible support at the end of my Ph.D.

This Ph.D. thesis is dedicated
to my father, Jean-Louis.

"Bounce back, my dear friend."
J-L.B.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

" Integrating two software systems is usually more like performing a heart transplant than snapping together LEGO blocks. It can be done — if there's a close enough match and the people doing it have enough skill — but the pieces don't fit together trivially. And failure may not be immediately obvious; it may take a while to see signs of rejection. "

John D. Cook [1]

Content

1.1 Context & Issues

Software systems became so huge that it is mandatory to develop them pieces by pieces. We decompose a software system following different practices: given known guidelines, considering features of the final system, or given in which technologies each piece is developed. This decomposition leads to a (re)composition step afterward that is not trivial. Bad recomposition can lead to ill-formed, under efficient, or non-conforming systems.

Model-driven development aims to ease the building of such software systems by formalizing and ensuring functional and non-functional properties on the artifact manipulated along with their (re)composition. We saw in the last decades the development of model-driven approaches such as model compositions and transformations. These approaches enable reasonings on these compositions and allow one to *safely* compose them to ensure that applying many transformations on the same artifact will end up in a well-formed and sound system. These methods have a strong background formalism and are used in (meta-)modeling (*e.g.*, transformation of meta-model using graph rewriting techniques).

Software engineering is the application of engineering to the development of software in a systematic method. Even if its definition is not unique and well-formalized, it implies, as any engineering practices, *the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software* [2]. Therefore, this software engineering thesis analyzes real situations where the composition is used in an industrial context.

Software engineering implies to see what is going on in the field, and its main purpose is to help practitioners and actual developers to build better software systems more efficiently. Thus, with a software engineering scope, this thesis analyzes the field of practices and confronts it to the state-of-the-art assumptions and methods. It appeared that, whereas state-of-the-art approaches considered artifacts to be composed along with the composition itself as *white-boxes*, practitioners are facing *black-boxes* mechanisms when working with software composition. Their *white-box* approach enables properties verification such as termination or conflict analysis, useful in an engineering context to ensure that automated transformations will eventually end, or the identification of incompatible transformations for instance. Unfortunately, none of these properties can be evaluated yet in the context of black-box industrial tools.

1.2 Contribution

The contribution of this thesis is to investigate, *(i)* which properties are interesting in a black-box context, and *(ii)* why, and *(iii)* adapt them from a white-box context to a black-box one, enabling the assessment of properties *close* to the ones ensured by the white-box approaches, *(iv)* in a domain-independent way, our proposition coming as a support for already existing solutions. To fulfill such tasks, our solution bridges the state-of-the-art and state-of-practice approaches by formalizing a *delta-oriented* approach to model composition and properties verification in a black-box context. This *non-intrusive* approach acknowledges the existence of non-well-formalized operators, and comes as a support to domain-specific compositions, in a *domain-independent way*.

Our proposition relies on two assumptions to be applied: the existence of the *diff* $\ominus$ and *patch* $\oplus$ operators. The $\ominus$ operator enables our contribution to *capture* the behavior of a black-box composition operator, shifting from an entity-based composition to a diff-based one, allowing us to reason on those diffs in a domain-independent way. The $\oplus$ operator allows our contribution to be *operationalized* at the *application domain level*, using domain-independent definitions to actually apply the result of this diff-based composition on domain-specific artifacts. Using this formalized deltas, we provide a generic termination assessment, allowing one to ensure that a composition ended, and conflicts-detection operators that yield conflicts at a fine-grained level, providing knowledge to the user and ensuring the whole composition in a black-box context.

We studied the state-of-practice by analyzing compositions in various domains, industrial and academic, heavily controlled and not controlled environments [3], [4] and our contribution has been successfully applied to three different application domains: the development of the Linux Kernel, and the optimization of Android applications and Docker artifacts. This shows that bridging state-of-practice approaches that considered black-box compositions, with state-of-the-art approaches that perform diff-based compositions [5] is feasible and useful for the final end-user.

We applied our approach in the Linux kernel ecosystem, and 19 of its versions. Our contribution allows one to ensure its whole maintenance process, by assessing conflict-free compositions or yielding error along with the involved elements. We applied our approach at large scale in the Docker environment where our proposition ensures conflict-free situations where no guarantee was ensured beforehand among 19 rules applied to 12,000 dockerfiles. Lastly, we applied our contribution to the Android ecosystem and quantitatively evaluate the appearance of depicted composition issues.

We thus validate, *(i)* the feasibility of our approach, *(ii)* that our assumptions hold in real-life scenarios, *(iii)* that our contribution allows one to ensure conflict-free situation, or yield errors along with the involved elements, *(iv)* reducing the search space in case of conflict, enabling human-in-the-loop interactions, *(v)* in a domain-independent way as it can be applied in various domains.

1.3 Outline

Chapter 2 motivates this Ph.D. thesis by giving a portrait of different use-cases of composition, from a practitioner point-of-view. We outline the *need* to work with composition to build nowadays software and how software composition and transformation are performed in a *black-box* way in industrial contexts.

Chapter 3 depicts the state-of-art in the domain of this thesis: software composition and model transformations. We present a description of each approach, its hypotheses, and its role in an engineering context.

Chapter 4 expresses the challenges of reusing the white-box approaches depicted in chapter 3 on use-cases described in chapter 2. Then, it maps the state-of-practice depicted in chapter 2, with the state-of-the-art described in chapter 3. We highlight that guarantees ensured by the state-of-the-art do not hold in the presented contexts, and define two main challenges to bring good white-box properties in a black-box context. We conclude by stating that no white-box properties can be ported *as-is* in a black-box context and that compromises must be made to ensure such properties in a best-effort way.

Chapter 5 describes the formal model that we propose to build a bridge between state-of-the-art approaches and black-box state-of-practice, by the use of delta-reasonings. We leverage two hypotheses on the application domain, to bring back the reasonings on black-box operators to reasonings on sequences of modifications.

Chapter 6 performs an in-depth large-scale validation on the Linux-kernel. It depicts how one can apply our contribution to a narrow environment. We assess that our proposition can be applied in an industrial use-case, where developers' skills and heavily controlled development should prevent any issues. We quantitatively validate that issues arise, even in this context, and successfully ensure conflict-free situation and yield non-conflict-free contexts.

Chapter 7 performs a wide large-scale validation on the Android and Docker environments. It depicts how one can apply our contribution to these two non-controlled contexts. We quantitatively validate that composition issues arise in such a context, and conflict even at a fine-grain level. We describe how such composition issues can badly affect the final composed artifact from a domain point-of-view. This chapter successfully validates that our contribution is domain-independent, and can be applied to various application domains.

Chapter 8 concludes this thesis by summarizing the context and the outcome of applying our proposed contribution in various domains. This chapter also depicts the perspectives of work that may follow this thesis. We highlight the need for reasoning at a meta-level, *i.e.*, on the composition of composition, and sketches perspectives related to extending the contribution of this thesis.

1.4 Publications

The research work done during this PhD has lead to the following peer-reviewed publications depicted below, along with still unpublished research results which are still in progress and described at the end of this section.

Journal

B. Benni, S. Mosser, N. Moha, *et al.*, « A delta-oriented approach to support the safe reuse of black-box code rewriters », *Journal of Software: Evolution and Process*, vol. 31, no. 8, e2208, 2019, e2208 smr.2208. DOI: 10.1002/smr.2208. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2208. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2208. This publication is an invitation to extend a previous conference paper to a journal article [5]. We applied the proposed approach and abstractions to real use-cases, one industrial and one academic validating the feasibility of our approach, and concretely measuring the outcome enabled in two various domains. The former use-case is the maintenance of the Linux kernel where our proposition successfully ensured syntactical-conflict-free situations, and successfully yielded narrowed semantic-conflicts. The latter analyzed depicted composition issues in the context of optimizing Android applications.

Conferences

— B. Benni, S. Mosser, N. Moha, *et al.*, « A Delta-oriented Approach to Support the Safe Reuse of Black-box Code Rewriters », in *17th International Conference on Software Reuse (ICSR'18)*, Madrid, France, 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01722040. In this publications, we lay out the foundations of our formalism and depict the proposition, hypotheses, outcome, and feasibility of our approach. We defined a diff-oriented approach to composition operators, allowing us to formalize conflict-checkings on black-box composition operators. Our proposition allows one to shift from an domain-specific entity-based composition to domain-independent diff-based one. It has been validated at large scale in [3].

— B. Benni, S. Mosser, P. Collet, *et al.*, « Supporting Micro-services Deployment in a Safer Way: a Static Analysis and Automated Rewriting Approach », in *Symposium on applied Computing*, Pau, France, Apr. 2018. DOI: 10.1145/3167132.3167314. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01659776. In this publication, we studied an industrial composition operator inside the Docker ecosystem. We proposed a formalism that allows one to reason on Docker artefacts, for various purposes. We validated, at large-scale on more than 23,000 Docker artefacts, that our proposition was feasible, and provided a meaningful outcome for Docker developpers and the Docker ecosystem. We outlined that some issues would not have been detected without a composition approach.

Workshops

— (*Proceedings not published yet*) DevOps@Models, Workshop of Models'19 conference. In the first edition of this workshop, we studied and raised questions and issues, related to the meaningful and efficient building, testing, and assessing, of machine learning pipelines. As the number of pipelines possible is too massive to be handled, even by machines, we applied composition mechanisms and reasonings allowing us to drastically reduce the number of pipelines that one can build or test.

— B. Benni, P. Collet, G. Molines, *et al.*, « Teaching DevOps at the Graduate Level: A report from Polytech Nice Sophia », in *First international workshop on software engineering aspects of continuous development and new paradigms of software production and deployment*, LASER foundation, Villebrumier, France, Mar. 2018. [Online]. Available: `https://hal.archives-ouvertes.fr/hal-01792773`. This publication depicts how DevOps is taught in Polytech Nice Sophia Antipolis. We described the issues that arise when software development effort is spread among different teams, with different skils, and that the whole system has to be composed, tested, and deployed as fast as possible.

In Progress

Some of the works that are part of this Ph.D. thesis have not yet been published and are described below.

— ECMFA 2020 - 16th European Conference on Modelling Foundations and Applications. In this publication, we propose a property-based testing approach to characterize black-box composition operators. We propose a domain-specific language that allows a domain-specialist to specify her composition operator and the manipulated elements. The DSL also allows one to specify the algebraic properties (*e.g.*, commutativity) that one wants to assess along with the equivalence-relation to be used to assess such properties. Then, the DSL generates portion of Java code that a domain-specialist needs to complete to use and call her operator effectively; allowing the experiment to be run and the measurements to be made in a semi-automated way.

— TSE - IEEE Transactions on Software Engineering. In this publication, we analyze conflicts-solving in the context of code versioning. We investigate the state-of-the-art results regarding the automated solving of code merge-conflicts. These conflicts happen when multiple developers worked on the same codebase in parallel and is known to be a complex and time-consuming task. The early results, based on an open conflicts-dataset, is that state-of-the-art techniques fail to automatically and correctly merge the majority of merge scenarios. Our approach is to investigate the code merge-conflict as a composition issue, where sets of modifications have to be applied on a codebase safely, to ultimately improve the handling of conflicts, easing developer's life by improving automation, and the overall quality of the merge process.

CHAPTER 2

Motivation

" A Captain ought, among all the other actions of his, endeavor with every art to divide the forces of the enemy, either by making him suspicious of his men in whom he trusted, or by giving him cause that he has to separate his forces, and, because of this, become weaker "

Niccolò Machiavelli, The Art of War, Book VI, 1521

Content

Software composition arose from the need to decompose things. Following different practices, meant to achieve different goals in various domains, composition was developed, refined, and widely used. These different uses lead to *domain-specific applications*: composition arose in a specific domain as a *tool* to achieve a given *goal*. Thus, this scattering curbs or prevents any cross-domain reuse of development effort made towards composition-based tools. This section gives background history on software compositions, why it arose, and how one can *use* compositions. Such black-box mechanisms can be found more and more with the emergence of AI systems, black-boxes by essence.

2.1 Ultra-Large Scale Systems

When building a software system, developers have to consider the software itself along with its surroundings. At the early ages of computers and software development, the surroundings were limited to the input the program should take, and the outputs it produces. We focused on low-level concerns such as "*is the input data on the stack*" or "*the registry in which the processed data should go*". The system under consideration evolved in a tiny isolated bubble.

> Programming [before the 1970s] was regarded as a private, puzzle-solving activity of writing computer instructions to work as a program.
>
> Harlan Mills [7]

Then, everything changed when computers started to interact with each other and gained in computing power and overall capabilities (*e.g.*, memory, storage capacities). Progressively, the surroundings-frontier moved outward, and its notion changed: we had to take more and more information into account when building a system, we needed to consider satellite systems, how to interact with the other systems, by which mean, following which protocol. The bubble expanded.

Eventually, this led to an even larger structure, where many systems have to interact with each other, at a large scale, to achieve various goals. We thus consider the interactions between large bubbles, leading to scaling issues, paradigm and language heterogeneity, various concerns, multiple stakeholders.

We are now, and for more than a decade, in the *Ultra Large Scale Systems* (ULSS) era: highly complex systems that try to sense the reality of a given field, and take action in the real world, thus changing its context of execution. These ULSS are characterized by their intrinsic complexity, implied by numerous factors among the amount of data to be handled, the volume of lines of code to consider, the number of people themselves involved and their respective roles, or the need for the system to evolve. These ULSS can be found in the evergrowing apparition of software-intensive systems such as autonomous cars, power-grid regulation, or smart cities, that come with a new set of challenges.

> The U. S. Department of Defense has a goal of information dominance to achieve and exploit superior collection, fusion, analysis, and use of information to meet mission objectives. This goal depends on increasingly complex systems characterized by thousands of platforms, sensors, decision nodes, weapons, and war fighters connected through heterogeneous wired and wireless networks. These systems will push far beyond the size of today's systems and systems of systems by every measure: number of lines of code; number of people employing the system for different purposes; amount of data stored, accessed, manipulated, and refined; number of connections and inter dependencies among software components; and number of hardware elements. They will be ultra-large scale systems.
>
> Northrop, Linda, et al. Ultra-large-scale systems: The software challenge of the future. 2006 [8]

2.2 Separation of Concerns: Divide to Conquer

We can no longer, and it has been that way for a while, build an entire software system with a single team, in a single shot. We need to *(i)* split things up, and *(ii)* build iteratively. The former is a quite old idea introduced by Edsger W. Dijkstra (which among other things is one of the pioneers of the software engineering discipline) that states that splitting a huge problem into smaller ones eases the finding of a solution; the latter is an idea that states that we simply cannot build a full software from scratch in a single shot and that many tries and validation are needed along the way.

This separation of concerns principle, applied to the software development process, means that we will build different parts of the software in parallel, knowing that other parts exist, but focusing only on an aspect of the final system at a time. This separation can be technical or business-driven.

> We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained —on the contrary!— by tackling these various aspects simultaneously. It is what I sometimes have called "the **separation of concerns**", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focussing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.
>
> On the role of scientific thought, E.W. Dijkstra, 1974 [9]

No matter *how* the system was split, given *which* guidelines, but at some

point, one needs to *recompose* every concern, every "sub-systems" into the final one. This can also be done iteratively and/or interactively and at different levels. The (re)composition can be done for instance at the model-level (*e.g.*, composing UML class-diagrams [10], [11]), or at the code level (*e.g.*, merging code modifications [12]), using different techniques such as match-and-merge, or weaving; hence, we talk about *composition*. The composition can also impact the structure of the whole system or change its behavior.

> "When we say "separation of concerns" we mean the idea that it should be possible to work with the design or implementation of a system in the natural units of concern – concept, goal, team structure etc. – rather than in units imposed on us by the tools we are using. We would like the modularity of a system to reflect the way "we want to think about it" rather than the way the language or other tools force us to think about it. In software, Parnas [13], [14] is generally credited with this idea."
>
> An Overview of AspectJ, Gregor Kiczales et al. [15]

2.3 Example: Composition in the Internet of Things

This section depicts a use-case that will serve as a motivating example. It is not meant to be an example of *real* scaling and composition issues but is a voluntarily small example to exemplify all that has been said before. We will take a simplified version of the context of Cyril Cecchinel's Ph.D. thesis [16] in which he is deploying software on physical sensors inside smart buildings.

The domain of the *Internet Of Things* (IoT) aims to sense the world through sensors and take automated decisions to take actions on the real world (*i.e.*, its context of execution). IoT addresses the smart-cities use-case, for instance, in order to improve the city in terms of energy consumption, traffic, or well being. This whole smart-city example is way too complex to be handled on its own, as a single piece. Even at a lower scale, handling only energy consumption is a far too complex task.

Thus, in this domain, *experts break down the complexity* by splitting the problem into smaller ones. Each expert will work on a single aspect of the whole system. In the context of Cecchinel's work, each expert will focus on a single aspect of a smart building, *e.g.*, security, air conditioning, energy consumption. Then, concerns are composed together, and the result is deployed onto a sensors network infrastructure.

The common representation used to model the complete or partial behavior of a sensor network is an *automaton*. Thus, each automaton defines the behavior of a part of the network, representing a part of the whole behavior. Then, all automata are "merged" together, to form a big(ger) and complex automaton, that one could not have conveniently defined on its own from scratch. At this stage, this final automaton can be interpreted, checked against expectations, guidelines, optimization rules, and may eventually be transformed into a set of running soft-

ware deployed on actual sensors, actuators, and gateways. As the size of the final automaton is an essential factor in the deployment process, one must ensure that it cannot be further optimized.

2.3.1 Domain-Specific Use-Cases

Let us take a small and concrete example here. The goal is to illustrate each step on a small and understandable example. This subsection describes what we mean by "composition" in this thesis without going straight to a big and industrial use-case. Let us assume that we have two experts, each one focused on a single aspect of a smart building. We will handle two common use-cases in smart buildings context: automated handling of air conditioning, and automated security checks of entry points in the building.

Automated handling of air conditioning (`UC1`). This use-case is meant to reduce the overall consumption of the building while keeping the same level of comfort for users. Part of this goal is to cut-off the AC of an office when its door is opened and put it back on when the door is closed.

Automated security checks of entry points (`UC2`). This use-case is meant to automatically check that entry points such as doors and windows are closed to ensure a global security level in the building for potential intruders. Part of this goal is to raise an alert when a door is opened for too long.

Both of these use-cases target smart-buildings context and will be actually deployed on a real building. The use-cases have to be defined by the experts separately; then, the two solutions must be automatically composed together and be deployed on a sensor network.

Both of the experts' solutions are using the door in their respective use-cases. One will merge their solutions before being deploying them. In another context where two distinct and widely different use-cases would have been defined, the composition would not happen as the use-cases target different sensors, *i.e.*, their intersection is empty.

2.3.2 Modeling of Seperate Domain-specific Use-cases

FSM and language involved. One way to model these kind of behaviors is to use **F**inite **S**tate **M**achines (FSM). Each use-case will be modelled separately as a FSM, and will both be working on the following alphabet:

— '`o`': an event has been fired stating that the door of the office has just been *opened*.

— '`c`': an event has been fired stating that the door of the office has just been *closed*.

— 't_h': an event has been fired stating that the temperature is too high and needs to be *cooled*.

— 't_l': an event has been fired stating that the temperature is too low and no longer needs to be *cooled*.

Modeling of `UC1`**.** The first use-case (`UC1`) is defined as follow:

1. Put the AC on (ON),
2. When the door has been opened (event `o`), or when the temperature is low enough (event t_l), shut off the AC (OFF),
3. When the door is closed (event `c`), enter a waiting state ($WAIT$)
4. from this state, either the door is opened again (`o`), and it shuts does the AC, or
5. the temperature is high enough (t_h), it puts the AC back on (ON).

The FSM of FIG. 2.1 models the use-case 1 by recognizing the language $(o|t_l)c(oc)^*t_h$.

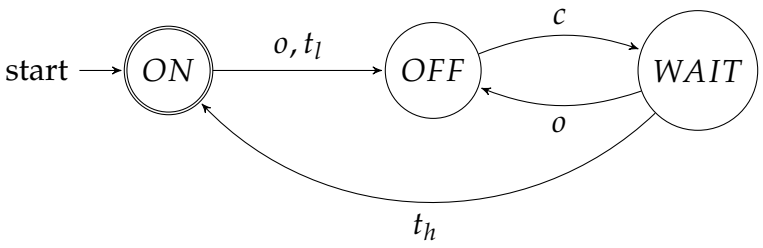


Fig. 2.1 – FSM_{exp1} as designed by the domain expert of `UC1`

Modeling of `UC2`**.** The second use-case (`UC2`) is defined as follow:

1. Waits for the door to be opened and the alarm is set to off (OFF),
2. When the door has been opened (event `o`), trigger an alert to the central (ON),
3. When the door is shut (event `c`), extinguish the alert and wait again (OFF).

The FSM of FIG. 2.2 models the `UC2` by recognizing the language $(co)^+$.

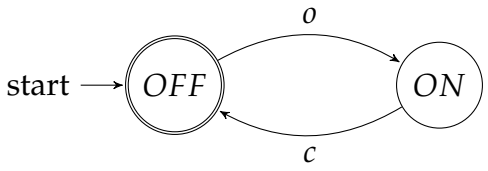


Fig. 2.2 – FSM_{exp2} as designed by the domain expert of `UC2`

2.3.3 Matching distributed context.

In the previous subsection, domain experts have designed finite state machines to match their respective use-cases. The behaviors described by these FSMs will be deployed on a sensor network inside a smart building. As sensor networks are a distributed environment, messages are not reliable and may be delivered more than once. Finite state machines defined by the experts need to be adjusted to match the operational context. Hence self-looping edges must

be added on nodes that do not fully specify outgoing edges for the respective alphabets of each FSM. Self-looping edges must be added on ON, OFF, and $WAIT$ of L_{UC_1}, and similar edges must be added on states of `UC2`. This adjustments are not part of the smart-building expertise and is part of the operational context.

2.3.4 Matching requirements of composition operator.

Avoiding conflicts. At this stage, states are automatically renamed to avoid any upcoming conflicts in the names of the states of both automata. Thus, states of `UC1` are denoted by `p`, and a number is appended; whereas states of `UC2` are denoted by `q`.

Aligning alphabet of FSM The finite state-machines developed by the experts are not working on the same alphabet. As the merge process work on FSMs using the exact same alphabet, the FSMs need again to be transformed to match requirements from the composition operator this time. Hence, the finite state machine of `UC2` needs to be transformed to take into account the symbols 't_l' and 't_h'. Self-looping edges are added to the states of the FSM with these symbols.

New FSMs At this stage, the FSMs have been transformed to match operational context, to avoid upcoming conflict in the merge process, and to match the expectations of the composition operator that will be used later. These FSMs recognize languages different than the one defined by the experts but are actually more permissive. Thus any contexts matched by the domain experts' FSM (*i.e.*, FMS_{exp}) will be matched by this updated FSM. Nevertheless, each transformation is expected to ensure that it will not reject previously matched context but may accept previously unmatched context (*e.g.*, FSM_{UC_2} now accepts words containing t_l or t_h). The resulting finite state machines FSM_{UC_1} and FSM_{UC_2} are depicted in FIG. 2.3.

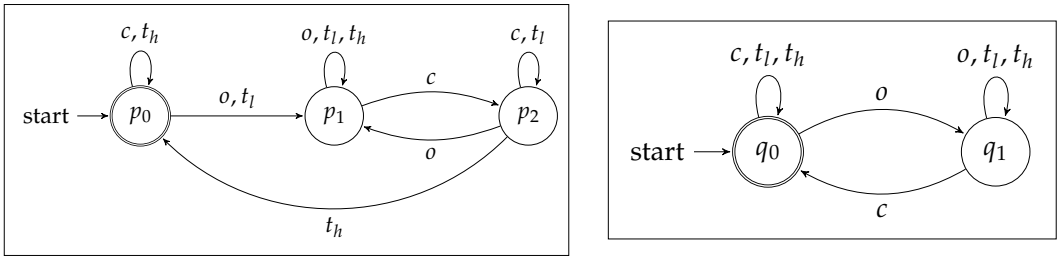


Fig. 2.3 – FSM_{UC_1} (left) and FSM_{UC_2} (right) follow operational requirements and expectations of the composition operator

2.3.5 Merging Behaviors by Merging FSMs

Different operators and different semantics. Once these two FSMs were specified and developed in isolation, transformed to match the operational context of distributed systems, renamed to avoid conflicts, and have their alphabets aligned, one can merge them to obtain a FSM modeling both of these behaviors [17]. Nevertheless, the semantics of the merge operator used will impact which final be-

havior is captured by the merged FSM. To merge FSMs, one can use different operations, each following a different semantic, notably among:

— **Union**: accepts a word if it is in L_1 **OR** in L_2. In our example context, this means that the overall behavior is to capture one of the use-cases but not necessarily both.

— **Intersection**: accepts a word if it is in L_1 **AND** in L_2. In our example context, this means that the overall behavior is to capture both of the use-cases at the same time.

— **Concatenation**: accepts a word if it can be cut as a word of L_1 **followed** by a word of L_2. In our example context, this means that the overall behavior is to capture a use-case, directly followed by the other.

Given the depicted semantic of the operators available, an expert will use either the **Union** or **Intersection** operations. The procedures triggered when using these two operators are heavily similar; thus, we will perform an **Intersection** and outline the differences that would have occurred if one would have used the **Union** operator.

Paradigm shift to perform the composition. The intersection is made by shifting the model-representation: one does not build the result of the merge from the automata themselves but build a transition table, made from a cross-product of every state of the two automata to be combined [18]. This procedure is deterministic and has a clear semantic. The resulting table is depicted in TAB. 2.1. One should read this table as follows: "when in state p_0q_0, and 'o' event happens, transit current state to p_1q_1". When using the **intersection** operator, final states are the ones that are finals in the first automaton *and* in the second automaton.

Table 2.1 – Transition table resulting of the intersection of FSM_{UC_1} and FMS_{UC_2}

State \ Input	o	c	t_l	t_h
$p_0q_0{}^*$	p_1q_1	p_0q_0	p_1q_0	p_0q_0
p_1q_1	p_1q_1	p_2q_0	p_1q_1	p_1q_1
p_1q_0	p_1q_1	p_2q_0	p_1q_0	p_1q_0
p_2q_0	p_1q_1	p_2q_0	p_2q_0	p_0q_0

At this level, we are far from smart-building expertise: the abstractions are different, the language and vocabulary used are not the same, the result of the composition nor its inputs have been written by none of the experts. All the work, transformations, and reasonings applied from this stage is out of the scope of the smart-building experts' world and should be done automatically.

Figure 2.4 depicts the raw FSM, named $FSM_{\cap}^{R}$, made directly from the transition table described in TAB. 2.1.

Optimization by equivalent-group reduction. At this stage, the transition table, and thus the final $FSM_{\cap}^{R}$, is not optimal in terms of size. For storing usage, bandwidth restriction, or future algorithms exponential explosion, it may be better to keep the finite state machine as small as possible. Thus, the transition table and $FSM_{\cap}^{R}$ can be optimized by minimizing its number of states and transitions.

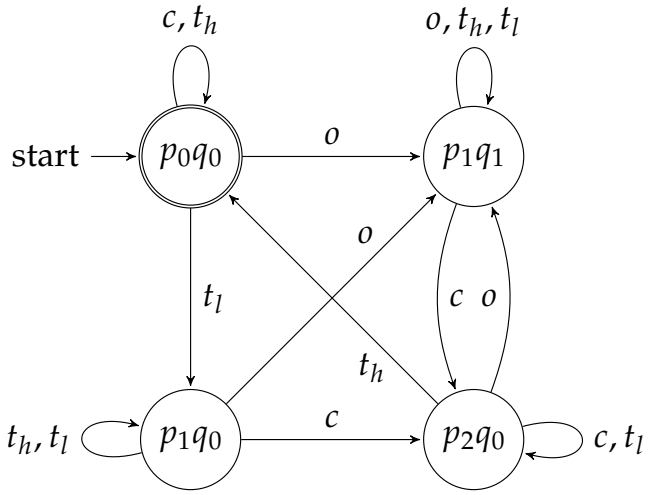


Fig. 2.4 – $FSM_{\cup}^{R}$ representing the minimized union of $FSM(L_1)$ and $FSM(L_2)$

The equivalent-group procedure takes the transition table as input and yields a set of sets of equivalent-states. This *minimization* procedure is a well-known, deterministic procedure, and is an optional extra-step of the composition step. We will not go into details of this procedure but kindly redirect the reader to the relevant articles [19], [20], but the equivalent groups are listed in TAB. 2.2.

Table 2.2 – Equivalent states as establised by equivalent-group optimization

Step	States groups
0 equivalent	$[p_0q_0]$ $[p_1q_1, p_1q_0, p_2q_0]$
1 equivalent	$[p_0q_0]$ $[p_1q_1, p_1q_0]$ $[p_2q_0]$
2 equivalent	$[p_0q_0]$ $[p_1q_1, p_1q_0]$ $[p_2q_0]$

The result of the minimization is depicted in 2.5 where the minimized $FSM_{\cap}^{M}$ contains 3 states, 25% less than the initial $FSM_{\cap}^{R}$.

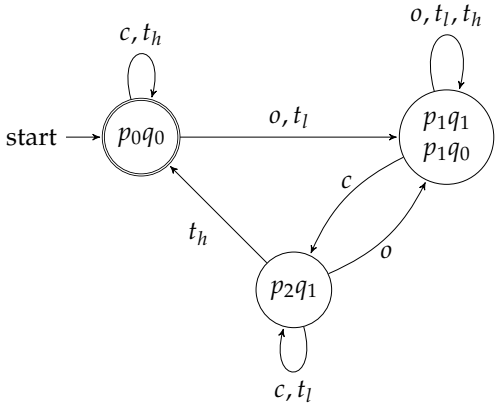


Fig. 2.5 – $FSM_{\cap}^{M}$ representing the minimized intersection of FSM_{UC_1} and FSM_{UC_2}

End to end, to obtain the minimized $FSM_{\cap}^{M}$ involved three pre-composition transformations:

— Make the FSM complete to match operational context,

— Align alphabet to conform to the composition operator requirements,

— Rename states to avoid collision during the composition process.

Then, in addition to these transformations, one out of the three following composition operations is applied, each one involving three different representations:

— Finite state machine to *represent* automata,

— Transition-table to *perform* the composition, which is a morphism of the FSM,

— Group-equivalence to *minimize* the automaton.

2.3.6 Issues and Conclusion

As the post-merge process involves many different procedures (*e.g.*, make the automaton deterministic, minimize its number of states and transitions, remove unreachable states), in which sequence is one supposed to run them? Is there a *correct* sequence? Does the order matter? Even with only four different procedures to be applied, it represents twenty-four different arrangments. How can one still find a proper sequence, without running and manually analyzing the whole 24 different sequences and find the one(s) that is(are) correct? Is the union of the minimized automata equivalent to the minimization of the merged automaton? **How one can safely compose these operators?** For instance, the post-merge operations aim at optimizing the resulting automaton. How one can be informed if rewriting rules overlap each others? Is there a particular order in which they should be run? How to ensure the application of each and all rewriting rules, as we cannot know nor interpret their internals? The issue is to list useful properties that need to be assessed and check them against existing black-box composition operators in a domain-independent way.

This example depicts the kind of composition this Ph.D. thesis targets: already existing use-cases of composition, involving multiple transformations, which are not formalized or even known. This composition may imply paradigm shifts and different variants of the same operator. Still, developers in these contexts need to ensure that multiple transformations are applied correctly.

2.4 Composition in the wild

This section gives a quick overview of various actual use-cases where software composition has a role in an industrial context. This section is not meant to be an exhaustive list but to give an insight into the variety of domains in which composition has a role in and characterize “real” use-cases using compositions. These use-cases have been selected because they are used in industrial contexts, and are overall representative of the artifacts with which developers interact.

2.4.1 Service deployment

Containers abstract software from the hardware environment and allow one to wrap a service and its related dependencies into an artifact. Each service can then be run into a *container* that ensures that the embedded software will run the same regardless of their real environment, easing the repeatability of build, test, and runtime executions. Containers are built from textual configuration files, often using a pre-defined set of instructions. However, such configuration is reused as off-the-shelf black-boxes. This black-box reuse easily leads to many forms of unexpected behaviors and does not help the one reusing or extending such artifacts in knowing what it does when she writes her own container's build file. The composition operator is hidden, and not known by the end-user, yet its result is known as it is the final container to-be-deployed. Thus, by essence, the composition operator is a black-box. In this context, Docker [21] is the leading container technology that offers, mid-2019, more than 5.8 million applications in its hub, downloaded more than 105 billion times [22] to be reused in a black-box way.

2.4.2 Code management

Developing software involves many people working on different parts or aspects of a final product. To ease and speed up the development process, code management tools such as Git [23] allow teams to version their code and develop features in parallel. At some point in time, when the team is ready, a different development branch will be merged to augment the set of features that a code fulfills. Merging different development branches is a tedious and complex task and is meant to be as much automated as possible. Code merges can even become a time-consuming operation, if they involve a huge set of code, *i.e.*, merging very different branches. As always in software engineering, it is a trade-off issue: one can use sophisticated algorithms and techniques to automate as much as possible, with a cost of a heavy load and high computing time; or perform the merge as fast as possible, and ask a human to perform what has not been done yet. GitHub, the most used hosting service for version control, hosted in mid-2018 100 million repositories [24] and had mid-2019 more than 32 million users [1]. This platform groups, mid-2019, more than 1.1 billion contributions and more than 200 million contributions from outside the development teams [25]. This shows the scale of the actual software development and the number of people who will benefit from smarter code-management tools.

2.4.3 Linux maintenance

Linux is one of the most-significant community-based software development in terms of longevity, volume of code, and number of contributors. As of 2017, 15,600 developers and growing, and more than 1,400 companies have contributed to the kernel [26]. Mid-2019, the official Linux repository showed more than 840,000 contribution [27]. As it involved different profiles of contributors, guidelines have been created, and contributions have to be formatted to conform to them. As common mistakes occur, they also need to be fixed. Finally, as the kernel evolves between a contribution and its integration into the kernel, they also

1. Result of the request `https://github.com/search?q=type:user&type=Users`

have to be modified. All these modifications have been as automated as possible to avoid unnecessary human labor. Rewriting rules are applied onto the source-code and fix a specific issue (*e.g.*, guideline, error, evolution). Because the Linux kernel is huge, and each rule is applied on the kernel source-code, a batch of rewritings takes a long time, hours on average. As multiple rules may be applied on the same portion of source-code, the order in which rules are applied may yield different results. As overlapping rules are applied in sequence, they silently fail without notice. Currently, the Linux kernel contains 62 rules [2].

2.4.4 Android automatic optimizations

`Android` is the most used operating system for smartphones with more than 2 billion monthly active users [28], and covering 85% of the mobile market [29]. As every embedded system `Android`, along with its applications, has to take energy consumption as an essential concern. Good and bad development practices evolve along with the development of `Android` itself, and some guidelines are even bad practices in another context such as plain object-oriented development. Thus, tools such as `Paprika` [30], aim to automatically detect these identified issues and tools such as `SPOON` [31] will modify the Abstract Syntax Tree of a codebase to fix the issue effectively. `Paprika` developers proposed a set of rules that conform to `Google` guidelines at the time. Applying related `SPOON` rules should lower the energy consumption of the application (if related issue has been detected in the codebase of course), applying two should lower even more, and so on. Thus, the most power-efficient version of the application should be obtained by applying each and every rule on an `Android` application. Yet, *sometimes*, it is not the case. Applying the whole rules set is worse than just applying a few of them or even none of them. How can one explain this? Is it due to overlapping rewriting rules? As rewriting rules are plain java codes, with heavy use of reflexivity and without normalization, it is quite hard or even impossible to infer the rule behavior and moreover to check if a pair of rules overlaps at a human scale.

2.4.5 Catalog construction

Catalogs represent a set of products that one can build and deliver. A way to automatically build a catalog is to build it from the description of products. The most used formalism to achieve such a goal is to represent products as *Feature Model*s (FM) that will be *merged* to obtain the final FM that represents the catalog [32]. The *merge* operation is known to be a complex and time-consuming task. As it is often used inside automatic processes, time-consumption is critical. Users of this operator report that placing the biggest FM as first operand has a substantial impact on execution time, are they right? How does the execution time evolve according to the size of input-FMs? Users of this operator also want to know if they can use it in a distributed context, *i.e.*, does the operator *supports* the merge of the *same* FM *multiple* times? As this merge operation involves paradigm-shifting where the input FMs are transformed into logical formulae, passed to a black-box solver, then the final FM is synthesized from the output of the solver thanks to a non-deterministic and heuristics-based process, the analy-

2. `https://github.com/torvalds/linux/tree/master/scripts/coccinelle`

sis of the merge process itself is not possible. Thus, one will consider the merge operator as a black-or-grey-box.

2.5 Conclusion

This section depicted industrial use-cases where composition occurs. We quickly summarized the role the composition occupies in each use-case, and the issues it solves at the domain-level. The described examples have shown that composition can come in different shapes and forms, and may serve different purposes, in very different contexts, targetting different artifacts. In every domain, composition plays a crucial and central role, answering a need to manage the recomposition of a decomposed problem that was too complex to be handled on its own. Composition operators are at the frontier between different teams (*e.g.*, developers, operationals) and a tremendous number of stakeholders use them every day. The need for insurances and safe compositions is crucial in such a context, and state-of-practice lacks both of them. In the next chapter, we will take a step-back and depict the state-of-the-art transformations and compositions, performing at the model-level or code-level. Then, we will match the state-of-practice, described in this chapter, against the state-of-the-art, described in the next chapter, and we will ultimately define the challenges that this thesis addresses. In the context of this thesis, we will use the use-cases depicted in Sec. 2.4.1, 2.4.3, and 2.4.4 to validate the results of this work.

CHAPTER 3

State of the art

" it is not the most intellectual of the species that survives; it is not the strongest that survives; but the species that survives is the one that is able best to adapt and adjust to the changing environment in which it finds itself. "

Leon C. Megginson [33]

Content

3.1 Introduction

In the previous chapter, we described the use-cases of compositions we want to address in this thesis, and we presented a classical example of composition which is the merging of automata. In the context of this thesis, we are interested in a family of composition which capture the use-cases described in the previous chapter. For the merge-of-automata example, there exists a language and a meta-model; thus, the automata-composition issue is brought back to composing models that conform to their meta-model. The goal of this chapter is to analyze the approaches that exist in the literature to compose or transform models.

To scope the content of this chapter, we address in this thesis structural compositions that statically modify the initial artefacts as this matches the use-cases we described in the previous chapter. One of the central areas of the structural composition is *model-composition*. As composition is a subset of *transformation* (the approaches of the former reusing techniques and tools of the latter), we will address model transformations first in this chapter; then, we will dive into composition approaches at the model-level and at the code-level that exist in the state-of-the-art.

3.2 Model Transformations

The most high-level manipulation of models of any kind is transformation. Before diving into composition at the model and code-level, this section analyzes the state-of-the-art of model transformations. This section illustrates the most widely used approaches in the model transformations ecosystem.

QVT (**Q**uery/**V**iew/**T**ransformation), is a standard set of languages for model transformation defined by the Object Management Group (OMG) [34]. It defines three model transformation languages that operate on models that conform to Meta-Object Facility metamodels. QVT is used for expressing M2M transformations only as QVT languages do not permit transformations *to* or *from* textual models, since each model must conform to some MOF metamodel. Model-to-text transformations are being standardized separately by OMG via MOFM2T. QVT is open to extensions as QVT-BlackBox allows one to invoke transformation facilities expressed in other languages (for example XSLT or XQuery), as long as the manipulated elements are EMF models.

MOFM2T (**MOF M**odel to **T**ext **T**ransformation Language) is an OMG specification for a model transformation language that can be used to express transformations that transform a model into text (*e.g.*, a model into source code or documentation) [35].

ATL ATL is a model transformation language developed by OBEO and Inria to answer the QVT Request For Proposal [36]. It is built on top of a model transformation Virtual Machine and allows one to perform syntactic or semantic translation. An ATL transformation program is composed of *rules* that define how source model elements are matched and navigated to create and initialize the elements

of the target models. A model-transformation-oriented virtual machine has been defined and implemented to provide *execution support* for ATL while maintaining a certain level of flexibility. ATL is executable because a specific transformation from its metamodel to the virtual machine bytecode exists. Therefore, one can extend ATL by specifying the execution semantics of the new language features in terms of simple instructions: basic actions on models (elements creations and properties assignments).

Viatra (**Vi**sual **A**utomated model **Tra**nsformations) is a framework at the core of a transformation-based verification and validation environment [37]. It allows one to automatically check consistency, completeness, and dependability requirements of UML models and manipulate well-formed model-transformations formalized as graph transformations or Abstract State Machines. Graph transformation rules describe pre- and postconditions to the transformations and are guaranteed to be executable, which is a main conceptual difference with OCL. Graph transformation rules are assembled into complex model transformations by abstract state machine rules, which provide a set of commonly used imperative control structures with precise semantics. Viatra is not metamodeled using the standard MOF, and is transformation language is not QVT.

Tefkat is a response to the OMG's MOF QVT Request for Proposals [38]. It defines a mapping from a set of source metamodels to a set of target metamodels. A Tefkat transformation consists of rules, patterns, and templates. Rules contain a source term and a target term. Tefkat does not use explicit rule-calling; all rules fire independently from all others, however, rules can be loosely coupled. The Tefkat language is defined in terms of (E)MOF 2.0. However, the engine is implemented in terms of Ecore, the EMOF-like metamodel at the center of EMF, and acts as an Eclipse plug-in for EMF. The language is very similar to the Relations package of QVT, however, it is not strictly compliant.

3.2.1 Summary

Model-transformation works on *formalized* transformations definitions. These transformations may target model or metamodel, may conform partially to the standards, but still, the presented approaches rely on the knowledge of the rules' definitions (*i.e.*, left-hand side and right-hand side). As depicted in the motivation chapter of this thesis, the approaches enumerated from the state-of-practice *cannot make this assumption*, as they do not formalize, nor handle, nor know the definition of the transformations they manipulate. To apply these approaches to our running example, one must *(i)* write the actual transformations in any of the tools above, *(ii)* write the code that enables paradigm shift between automata and *(iii)* transition table, and *(iv)* the other way around. These approaches are really powerful as they allow advanced reasonings on manipulated transformations, but they are in an *all-or-nothing* approach where one has to formalize *already existing model and transformations* into these frameworks to benefit from their outcome, or the approaches are not applicable at all, and none of the potential outcome can benefit the developer. In the next sections, we will narrow the field of study to analyze compositions, successively at the model-level, then at the code-level.

3.3 Composition at the Model-level

The previous section targetted *transformations* in general, which is a superset of *compositions*. As many approaches of model composition use results and tools of model *transformation* tools, it seems interesting to analyze the state-of-the-art model compositions. We structure this study in two sections; this one focuses on model-composition; the next one will focus on compositions that occur at the code-level. There are two majors approaches to perform structural composition: the merge approach and the weaving approach. The merge approach is a symmetric operation (*i.e.*, that takes models of the same type) and can be an endogenous or exogeneous operation (*i.e.*, outputting a model of the same type or a different type respectively). The weaving approach is an asymmetric and endogenous operation that considers an already existing codebase which will be modified by weaving an aspect on it. The section depicts solutions that are part of one of these two approaches.

3.3.1 Criteria

— **Model-Driven:** A model-driven approach is a top-down approach. The model is already existing, and one must conform to this (meta-)model to make use of the solution and its gains. Thus, this criterion can take the values `Yes` or `No`.

— **Expressiveness:** The expressiveness describes *how* one uses one of the approaches. One can *use* a solution as is, like a tool; or the approach is *meant* to *create* a composition operator. The former is closer to the use of an external library, whereas the latter is closer to a framework approach. Thus, this criterion can take the values `USE` or `CREATE`.

— **Tunable composition operator:** Compositions can target various types of models, and use different techniques (*e.g.*, weaving aspects on a source-code). This composition can be extensible, or tunable, *e.g.*, specifying pre- or post-processing steps, or providing heuristics. This boolean criterion states if one can, or cannot, tune a given composition operator.

— **Open Composition of Composition Operator:** A composition of compositions is triggered when multiple transformations are to be applied to a codebase. This composition of compositions can be defined and not being replaceable, extensible, or even tunable. This boolean criterion states if one can, or cannot, specify her own way of composing multiple rules.

— **Entity- or diff-based:** Some solutions work at the entities-level, other work at the diff-level. The former may manipulate models such as class diagrams, sequence diagrams, or code, whereas the latter solutions work on a set or sequence of modifications. The former solutions merge entities, whereas the latter solutions reconcile modifications made on entities.

— **Conflict detection:** When multiple transformations are composed, *i.e.*, are applied together on a model, conflicts may occur. Conflicts happen in various shapes and forms, but this boolean criterion focuses on the capability of a given tool to check for conflicts in general, or not, and yield them to the final user.

3.3.2 Composition Approaches

AoURN (**A**spect-**o**riented **U**ser **R**equirements **N**otation) is a multi-views modeling framework that combines aspect-oriented, goal-oriented, and scenario-based modeling [39]. Whereas the standard URN (User Requirements Notation) combines goal-oriented and scenario-based models, AoURN extends it with aspects modeling, enabling clean reuse of crosscutting concerns. When multiple concerns are to be applied, the framework takes care of the interactions by applying concerns to the other concerns. Conflicts are formalized at the model-level inside a concern interaction graph.

Kermeta (**Ker**nel **Meta**modeling) is a *metamodeling language* that extends EMOF with an action language that allows the specification of behavioral semantics for metamodels [40]. The Kermeta language can be used for the definition of metamodels along with the implementation of their semantics, constraints, and transformations. It allows one to *develop* and execute three DSLs; each one mapped to a concern: abstract syntax in ECore, static semantic in OCL, and behavioral semantic in Kermeta. Each DSLs is projected as code, and composition occurs at this level. One can also *refine* the composition by adding a custom algorithm in the body of the operations defined in the composition metamodel. Kermeta is compatible with the ***E**clipse **M**odeling **F**ramework* (EMF) [41] and *does not provide any conflict handling* mechanisms.

Kompose is a *model-driven* tool based on Kermeta that composes homogeneous meta-*models* as long as they conform to the EMOF [42]. It performs a general-purpose match-and-merge composition where matched elements are merged, and unmatched ones are outputted as is. One can *specialize* the proposed generic composition operator to a particular modeling language described by a metamodel, considering a single operator that can be fine-tuned. Kompose formalizes *pre-* and *post-processing* steps, respectively to avoid issues and ensure coherence of the resulting metamodel, but it is left to the developer to *implement* them correctly.

Kertheme (**Ker**meta and **Theme**) [43] is an extension of Theme [44], an aspect-oriented approach built on *UML*. It proposes two types of compositions that one *cannot override* - one merging so-called "basic" concerns, the other weaving so-called "aspect" concerns into basic ones. As concerns consist of a pair of *models*: a class diagram as structural model and a sequence diagram as behavioral model ; each type of composition embed two composition operators. *No conflicts handling* are formalized nor proposed in Kertheme.

EML (**E**psilon **M**erging **L**anguage) is a hybrid, rule-based *language* for *statically* merging homogeneous or heterogeneous *models* [45]. It allows one to *develop* custom *merging algorithms* in *EML format*, working over models implemented in EMF. It also allows one to declare comparison operator (in ECL), and call arbitrarily complex external java methods. When merging models, the resulting model may not conform to the initial metamodel. Indeed, the merging process may have introduced elements that are not represented in the metamodel. Thus, EML *does*

not provide any conflict handling mechanisms, but an error-detection mechanism implemented as a metamodel conformance check.

CORE (**C**oncern-**O**riented **RE**use) aims for the reusability of development effort [46]. It proposes to modularize *models* of a system by domains of abstraction within units of reuse called "concerns". Inter- and intra-concerns reuse are possible. It allows one to model heterogeneous models by formalizing each one with *UML* models (*e.g.*, class diagram as structural model, sequence diagrams as behavioral model). The composition is *static* and is implemented with weaving techniques. The weaver is in charge of handling coherence between views of the system. Conflicts are *defined by the developer* in an aspect conflict resolution model, to be automatically fixed when occurring at the user-level.

GeKo (Generic weaving with Kermeta) is a *static* generic *model* weaver that can be used to weave any kind of models that conform to EMOF [47], [48]. In GeKo, an aspect is defined as a pair of *models*: a left-hand side for the specification of where to "cut", and a right-hand side that represents the expected elements at the "join" point. GeKo weaving approach may extend the matched behavior, replace it with a new behavior, or remove it entirely. GeKo defines a two-phased weaving process: a generic detection and a generic composition, both *non-user-replaceable* nor tunable.

MATA (**M**odeling **A**spects using a **T**ransformation **A**pproach) is an Aspect-oriented Modelling approach for aspect weaving in *UML models* [49]. MATA is really similar to GeKo but uses graph theory as a backend to analyze *interactions* between woven aspects and base models. The resulting graph is outputted as a UML model. A MATA model is made of left-hand side, right-hand side transformation-rules which are defined using UML stereotypes to manipulate model elements. Graph theory and tools allow MATA to perform conflict detections such as aspect and feature interactions.

Models@runtime combines model-driven and aspect-oriented approaches and techniques to tame the complexity of developing and maintaining dynamically adaptive software systems [50]. It is made of a complex events processor, reacting to sensors' values, triggering aspects composition on the system, conforming to a predefined goal. This reasoning can be tuned at the developer-level as Models@Runtime reasonings formalisms is open by design. Based on aspect oriented approaches' backend that rely on graph theory, Models@Runtime performs conflict detections between aspects.

ModelBus defines a merge operator to reconcile concurrent *modifications* made on a *model* in a collaborative context [51]. It focuses on its scaling capabilities in a collaborative context. For every input model, ModelBus computes the modifications, *i.e.*, the delta, that were made between the base model and the input one. As it performs such computation on every input models, it then *reconciles* all the modifications and *check for conflicts* before applying them to the input model. The definition of conflict is domain-independent; thus, they are consistency checks,

e.g., updating a deleted element, modifying the same property with two different values. The reconciliation step *cannot be refined* nor tuned.

3.3.3 Summary

The presented approaches of model composition are all model-driven, and half of them allow the developer to create her composition operator; the other half allows direct use by developers and does not allow their composition operator to be overridden (see TAB. 3.1). Moreover, none of the approaches depicted in this section consider the composition of compositions operators as a main issue, which mainly explain the next point. No support for automated conflict detection, nor general checking capabilities have been identified in these approaches. Conflicts detection and their respective fixes are sometimes acknowledged as some approaches formalize interface to yield, capture and fix conflicts, yet it is still the developer's responsibility to actually detect, yield, and fix them, manually. In the next section, we will analyze compositions that occur on the code-level with more tool-oriented approaches.

Table 3.1 – Summary of the approaches of compositions at the model-level

Approach	Expressiveness	Tunable Composition Operator	Open Composition of Composition Operator	Entity- or Diff-based	Conflict detection
Kompose	Creation	Yes	No	Entity	No
Kermeta	Creation	Yes	No	Entity	No
Kertheme	Usage	No	No	Entity	No
AoURN	Usage	No	No	Entity	Yes
EML	Creation	Yes	No	Entity	No
CORE	Usage Creation	No	No	Entity	No
GeKo	Usage	No	No	Entity	No
MATA	Usage	No	No	Entity	Yes
MATA	Creation	Yes	No	Entity	Yes
ModelBus	Usage	No	No	Diff	Yes

3.4 Composition at the Code-level

3.4.1 Introduction

Yet, a subset of model-transformations may be of interest and have to be part of this state-of-the-art chapter: code transformation. A code is a model the same way a class diagram is a UML model: it is a way to define the structure and behavior of an artifact. This subset of software-composition contains tool-oriented approaches more than reasoning-oriented ones; thus, this section lists code compositions tools, with the particularity that composition occurs at the code-level. Some tools described in this section may rely on models to know how to apply their composition operators, nevertheless their compositions occur at the code level, and target code.

3.4.2 Criteria

— **Build to match domain specific requirements:** This criteria states if a tool has been developped to match domain-specific requirements, or as a generic tool to perform many operations targeting various purposes. A tool may have been developed as a general purpose tool, or as a tailored tool for a specific requirement of their application domain. For instance, a tool may be developped to rewrite java code, it is its purpose and it has been built to do that, **but** *why* is someone rewriting a java code is out of its scope. The rewriting abstractions, formalisms, operations, and intent of the generic tool are dissociated from its final use.

— **Used in industry:** Tools may be used as proof-of-concept, in academia, as a rasearch tool, in an industrial context (*e.g.*, library, API), etc. This boolean criteria specifies if a given tool is used in an industrial context or not. A tool is used in an industrial context if it has been used for a long time, in a public project.

— **Expressiveness:** The expressiveness describes *how* one uses one of the tools. One can *use* a solution as is, or the tool is *meant* to *create* a transformations. The former is closer to the use of an external library , whereas the latter is closer to a framework approach. Thus, this criterion can take the values `USE` or `CREATE`.

— **Target:** In this section, tools may target source-code or text, but transformations in general can target models also. This criterion specifies the amount of structure a given tool has on its inputs. A tool can take *any* `textual` file as input, or a source code of a given language, or a model. Therefore, it can take a values between `Text`, `Source code`, `Model`.

— **Customizable composition operator:** This composition can be defined, *i.e.*, hard-coded, and not being replaceable, nor extensible. Other operators may be hard-coded but can be customizable by allowing the user to specify partial order between transformations. Thus, this criteria defines the level of customization as `CLOSED`, *i.e.*, closed hard-coded and non customizable operator, `CUSTOMIZABLE` if one cannot fully override the operator but can change some of its parameters, and `OPEN` if one can fully swap the composition operator.

— **Open composition of composition operators:** A composition of compositions is triggered when multiple transformations are to be applied on a code. This composition of compositions can be defined and not being replaceable, extensible, or even tunable. This boolean criterion states if one can, or cannot, specify her own way of composing multiple rules.

— **Composition order:** When multiple transformations are defined and composed to-be-applied in a code base, an order in which they are applied is defined. This order can be partial or total, but this criterion focuses only on two things: Does the order in which they are applied follow a domain specific expertise ? If so, the criterion is evaluated to `MANUAL`, `ARBITRARY` otherwise. Then, it specifies the actual order, *e.g.*, `SEQUENTIAL`, `PARALLEL`.

— **Use of diffs:** Code transformations tools may use diffs for various purposes, *e.g.*, as basic model, to solve conflicts, etc. This boolean criterion states if a given tool makes use of diff or not.

— **Static/Dynamic composition:** Applying a rule or a set of rules can be done statically (*i.e.*, at compile-time) or dynamically (*i.e.*, at runtime). Some tools allow both and thus this criterion can take the values between `STATIC`, `DYNAMIC`, `BOTH`.

— **Conflict detection:** When multiple transformations are composed, *i.e.*, are applied together on a code base, conflicts may occur. Conflicts happen in various shapes and forms but this boolean criterion focuses on the capability of a given tool to check for conflicts in general, or not, and yield them to the final user.

3.4.3 Tools considered

SPOON - is a `Java` tool that allows one to transform any `Java` code. It works at the `AST`-level [52]. One specifies transformations inside `SPOON` processors, implemented as a `Java` class extending the `SPOON` framework, that will work on code elements specified in the `SPOON` framework (*e.g.*, `CtClass`, `CtMethod`, `CtStatement`). Each `SPOON` rule must specify 3 things: *(i)* the `AST` elements it captures (*e.g.*, `CtClass`), *(ii)* if it has to be applied on a given element, *(iii)* the method to actually apply the rule and transform the `AST`.

— The element captured is specified by `Java` generic mechanism in the signature of the extension of the `SPOON` class.

— The *applicability* is assessed via a $isToBeProcess(M) \mapsto Boolean$ method.

— The transformation is implemented in an $apply(M) \mapsto void$ java method that will use `SPOON` factories and abstraction to modify the element, thus the `AST`.

The `AST` is updated and passed between `SPOON` processors that are executed in sequence as defined in the order they are specified in a configuration file. Updates of the `AST` are black-boxed inside `SPOON` processors, without use of diffs. No detection of conflicts between processors is available.

Coccinnelle - is a tool that enables transformations of a codebase by specifying *semantic patches* [53]. These *patches* are written in SmPL (Semantic Patch Lan-

guage) and are used in the context of the development of the Linux Kernel. Coccinnelle was developped to automate evolutions and fixes of common issues in the Linux kernel. A semantic patch is a textual file, made of potentially different cases. A case is made of *(i)* context description, *i.e.*, the context to be matched in a file to trigger the rule, *(ii)* addition and deletion, respectively in the form of lines preceded by ++ and - symbols, *(iii)* potential guards that restrict the context in which the patch has to be triggered. Coccinnelle defines 54 semantic patches for the Linux Kernel that will be applied in sequence to check a given version of the Linux Kernel but more rules can be addded. Each semantic patch will be applied in sequence, in alpha-numerical order, on the Linux kernel and the result is passed to the next one. No detection of conflicts between semantic patches is done, statically or dynamically.

EJB (**E**nterprise **J**ava**B**eans) is a server-side software component that encapsulates business logic of an application and allows modular construction of enterprise software in `Java` by reusing concerns [54]. Such software addresses the same types of problem, and solutions to these problems are often repeatedly re-implemented by programmers, thus EJB is intended to handle such common concerns as persistence, transactional integrity and security in a standard way, leaving programmers free to concentrate on the particular parts of the enterprise software at hand. The EJB specification was originally developed in 1997 by IBM and later adopted by Sun Microsystems, and enhanced under the Java Community Process. A set of EJB interceptors, allowing composition of these reusable concerns, can be added or removed *statically*. The order in which they are invoked is given by the order in which they are *declared*, or can be overriden by the *developper* in a configuration file.

AspectJ AspectJ is a *language*, supersetting `Java`, that allows one to *implement* crosscutting concerns through *pointcuts* (collections of principle points in the execution of a program), and *advices* (method-like structures attached to pointcuts) [15]. When multiple advices have to be applied at a joint point, composition issue is solved thanks to *precedence rules* defined by the developper. AspectJ is *used in industry* via the SPRING framework, widely used in many projects.

JAC - (**J**ava **A**spect **C**omponents) is a general-purpose `Java` framework for dynamic aspect-oriented programming [55]. It allows one to specify aspects in `Java` to be woven on a `Java` object at runtime, along with specifying their compositions, also in `Java`. The weaving can be added as wrapping of an existing object, to enhance an existing object's role, or to handle exceptions. When multiple aspects have to be woven in an application, different composition issues may arise: *(i)* compatibility of an aspect with an application, *(ii)* inter-aspect compatibility, *(iii)* inter-aspect dependance, *(iv)* aspect redundancy, and *(v)* aspect-ordering issues. In the `JAC` framework, the core idea is to provide abstractions and to formalize composition of these weavers as an external, simple, and easily reusable artefact, in an AOP fashion. A *controller is in charge of the aspect composition*, which should be developped by a human that knows all the aspects of the application. "*Indeed, JAC programmers can cleanly describe how the composition of the aspects will be*

hanlded by the application within a well-bounded part of the program called a wrapping controller."

Python 2to3 - is a tool that allows one to transform a `Python 2.x` codebase into a `Python 3.x` compliant codebase [56]. One may just run a `Python 2` code under `Python 3` and fix each problem as it turns up, but `2to3` does *most* of these changes automatically. The tool `2to3` is made up of a core, named `lib2to3` that allows one to refactor `Python` code by analyzing the code and building up a parse tree from it. This core acts as a framework to develop so-called "*fixers*" that will perform specific refactorings, such as changing a `print` statement into a `print()` function call directly inside the python source-code. The set of standard fixers consists of 52 fixers that allow `2to3` to convert `Python 2` code to `Python 3` code. This set is opened, its rules are executed in alpha-numerical order in an arbitray fashion. No detection of conflicts is available between multiple fixers.

Git Merge is an operation available in the source code versioning tool `Git`. When developing software in a collaborative environment, different developers may concurrently modify the same portion of the codebase [57]–[59]. Each of them will, usually, develop on a dedicated development branch, then these branches will be merged, and the two versions of the codebase will also be merged. Actually, the codebase itself is not merged, but their modifications are. Git merge is a diff based composition operator that works on modifications sets and outputs a modification set. One cannot interfere with the composition process, even if the Git Merge can be replaced by another tool in the git tool-suite. Conflict detection is done at the textual level, regardless of the language used to develop the system, which is its strongest pros (*i.e.*, language independent) and cons (*i.e.*, poor precision and recall).

Patch is a tool that allows one to *modify* a textual codebase given a textual *patch* [60], [61]. A patch targets a specific line of code and provides the context to be captured (lines preceded by "–" symbols), and by which content to replace it (lines preceded by "++" symbols). No other actions (*e.g.*, update) are available nor can be added. The order of the patches is defined by the order they are given as input. No detection of conflicts is available, before appliying patches; but a warning will be yielded if a patch's context can no longer be matched, *i.e.*, another patch that has been applied before has changed the context of a future patch.

C preprocessor is part of the tools of the C programming language [62]–[64]. It defines two directives that we consider as composition operators.

— The first directive is `#include`, and tells the preprocessor to replace the directive's instruction by the contents of another file, directy at the source-code level. This allows one to "compose" a code base by assembling arbitrary complex parts of source code. The semantic of this directive cannot be overriden at the developer-level. As the directive is replace at-compile time, as the preprocessor finds it, multiple includes will lead to sequential replacements; and nested includes will be processed recursively, as they are found.

— The second directive `IfNDef` is a conditionnal `if` statement at the preprocessor level. Combined with the previously introduced `include` directive, and along with the `define` directive, it allows development teams to handle variability directly at the source-code level. For instance, **if** a global preprocessing variable has been **defined**, process (*i.e.*, add at the current position) the following statements (that may again conditionnally define variables). The entire Linux kernel variability is handled via this mechanism. It is fast and allows entire portions of the code base to not be considered at compile-time. The semantic of `ifndef` cannot be overriden at the developer-level. As the `include` directive, this conditional directive is handled as is, and in place.

3.4.4 Summary

Tool	Express.	Target	Built w/ domain specific req.	Used in industry	Custom. Compo. Operator	Open compo. of compo. operators	Composition order	Diffs?	Static Dynamic Compos.	Conflicts detection
SPOON	Creation	Source code	No	No	Closed	Closed	Arbitrary sequential	No	Static	No
Coccinnelle	Usage	Text	Yes	Yes	Closed	Closed	Arbitrary sequential	No	Static	No
JAC	Creation	Source code	No	No	Custom.	Opened	Manual sequential	No	Both	Yes
2to3	Usage	Source code	Yes	No	Custom.	Closed	Arbitrary sequential	No	Static	No
EJB	Creation	Source code	No	Yes	Custom.	Opened (partial)	Manual sequential	No	Static	No
AspectJ	Creation	Source code	No	Yes	Custom.	Opened	Manual sequential	No	Static	Yes
Git Merge	Usage	Text	Yes	Yes	Opened	Opened	Arbitrary sequential	Yes	Static	Yes
Patch	Usage	Text	Yes	Yes	Closed	Closed	Arbitrary sequential	Yes	Static	No
C preprocessor	Usage	Source code	Yes	Yes	Closed	Closed	Arbitrary sequential	No	Static	No

Table 3.2 – Summary of the approaches composing at the code-level

3.5 Conclusion

In this chapter we studied the state-of-the-art approaches regarding model compositions, model and code transformations. Model compositions were not compatible with our context as their *all-or-nothing* approach prevents any already existing operator to benefit from their outcome. Whereas model transformations enable powerful reasonings on conflict detection, and order of transformations, their hypotheses of a *white-box* well-formalized transformation doesn't match the context of this thesis. Finally, the code transformation approaches are tailored for a more or less generic industrial usage, where one cannot often specify her own composition operators. These approaches, closer to a tooling approach than a reasoning one, also offer poor support for conflicts detection and reasoning capabilities in general. They target source-code or any textual artefact only, and the order of the composition is defined manually in the best case, or cannot be overriden otherwise. Now that the state-of-the-art approaches have been analyzed, we will map in the next chapter the state-of-practise depicted in Chapter 2 with the characteristics described in the current chapter to define the challenges of this thesis.

CHAPTER 4

Background and Challenges

"A convincing demonstration of correctness being impossible as long as the mechanism is regarded as a black box, our only hope lies in not regarding the mechanism as a black box."

Dijkstra [65]

Content

4.1 Introduction

In the previous chapters, we depicted real-life composition use-cases, their characteristics and commonalities, before depicting how software composition is addressed in the literature. We outlined that on the one hand, state-of-the-art compositions make strong assumptions on the composed elements, but on the other hand, they ensure powerful and useful properties. This chapter links the two previous chapters as it maps the use-cases described in 2 and the approaches depicted in the previous state-of-the-art chapter. This chapter depicts such a white-box approach, and presents the issues that arise when such automated transformations are, in fact, black-boxes: one cannot know what it captures or produces. It depicts how the assessment of white-box properties is not feasible when facing black-boxes. Reasoning on such black-box artifacts and ensuring guarantees in a domain-independent way are challenges that we will outline.

4.2 White box rewriting rules are not enough

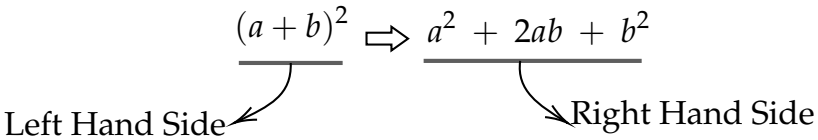
Models of any kind can be transformed thanks to automated transformations. Such mechanisms can be put in place to fix common mistakes, perform custom operations at large scale, to avoid manual intervention and human effort. Automated fix of common mistakes, evolutions of API, modifications of large UML models, are a few use-cases where transformations are automatically applied to an artifact as described in 3. Thus, one can regard these rules as off-the-shelf functions that one can use to transform a specific portion of a model.

For instance, UML class diagrams can be transformed to match a metamodel update or to adapt to a new requirement; and the Linux kernel embeds rewriting rules (named patches) that automatically fix common mistakes or operate at scale by aligning the whole codebase to a new version of the kernel API. These rules can operate on various types of models (*e.g.*, UML models, code), and are implemented via different techniques and tools. Issues may arise when several of these rules are applied on the same model, *e.g.*, a rule can delete what would have been captured by another, or two rules can modify the same model element in two different ways.

As described in the previous chapter, these issues are not new and were addressed in the literature, mainly by providing a formalism for these rewriting rules that enable reasoning techniques, able to detect overlapping or conflicting rules. These transformations are formalized in the literature as white-boxes, often as term-rewritings rules. What a transformation captures as input and produces as output is known and enables reasonings capabilities, avoiding conflicting rules and making the whole set of transformations safer.

In a "*white-box approach*," rewriting rules are implemented following a known, precise, and sound formalism, that one can check beforehand for overlaps or conflicts. Term rewriting is a paradigm where rewriting rules are formalized as equations: a left-hand side (*i.e.*, a *term*) that describes the context to-be-matched for the rule to be applied, and a right-hand side (*i.e.*, a *term*) that describes the matched context once the rule has been applied [66], [67]. When using basic high-school algebra, one is working with such rewriting rules. Examples such as the

equation below are formalized as term rewritings, via what we call a white-boxes approach.



Terms rewriting can be (excessively) simplified to *rewriting rules* that *substitute* the left *term* (*LHS*) by the right term (*RHS*) if the left term is *matched* in a given *context*. Even with such simple formalism, term rewritings enable useful reasoning techniques to compose multiple rules safely. Term rewritings paradigm can be applied to various applications from software transformation, to type-checking or even interpretation of programs [68]–[70]. These approaches also include pre-conditions to allow conditional application and post-conditions to validate the resulting output.

4.2.1 Optimizing Automata With Rewriting Rules

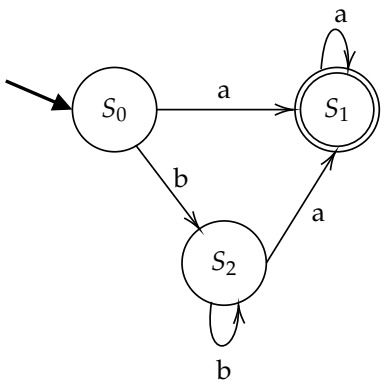


Fig. 4.1 – Initial term t, automaton result of a merge process.

Let us take the automaton depicted in FIG. 4.1 and assume that this automaton is the result of a merge process and that operators of *minimization* and *determinization* are applied to it. Each of these two operators can be implemented as a set of rewriting rules. Thus, this example considers *automata* as *terms*, and the automaton depicted in FIG. 4.1 is the initial term t that will be modified by rewriting rules.

The example rewriting systems is composed of two rules. The first rule is depicted in FIG. 4.2. It locally detects equivalent states to lower the number of states reachable and avoiding useless execution paths. The second one is depicted in FIG. 4.3. It deletes unreachable states that have no incoming/outcoming edges from/to other states (*i.e.*, dangling states, states that are not linked to other states) to avoid combinatorial explosion during the merge process. Of course, these rules are partial and naive implementation of the minimization of automaton but are enough to exemplify the rewriting approach.

Applying both of these rules on the term t can be done regardless of which order the rules are applied. One can quickly note that rule R_2 will never modify

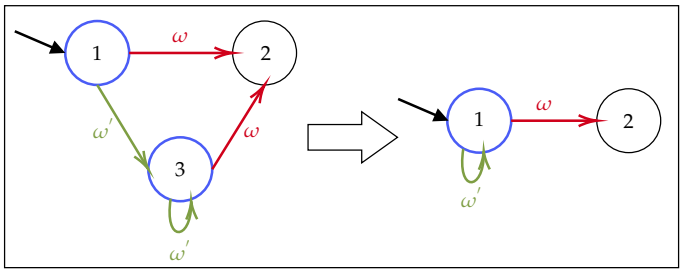


Fig. 4.2 – Example of rule R_0 merging equivalent states

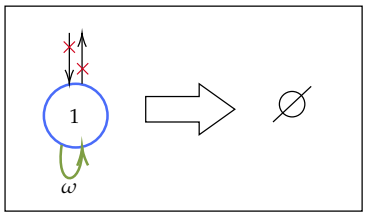


Fig. 4.3 – Example of rule R_2, removing dangling states

the term t as its left-hand-side is not matched in it. Thus, $R_0(R_2(t)) \equiv R_2(R_0(t))$ and both sequences lead to the same result.

Now, let us consider the rewriting system composed of t (FIG. 4.1), R_1 (FIG. 4.4), and R_2 (FIG. 4.3). Considering these transformations as white-boxes by design, tooling can be developed to analyze them and ensure their *safe* composition automatically.

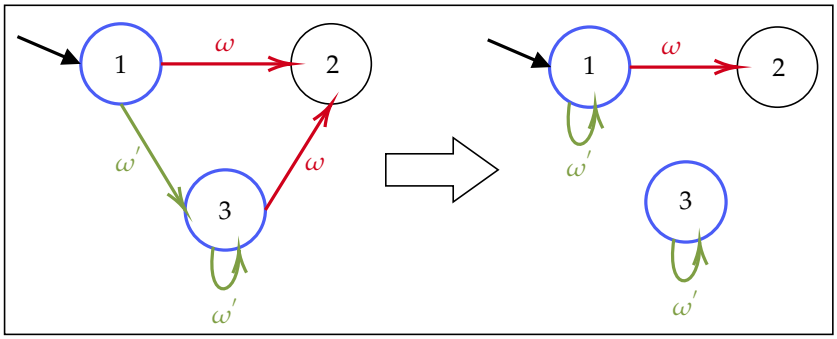


Fig. 4.4 – R_1, alternative version of R_0

4.2.1.1 Order-related issues.

Considering this rewriting system composed of t, R_1, and R_2, state-of-the-art tools will detect an issue. Indeed, depending on the application order, the result will differ. In a concrete context, such transformations may target code and have a significant impact on the final system. Thus, having different results given the order of the execution without the user being aware of it is not conceivable. The rules R_1 and R_2 are not safe to be applied as the rule R_1 produces as RHS a term partially matched by the LHS of R_2. These two rules can be applied following two sequences, $R_1(R_2(t))$ or $R_2(R_1(t))$.

Sequence $R_1(R_2(t))$**.** Applying R_2 first does not modify t since its left-hand-side (LHS) is not matched, *i.e.*, there is no dangling state in t. Then, applying R_1 to $R_2(t)$ effectively modifies t since its LHS is matched. Thus the term t is modified with the right-hand-side (RHS) of R_1 to obtain a new term t_{21} depicted in FIG. 4.5. However, in the final term t_{21}, the rule R_2 could still have been applied because now its LHS matches: there is still a dangling state, even if R_2 was put in the sequence of rules to be applied. This implies that R_2 has not been applied

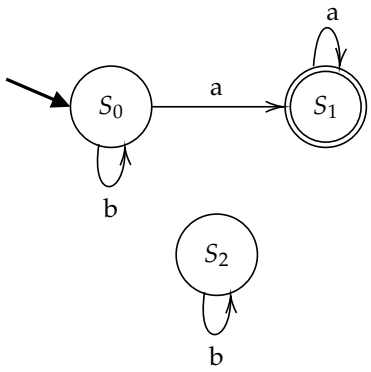


Fig. 4.5 – $t_{21} = (R_1(R_2(t)))$

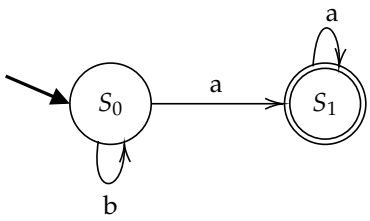


Fig. 4.6 – $t_{12} = (R_2(R_1(t)))$

correctly in the whole sequence and that applying R_2 last would have been *the right way* to go.

Sequence $R_2(R_1(t))$**.** Applying R_1 first, effectively modifies t since its left-hand-side is matched. Then, applying R_2 to $R_1(t)$ modifies the automaton as there is a dangling state (*i.e.*, *S*2) that can be deleted. Thus the term t is modified to obtain a new term t_{12} depicted in FIG. 4.6. The final term, t_{12}, does not have any dangling states or redundant paths in it. Thus, applying the whole sequence again, on t_{12} is not necessary as it will yield the same result. This sequence is *the good way to go* as every rule has done all it was supposed to do.

This is a toy example with the smallest set of rules possible. As the size of the rule-set grows, computing all sequences to find the *good* one is not feasible. Toolings working in a white-box context would detect and avoid such issues. By analyzing the definitions of the rules (hence the white-box approach), it will either sort the sequence of rules to be applied to avoid a sequence or re-apply rules if needed (*e.g.*, R_2).

4.2.1.2 Non order-related issues.

Now, we change the rules-set to R_0 (FIG. 4.2) and R_3 (FIG. 4.7). The rule R_0 still avoids a path containing S_2, whereas R_3 avoids a self-looping edge by moving the self-looping edge of S_2 into S_0, modifying S_2. These two rules (R_0 and

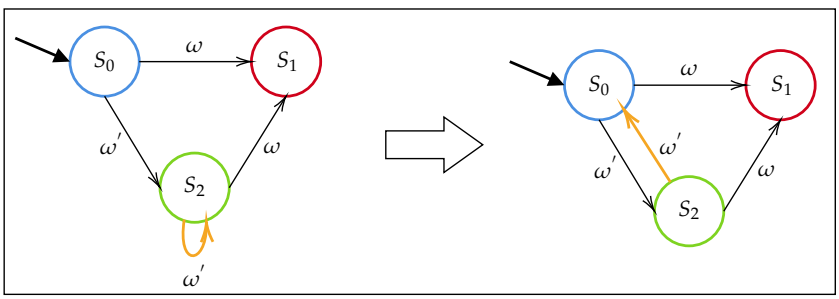


Fig. 4.7 – Rule R_3 redirecting looping edge ω' from S_2 to S_0

R_3) have non-empty intersections between their respective *LHS*, and produce a different output (*RHS*): one deletes S_2, the other uses it as reference. This non-empty intersection implies that the order in which they will be applied matters,

and that each sequence will yield a different result. It is still an ordering issue, but it differs from the previous example. In the previous example, one rule produced an output matched by the other one; thus, a *proper* sequence exists (*i.e.*, a sequence where all rules applicable were applied). However, in the current rewriting system (t, R_0, and R_3), there is no *good* or *bad* sequence. These two rules are incompatible in this context, and both sequences ($R_0(R_3(t))$ and $R_3(R_0(t))$) produce two different results.

4.2.2 Properties With Rewriting Systems

State-of-the-art techniques would leverage such formalism, to perform compatibility or ordering checks on these off-the-shelf rewriting rules [66], [71]. Such reasoning is known as a confluence assessment between term rewriting rules and is possible only because rules are formalized as white-boxes term rewritings, *i.e.*, their left- and right-hand sides are known and actionable.

In abstract rewriting systems vocabulary, the depicted rewriting system is not *confluent* given this context and sequences of rules that lead to automata described in FIG. 4.6 and FIG. 4.5 form a critical pair.

Confluence. Confluence property describes the order-free of rewriting rules transformations [72]. Confluence states that no matter how two sequences of rewriting rules diverge at a given point, the paths built by successive application of rewriting rules are joining at some point later. A strongly confluent rewriting system implies that the order in which one applied the rules does not matter, *i.e.*, it will eventually lead to the same result. One can see how this property is useful in a software engineering scenario: there is no bad sequence of application so any order will be fine, and rules can be applied without extra computation. For example, the rewriting systems described previously ((R_1, R_2) and (R_0, R_3)) are not strongly confluent, but (R_0, R_2) is.

Termination. Termination of rewriting systems is a property that one can assess on such white-box systems. It is another critical property that can be assessed in term rewriting systems [73]. This property guarantees that the system will *eventually* lead to a term that cannot be reduced further. Such tools' algorithms rely on three majors aspects: *(i)* simplification orders [74], *(ii)* dependency pairs [75], and *(iii)* the size-change principle [76].

Critical pairs. Analysis of critical pairs highlights the involved rules' sequences and allows human-in-the-loop analysis. In the context of a huge rules-set, critical pair analysis allows to reduce the search space by providing only the sequences of rules application that lead to a non-confluent system. This enables developers to either refine the rules to avoid the issue or select which rule to apply in the actual context, reducing the conflict space and avoiding manual checks or arbitrary choices.

4.2.3 Challenges for Software Developers to Use White-box Approaches

Let us consider the context of a software developer in a company, that needs to develop a new composition operator. She faces the choice of developing it using mainstream languages, as its company has always done, or using model-driven tools using white-box transformations as described quickly in this chapter.

The former option has the benefit of:

— no training is needed since she already has the set of skills required to develop using mainstream development languages,

— plenty of qualified workforce and support available,

— the two last points allow teams to ship software in a time-to-market frame that keeps getting shorter,

— no need for higher level approvals since the company always worked that way,

— and a known schedule, since the teams know common mistakes, common issues and traps, and regular troubleshootings and solutions, reinforcing the hierarchy trust in this option.

The latter option of developing the new feature using MDE tools and approaches has the following drawbacks:

— training is needed since the developer is more likely to not be used to MDE tools,

— compatibility with legacy systems and its assessment may be a difficult and tedious task. Why developing a new feature in a way that it will be isolated from the rest of our codebase? Is there a bridge to/from this MDE and from/to our ecosystem?

— assessing *what* can (or can not) be developped using MDE tools may be a difficult task that requires time, human labor, and skills,

— assessing the adequation of such tools with software requirements such as the execution-time is not an easy task,

— hierarchy has to be convinced since they may not see the benefits of such an MDE tool. Highlighting the anticipated benefits, weighing the gains, efforts, and assessing risks is a hard task, especially for a developer with little experience in these tools.

— assessing the dependency, viability, and sustainability of such tools, as any other tools, is hard.

Using a tool that she has not used before is a hard choice and definitely not the easy path. As any other tools, model-driven ones are difficult to be chosen by teams of developers if they do not have the knowledge and skills needed to master them.

Creators of tools used by skilled developers, such as the creators of Coccinnelle, acknowledge these issues. Convincing Linux developers to use tools such as Coccinnelle, shaped the whole way Coccinnelle was built and formalized. Coccinnelle semantic patches are written as code fragment, developed in C and in the C-ecosystem because Linux developers were used to it **lawall-talk**. Coccinelle's

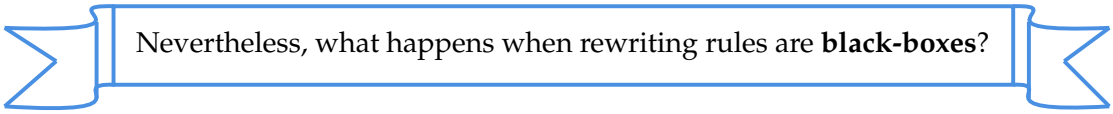
creators listed *ease of use* and *preservation of coding style* as the first two requirements for the Coccinelle toolsuite to convince Linux developers to use it [77].

Of course, MDE tools have great benefits but they are hard to grasp and assessed for a non-specialist, whereas both benefits and drawbacks of the mainstream development are known and mastered by most of the developers as this approach is more likely to be part of their culture and education curriculum. It is not surprising that, both from a development and project-management point-of-views, the choices tend to lead to a well-known, well-handled comfort zone. As described in the previous chapters and as formalized in the next section, this comfort-zone solution comes without well-formalized white-box artefacts, but with black-box ones. The next section depicts the challenges we outlined to work with such black-boxes.

4.3 Black-box Rewriting Rules

4.3.1 Composition in a Black-Box Context

The previous section depicted how term rewritings using a white-box approach enable the assessment of properties such as termination and confluence and allow one to highlight rules that will lead to different results. By making the strong assumption that one formalized the rules as *white-boxes* term rewriting, such formalism allowed powerful and useful reasonings.



Development contexts may prevent the rules from being white-boxes. As described in Chap. 2, they can be part of legacy systems, can be too complex to be analyzed such as reflexive Java code, and as a consequence are considered as black-boxes. In actual scenarios where rules are *not* formalized in a white-box approach, developers still need such assessment and reasonings. If we limit the visibility of a rewriting rules' definition as a black-box, depicted in FIG. 4.8, how does it change the guarantees and properties we can ensure?

Fig. 4.8 – Rewriting rule as a black-box

In a black-box context, a rewriting rule ρ can be seen as an endogenous function φ that operates over a model M and yields a model M. Each rule comes with a postcondition checker χ that states if it was applied correctly. It takes the initial and final models and states if all needed modifications were made correctly. We do not have more information at our disposal. Thus, as depicted in Eq. 4.1, rewriting rules are a pair: one actual rewriting function, and a check function.

For any input model, checking χ over the application of φ holds, *as we considered bug-free implementations*.

$$\begin{aligned} &\text{Let } \rho = (\varphi, \chi) \in (\Phi \times X) = P, \quad (\varphi : M \to M) \in \Phi \\ &\quad \chi : M \times M \to \mathbb{B} \in X, \qquad \forall m \in M,\ \chi(m, \varphi(m)) \\ &\quad \varphi(m) = m \text{ if } \varphi \text{ is not applicable to } m \end{aligned} \tag{4.1}$$

Working with such definitions shape the whole way rules may be composed together. Analyzing a black-box rule in itself and ensuring properties on it, is trivial. Issues arise when black-box rules are *composed* together when several black-box rules must be applied.

4.3.2 Classical Composition Operator *apply*

Composing rules can be made using the classical $\circ$ operator that chains the applications of the rules. This approach takes rules and passes the output of the first to be applied, as input to the second one. They are daisy-chained at the execution time (FIG. 4.9).

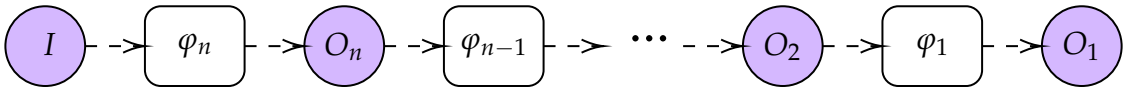


Fig. 4.9 – Daisy-chaining (sharing inputs - I, and outputs - O) of black-box rules (r)

For a set of rules $[\rho_1, \ldots, \rho_n]$ to be applied, they are consumed in *sequence*. This leads to a situation where only the last postcondition (χ_1) can be ensured in the resulting program, by construction. This rule composition operator, named $apply$, models the classical behavior for rule composition in the state of practice. Note that since $\circ$ is a *sequential* operator (as denoted by $<$), it unfolds into $\varphi_1(\varphi_2(...(\varphi_n(m))))$. Thus, only the last postcondition *to be applied*, *i.e.*, χ_1, holds.

$$\begin{aligned} &apply : M \times R^n_< \to M \\ &\quad m, [\rho_1, \ldots, \rho_n] \mapsto \text{Let } m_{2.n} = (\overset{n}{\underset{i=2}{\circ}} \varphi_i)(m),\ m' = \varphi_1(m_{2.n}), \quad \chi_1(m_{2.n}, m') \end{aligned} \tag{4.2}$$

The main issue with this approach is the impact of overlapping rules on the yielded model. This daisy-chained composition, which is mandatory given the provided formalism, has multiple drawbacks:

— It is order sensitive since the index of a given operator in the sequence may have an impact on the final output O,

— Considering large software systems where separation of concerns matters, each rewriting rules is defined independently. As a consequence, if two rules do not commute ($r_1(r_2(p)) \neq r_2(r_1(p))$), it is up to the developer to *(i)* identify that *these* rules are conflicting inside the whole rule sequence, and *(ii)* fix the rules or the application sequence to yield the expected result.

— One can only ensure that the last rules R_n has done all its work since it is the last one to-be-applied. Generally speaking, operators with an index n, can output a program that would have been modified by an operator of index m, where $m < n$. Of course, we assume that an operator is bug-free and deterministic.

In a white-box context, tools would have been done these reasonings automatically, based on the transformations' definitions. *We cannot assess rules confluence, nor outline the critical pairs to the user using the $\circ$ composition operator.*

4.3.3 Parallel Composition Operator ||

Black-box rewriting rules can be applied in parallel. Each rule will take the same input, and each one will yield an output model as described in FIG. 4.10.

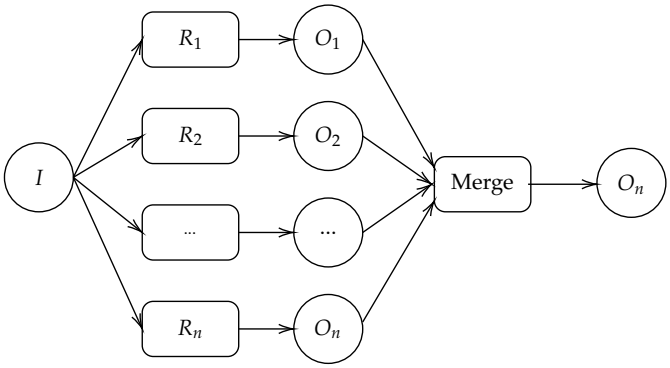


Fig. 4.10 – Parallel composition of rules with a merge operation

Thus, the parallelization approach involves a **domain-dependent** *merge* of models at some point. Then, all checker χ can be applied and assessed on the merged model O_n.

4.4 Challenges of Ensuring Properties in a Black-box Context

4.4.1 Introduction

This chapter started by depicting state-of-the-art approaches of reasonings on transformations in a broad meaning. It depicted how these approaches enabled powerful and useful reasonings based on the transformations' definitions. Then, in the next section, we depicted how the context changed when the considered transformations were black-boxes. We depicted the operators that can be used in such a context to perform composition and outlined that none of the properties from the state-of-the-art can be assessed. In this section, we outline the challenges this thesis addresses in the previously described black-box context.

4.4.2 Challenge C.1 - White-box properties in a Black-box Context

Confluence. Confluence assessment tools require to formalize rewriting rules as term rewritings where each side is accessible and "readable." This formalization is mandatory for the whole assessment process. In the context of a black-box rewriting rule where its definition is not known nor formalized, such property *cannot be assessed*.

Termination. Termination of rewriting systems is another critical property that can be assessed automatically. Termination tools' algorithms rely on three major aspects: *(i)* simplification orders, *(ii)* dependency pairs, and *(iii)* the size-change principle. The *first* one requires so-called rewrite-relation that *needs access to the definition* of the rewriting rule ; the *second* one identifies dependencies between left-hand and right-hand sides of rewriting rules, thus *needs access to the definition* of the rewriting rule, and the *third* one aims at program termination and decidability and works on programs' function calls and is therefore *out-of-topic*. In a context of a *black-box* rewriting rule where its definition is not known nor formalized, **none of these levers can be applied**; therefore, the termination property *cannot be assessed*.

In scenarios where rules are *not* formalized in a white-box approach, developers still need such assessment and reasonings. When working with black-boxes, the need for automated checks for order-issue, automated extraction of conflicting rewriting rules is critical as they cannot be opened or easily analyzed. Their black-box nature exacerbates the need for such automated reasonings. Development teams working in such a context still need *somehow* similar guarantees. We **cannot** ensure confluence nor termination as described and assessed in the state-of-the-art, but still need *something close*. Thus, we outline the following challenges to propose similar properties.

Confluence-like - How to assess the order-free of a rewriting system made of black-box rules?

Conflict - How to detect rules conflicts in a black-box context?

Termination-like - How to assess the termination of rewriting systems in a black-box context?

Critical pairs - How to reduce the conflict space, extracting the non-confluent rules to yield critical pairs to the final user, in a black-box context?

4.4.3 Challenge C.2 - Domain Independance

Black-boxes rewriting rules are not tied up to a specific application domain and can be found in various and different domains. Each domain comes with its custom properties and tools that can vary in arbitrary complex form. Thus, our proposition must overcome the outlined challenges in a domain-independent way. Domain-independence goes against using a model-dependent $\|$ parallel

composition operator. Thus we consider that the domain under experiment does not provide any support to compose these functions excepting the classical composition operator $\circ$.

4.5 Conclusion

Considering a black-box approach as an actual real-life scenario prevents any form of assessments on off-the-shelf rewriting rules. We defined two challenges we want to address in this work: enabling assessment of white-box properties that are useful in our selected black-box context, in a domain-independent way. In the remainder of this Ph.D. thesis, we propose a formalism that allows one to assess similar properties on black-box rewriting rules, before validating the approach on real large-scale use-cases.

CHAPTER 5

Ensuring Properties on Composition of Black-box Rewriting Rules

"Great things are not done by impulse, but by a series of small things brought together"

Vincent van Gogh in a letter to his brother Theo.

Content

5.1 Introduction

In the previous chapters, we successively depicted compositions of black-box rewriting rules and challenges that arise when assessing white-box properties in such a context. In this chapter, we propose a delta-based formalism, show how it is mapped to existing black-box operators, and how reasonings on top of actions tackle the presented issues and overcome the listed challenges.

5.2 From Black-box Rules to Actions

In this section, we introduce the main and the core idea: bring the problematic of reasoning on black-box rewriting rules to reasoning on their *modifications*. Since properties we want to ensure need to know *what* a rule *has done* (*e.g.*, termination checks that applying the rule reduces the size of the model, confluence needs to know the LHS and RHS of a rule), we need to find a way to compute these modifications that have been made by a rule.

We propose a model that takes the definition of black-box rewriting rules and leverage two hypotheses to yield actions that have been performed by the rule. These modifications are *actions* to be applied to the input to obtain the output, similarly to patch theory [78], [79]. This paradigm shift is depicted step by step in the following subsections.

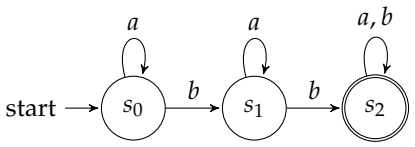
5.2.1 Delta as Models and Vice-versa

Research results from the model community state that a model can be considered as actions [80]. Any model, *e.g.*, UML class diagram, Java code, workflow, can be considered as actions that would create model elements, equivalent to the initial model.

> *Every model can be expressed as a sequence of elementary construction operations. The sequence of operations that produces a model is composed of the operations performed to define each model element.* [80]

Let us illustrate this equivalence, on one of the finite state machines of the running example from SEC. 4.2.1.

Figure 5.1 depicts an example of such equivalence. We can either consider a finite state machine (FIG. 5.1a) or consider its equivalent sequence of actions (LIST. 5.1b). These are PRAXIS [80] domain-independent actions. The *actions-sequence* depicted in LIST. 5.1b can build the FSM modeled in FIG. 5.1a. Each one of these representations can be obtained via a deterministic procedure on the other.



(a) A model of finite state machine

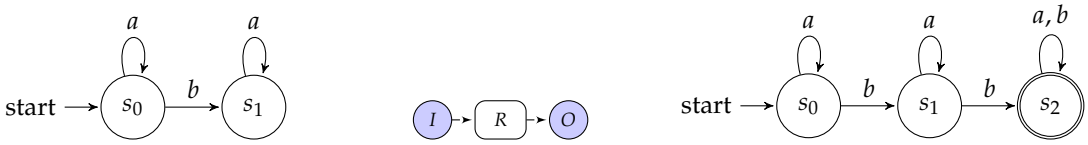
```
[ create('state', s0), setIsInitial(s0)
    ,
create( 'transition', s0, s0, ['a']),
create('state', s1),
create( 'transition', s0, s1, ['b']),
create( 'transition', s1, s1, ['a']),
create('state', s2), setIsFinal(s2),
create( 'transition', s1, s2, ['b']),
create( 'transition', s2, s2, ['a', 'b
    ']) ]
```

(b) A PRAXIS action-based model of the automaton of FIG. 5.1a

Fig. 5.1 – Equivalence between an automaton (left) and a sequence of actions (right)

Example Context Rewriting rules under study are black-boxes. They take a model $m_i \in M$ and output a model $m_o \in M$. Our proposition is as follows: instead of reasoning on black-box rewriting rule, we will reason on their actions. We have shown in the previous subsection that one can consider a model or a sequence of actions that allow one to build it. We need to bridge the gap between our proposition of an action-based approach, and the actual definition of a rewriting rule. In figures, actions are represented as Δ or δ. Along the remainder of this section, we will use the action-based model to formalize our operators and our proposed reasonings. We will leverage the fact that we can either consider a model (*e.g.*, an FSM), or the actions that one can interpret to build it. Actions will be used in the next subsections to define composition operators and two operations: $\oplus$ (patch) and $\ominus$ (diff).

Let us take the following context: an automaton (depicted in FIG. 5.3a) is passed as an *input* to a black-box rewriting rule R, which outputs another automaton (depicted in FIG. 5.3b).



(a) Automaton I input of R (b) Definition of R (c) Automaton O yielded by R

Fig. 5.2 – Example of application of a black-box rewriting rule R (middle), on an input I (left), yielding O (right)

5.2.2 Performing a *diff* Between Models ($\ominus$)

Principle. How are we going from black-boxes rewriting rules to actions? The rules may already take and produce actions. In such a case, we do not need to formalize them specifically. If rewriting rules under study *do not* produce actions, we hypothesize that one can compute them given the *input* and the *output* of the rule via a *diff* operation. This *diff* operation computes the differences that exist between the input and the output models, hence, it allows us to compute, at fine-grains, *what a rule has done* instead of having a plain output model. This

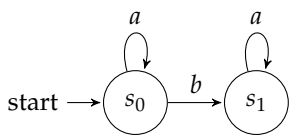
diff operation is called after the classical code-diff operation. In the following, actions are denoted by δ, and actions sequence by Δ. We denote this operation $\ominus$ as specified in EQ. 5.1.

Definition. An operator $\ominus$ exists to perform a *diff* operation between the *input* and the *output*, yielding a sequence of actions. By definition, applying ($\oplus$) this sequence of actions (Δ) on the input (m) yields the output (m'). The definition of the *diff* ($\ominus$) requires the definition of a *patch* ($\oplus$) to be operationalized.

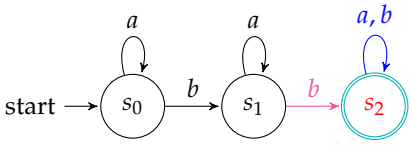
Hypothesis 1: A $\ominus$ operation is available in the application domain to perform a *diff* between models.

$$\begin{aligned} \ominus : M \times M &\rightarrow D^*_< \\ (m', m) &\mapsto \Delta, \text{ where } m' = m \oplus \Delta \end{aligned} \tag{5.1}$$

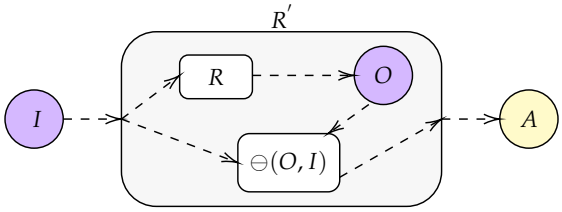
Example of application. If we take the automata I and O depicted in FIG. 5.2, and apply the $\ominus$ operation on them, it will yield the sequence of actions depicted in FIG. 5.3d. The $\ominus$ operation will be called with the output O yielded by R (FIG. 5.3b) and the input I taken as input by R (FIG. 5.3a). It will compute the differences between the output and the input; thus, computing that *the state* s_2 *has been added*, that a *transition between* s_1 *and* s_2 *has been added*, and so on. The complete diff is shown in FIG. 5.3d. Conforming to our new formalism, we bring black-box rewriting rules to a sequence of actions. Our proposition yields a *diff* as depicted in FIG. 5.3d (FIG. 5.3c).



(a) Automaton I input of R



(b) Automaton O yielded by R



(c) Performing a *diff* on input/ouput of R

$$\ominus(O, I) = [\, create(\text{'state'}, \text{'}s_2'), \; set(\text{'isFinal'}, \text{'true'}, s_2), \; create(\text{'transition'}, s_1, s_2, [\text{'b'}]), \; create(\text{'transition'}, s_2, s_2, [\text{'a'}, \text{'b'}])]$$

(d) Sequence A, diff obtained via the $\ominus$ operation

Fig. 5.3 – From rule R to actions sequence A (elements part of the diff A are colored in O)

5.2.3 Performing a *patch* on a Model Given a Sequence of Actions ($\oplus$)

Principle. In order to operationalize a *diff*, we formalize a *patch* operation. This operation allows us to be able to apply the rules, compose them together, and yield a final result. Thus, we assume that a *patch* operation exists, allowing one to take an *input* model and a *sequence of actions*, to yield an *output* model. This operation is for compatibility purposes and allows our proposition to be operationalized at the domain-level. This *patch* operation is called after the classical code-patch operation. An operator $\oplus$ is available, taking an input model and a sequence of actions, and applying the sequence on the input yields the output model.

Definition. The $\oplus$ operation relies on domain-specific application (*i.e.*, *exec*) but its signature is domain-independent, which is sufficient enough for our proposition to work. It is part of our hypotheses that: *(i)* during the application of a sequence of actions on a model, the latter may be in an inconsistent state, but *(ii)* applying all of its actions *must* end in a consistent and correct result. For instance, dangling references or naming issues may occur *during* the application of an actions-sequence, but once *ended*, the *patch* operation yields a consistent model (*e.g.*, without dangling references or naming issues).

Hypothesis 2: A $\oplus$ operation is available in the application domain to perform a *patch* on a model.

$$\oplus : M \times D^*_< \to M$$
$$(m, \Delta) \mapsto \begin{cases} \Delta = \varnothing & \Rightarrow m \\ \Delta = \delta|\Delta' & \Rightarrow exec(m, \delta) \oplus \Delta',\ exec \text{ being domain specific.} \end{cases} \tag{5.2}$$

Example of application. In order to still operationalize our proposition, *i.e.*, being compatible with the initial definition of rewriting rules, we defined the $\oplus$ operator that applies actions on top of a model to yield a modified one. Applied to our example, the $\oplus$ operator takes A (FIG. 5.3d) and I (FIG. 5.2) to produce O (FIG. 5.4). The application of each individual action is domain-specific, *e.g.*, applying an action *create* on a graph, is not the same as applying it on a Java code to create a statement. Nevertheless, the definitions of $\ominus$ and $\oplus$ are domain- and implementation-independent. The whole process, using $\ominus$ and $\oplus$, is depicted in the next section.

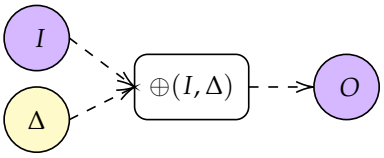


Fig. 5.4 – Application of $\oplus$

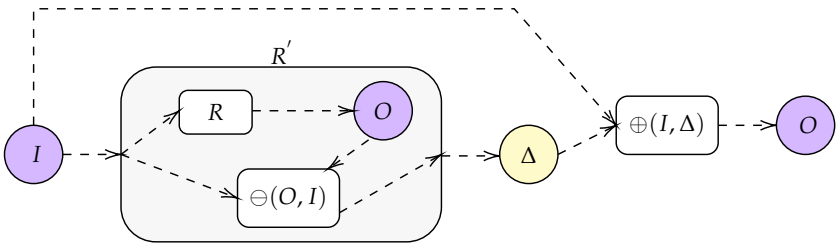


Fig. 5.5 – Compatibility with *apply*

5.3 Composition Operators on Action-based Approach

5.3.1 Compatibility with apply

In the previous section, we defined the $\circ$ operation via the *apply* operator. This operator represented the classical sequential composition of rules. The action-based representation, along with the $\ominus$ and $\oplus$ operators, is compatible with the previously defined semantics for the *apply* composition operator as depicted in FIG. 5.5.

The EQ. 5.3 describes how to apply two rewriting rules using $\ominus$ and $\oplus$ operations and shows the mapping between the original *apply* operator and our proposition to use actions instead. A model m_1 obtained via applying φ_1 on a model m, is equal to applying the modifications made by φ_1 (*i.e.*, Δ_1) on the model m. To avoid confusion, modifications are denoted with Δ or Δ' if they are different.

$$
\begin{aligned}
&\text{Let } m \in M,\ \rho_1 = (\varphi_1, \chi_1) \in R,\ \rho_2 = (\varphi_2, \chi_2) \in R \\
&\quad m_1 = \varphi_1(m) = m \oplus (m_1 \ominus m) = m \oplus \Delta_1, && \chi_1(m, m_1) \text{ holds} \\
&\quad m_2 = \varphi_2(m) = m \oplus (m_2 \ominus m) = m \oplus \Delta_2, && \chi_2(m, m_2) \text{ holds} \\
&\quad m_{12} = apply(m, [\rho_1, \rho_2]) = \varphi_1 \circ \varphi_2(m) = \varphi_1(\varphi_2(m)) = \varphi_1(m \oplus \Delta_2) \\
&\qquad = (m \oplus \Delta_2) \oplus \Delta_1', && \chi_1(m_2, m_{12}) \text{ holds} \\
&\quad m_{21} = apply(m, [\rho_2, \rho_1]) = \varphi_2 \circ \varphi_1(m) = \varphi_2(\varphi_1(m)) = \varphi_2(m \oplus \Delta_1) \\
&\qquad = (m \oplus \Delta_1) \oplus \Delta_2', && \chi_2(m_1, m_{21}) \text{ holds}
\end{aligned}
\tag{5.3}
$$

5.3.2 The *seq* Composition Operator

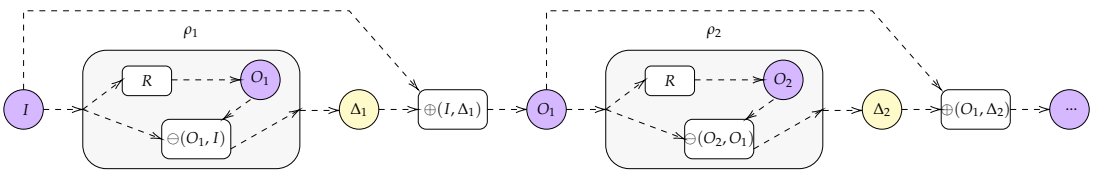


Fig. 5.6 – Mapping between our proposition, and the state-of-practice *seq* operator

Based on the new action-based approach, we add the definition of a new composition operators: *seq*. It allows one to perform rewriting rules in sequence and

check that *all* post-conditions hold, not just the last-one as with the legacy *apply* operator. It chains the calls to rewriting rules and using our proposition.

$$\begin{aligned} &seq : AST \times P^n_< \to AST \\ &p, [\rho_1, \ldots, \rho_n] \mapsto p_{seq} = (\overset{n}{\underset{i=1}{\circ}} \varphi_i)(p), \qquad \overset{n}{\underset{i=1}{\wedge}} \chi_i(p, p_{seq}) \end{aligned} \tag{5.4}$$

5.3.3 The *iso* Composition Operator

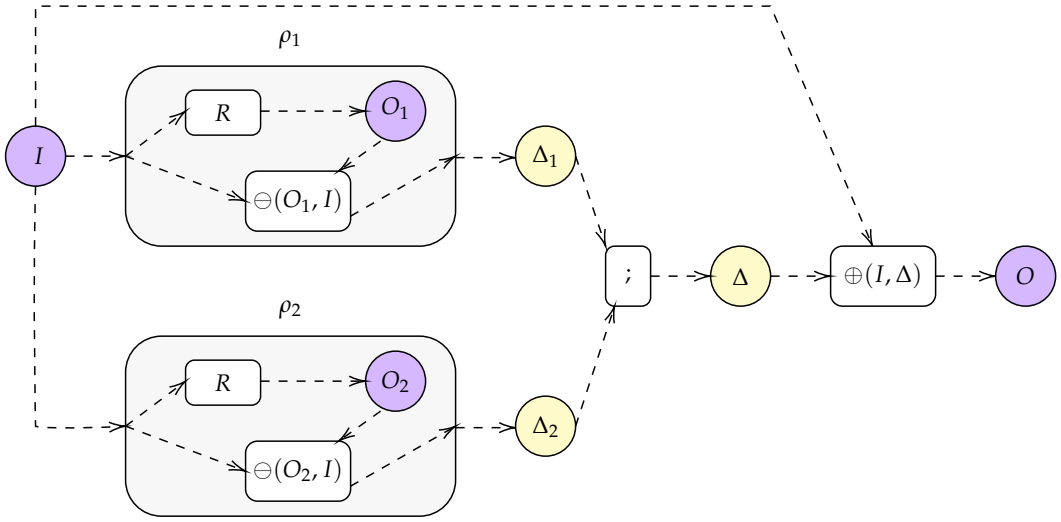


Fig. 5.7 – Example of *isolated* application of transformations using our *iso* operator

Whereas *apply* and *seq* are both sequential operators, we leverage the new action-based approach to add the definition of a parallel composition operator: *iso*. It allows one to perform rules in parallel, in **isolation** of each others (hence *iso*), each one reduced to ultimately yielding a sequence of actions, and then merge their respective results to obtain the final sequence of actions Δ.

$$\begin{aligned} &iso : AST \times P^n \to AST \\ &p, \{\rho_1, \ldots, \rho_n\} \mapsto p_{iso} = p \oplus (\overset{n}{\underset{i=1}{;}} (\varphi_i(p) \ominus p)), \quad \overset{n}{\underset{i=1}{\wedge}} \chi_i(p, p_{iso}) \end{aligned} \tag{5.5}$$

This operator allows one to check their proposed modifications *before* applying them. Thanks to the *iso* operator, one can check deltas together, and ensure the application of all black-box rules, ensuring all post-conditions and one can leverage it also for scaling purposes since this approach allows one to process and apply everything in parallel.

By definition, this operator is associative in conflict-free situations. The output will remain the same, regardless of which order the modifications (*i.e.*, delta) were applied (*i.e.*, via the *patch* $\oplus$ operation).

Our proposition brought reasonings based on black-box rewriting rules to action-based reasonings. The *iso* composition operator allows one to analyze *all* the merged modifications to-be-made before applying them. The next section formalizes what kind of reasonings can be performed on it.

5.4 From Rewriting Rules Reasonings to Actions Reasonings

In the previous section, we introduced an action-based approach and depicted how to go from black-box rewriting rules to actions. This section illustrates the reasonings that can be performed on action-based approaches and depicts the detection of issues named *conflicts*. We discriminate conflicts according to two types: *(i)* syntactic conflicts and *(ii)* semantic conflicts. The latter is related to the violation of postconditions associated to the rewriting rules. The former is a side effect of the *iso* operator, considering that Δs might perform concurrent modifications of the very same tree elements when applied in an isolated way.

5.4.1 Syntactic Conflicts as Overlapping Deltas

Syntactical issues may arise when using the *iso* operator. As a reminder, this operator applies every rule on the same input term and works on the actions they want to perform, before merging the actions and applying the result on the input. For instance, a state cannot be *delete*d and used as a reference at the very same time.

$$\begin{aligned}
&\varphi_1(t) \ominus t = \Delta_1 = [\ldots, \boldsymbol{delete(S_2)}, \ldots] \\
&\varphi_2(t) \ominus t = \Delta_2 = [\ldots, add(\text{“}edge\text{”}, E_{2\omega 0}), \\
&\qquad\qquad \boldsymbol{setReference(E_{2\omega 0}, \text{“}start\text{”}, S_2)}, \\
&\qquad\qquad setReference(E_{2\omega 0}, \text{“}end\text{”}, S_0), \\
&\qquad\qquad setProperty(E_{2\omega 0}, \text{“}value\text{”}, \omega) \ldots]
\end{aligned} \tag{5.6}$$

The *seq* operator cannot encounter such syntactical conflicts, as it is passing the output of a rule to another. The *iso* operator on its part can encounter three kinds of conflicts (EQ. 5.7) at the syntax level[1]: *Concurrent Property Modification* (CPM), *Concurrent Reference Modification* (CRM) and *Dangling reference* (DR). The first and second situations identify a situation where two rules set a property (or a reference) to different values. It is not possible to automatically decide which one is the right one. The latter situation is identified when a rule uses a model element that is deleted by the other one. Using the definition of these conflicting situations, it is now possible to check if a pair of Δs is conflicting through the definition of a dedicated function *conflict*.

$$conflict? : A^*_< \times A^*_< \rightarrow \mathbb{B}$$

If this function returns *true*, it means that the two rewriting rules cannot be applied independently on the very same program. One can generalize the *conflict?*

1. See the PRAXIS seminal paper [80] for a more comprehensive description of conflict detection in the general case.

function to a set of Δs by applying it to the elements that compose the Cartesian product of the Δs to be applied on a term t.

$$
\begin{aligned}
CPM:\ & A^*_< \times A^*_< \to \mathbb{B} \\
& \Delta, \Delta' \mapsto \exists \delta \in \Delta, \delta' \in \Delta', \delta = setProperty(elem, prop, value) \\
& \qquad \alpha' = setProperty(elem, prop, value'), value \neq value' \\
CRM:\ & A^*_< \times A^*_< \to \mathbb{B} \\
& \Delta, \Delta' \mapsto \exists \delta \in \Delta, \delta' \in \Delta', \delta = setReference(elem, ref, elem') \\
& \qquad \delta' = setReference(elem, ref, elem''), elem' \neq elem'' \\
DR:\ & A^*_< \times A^*_< \to \mathbb{B} \\
& \Delta, \Delta' \mapsto \exists \delta \in \Delta, \delta' \in \Delta', \delta = \_(elem, \_, \_) \\
& \qquad \delta' = delete(elem) \\
conflict?:\ & A^*_< \times A^*_< \to \mathbb{B} \\
& \Delta, \Delta' \mapsto CPM(\Delta, \Delta') \vee CRM(\Delta, \Delta') \vee DR(\Delta, \Delta') \vee DR(\Delta', \Delta)
\end{aligned}
\tag{5.7}
$$

5.4.2 Semantic conflicts as postcondition violations

We now consider rewriting rules that are not conflicting at the syntactical level. Semantic assessment is domain-dependent and cannot rely on generic reasonings. This is the reason why we use postconditions as a way to assess the presence of semantic conflicts. We focus here on the postconditions defined for each rule, *w.r.t.* the legacy, sequential and isolated composition operators. The existence of such postconditions is a requirement for our proposition to work at its fullest extent, yet we acknowledge that such postconditions may not be found in all contexts; thus we provide a domain-independent post-condition definition in SEC. 5.5.2.4.

When composed using the $apply$ operator ($p' = apply(p, rules)$), the only guarantee is that the *last postcondition* is true.

When using the *seq* operator, ordering issues may be detected as it checks that *all* postconditions hold. Let $rules = [\rho_1, \ldots, \rho_n] \in P^n$ be a set of rewriting rules. Applying these rules using the *seq* operator can lead to semantic conflict if a rule ρ_i sees its postcondition violated given the initial input term and the final output final. For instance, this can happen if a rule ρ_j, where $j > i$ creates an element that should have been deleted by ρ_i.

When composed using the *iso* operator ($p' = iso(p, rules)$), the resulting program is valid only when *all* the postconditions hold when the rules are *simultaneously* applied to the input program. The fact that at least one postcondition is violated when using the *iso* operator gives a piece of important information: these two rewriting rules cannot be applied *independently* on this program.

5.5 Assessing Properties On Running Example

In this section, we close the loop and apply our proposition on the running example depicted at the beginning of this chapter. We will show, on concrete and

small examples, how our proposition contributes to safely compose black-box rewriting rules as they enable detection of issues that were silenced otherwise.

5.5.1 Detecting Incompatible Rewriting Rules

In this subsection, we will go step-by-step and depict successively the context we take, the operator used, how to apply it to the context, and how to yield syntactical conflicts.

5.5.1.1 Description of the Rewriting System

We recall below the rewriting system of the running example: t, R_0, and R_3. As the rules composing the system are *black-box* rewriting rules, their definitions are not known. Thus, one can only analyze the result of their applications on the initial term t depicted in FIG. 5.8a). The intermediate and final results of the two sequences are depicted respectively in FIG. 5.8b and FIG. 5.8c; and FIG. 5.8d and FIG. 5.8e. Let us apply step-by-step our contribution to this example using the *iso* operator.

5.5.1.2 Paradigm Shift

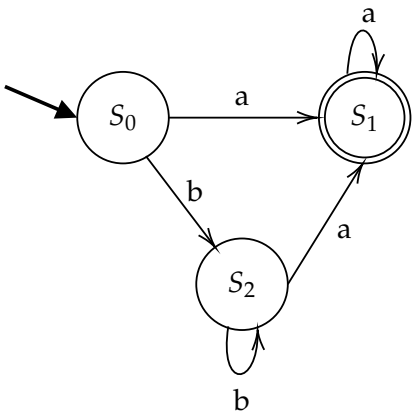
Our proposition is to reason on actions instead of rewriting rules. As described in SEC. 5.2.2, we compute the *diff* between the initial term t and the result of each rule. As we use the *iso* operator, we perform the diffs between t and t_3, and on t and t_0 (or the other way around). A partial view of these diffs is depicted in EQ. 5.8.

$$\begin{aligned}
&\varphi_0(t) \ominus t = t_0 \ominus t = \Delta_0 = [\dots, delete(S_2), \dots] \\
&\varphi_3(t) \ominus t = t_3 \ominus t = \Delta_3 = [\dots, add(\text{“}edge\text{”}, E_{2\omega 0}), \\
&\qquad\qquad setReference(E_{2\omega 0}, \text{“}start\text{”}, S_2), \\
&\qquad\qquad setReference(E_{2\omega 0}, \text{“}end\text{”}, S_0), \\
&\qquad\qquad setProperty(E_{2\omega 0}, \text{“}symbol\text{”}, \omega) \dots]
\end{aligned} \tag{5.8}$$

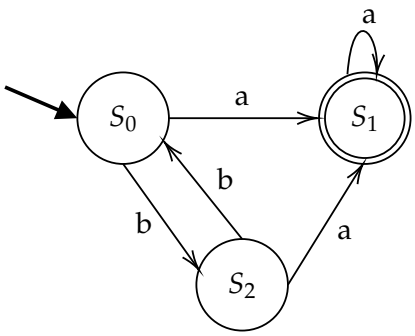
5.5.1.3 Syntactic conflict

The deltas Δ_0 and Δ_3 are then concatenated together (FIG. 5.8) and the resulting Δ is analyzed for conflicting situations. Among the rules that detect conflict, ***D**angling**R**eference* returns true as Δ_0 *deletes* S_2 that is used as a *reference* in Δ_3. Thus, a syntactic conflict is detected, and the two deltas involved can be highlighted. Following the conflicts defined in SEC. 5.4.1, a *DR*-conflict is detected (EQ. 5.9). The syntactical conflict detection gives a piece of information: *among all the rules* used to rewrite the initial term, there exists a pair of rules that cannot be applied *independently*.

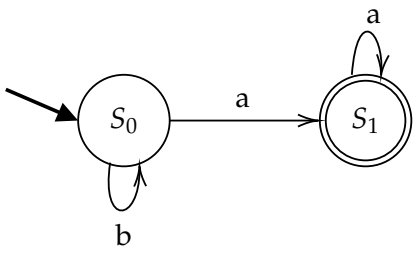
$$\begin{aligned}
DR: \ & A^*_< \times A^*_< \to \mathbb{B} \\
& \Delta, \Delta' \mapsto \exists \delta \in \Delta, \delta' \in \Delta', \delta = _(elem, _, _) \\
& \qquad\qquad \delta' = delete(elem)
\end{aligned} \tag{5.9}$$



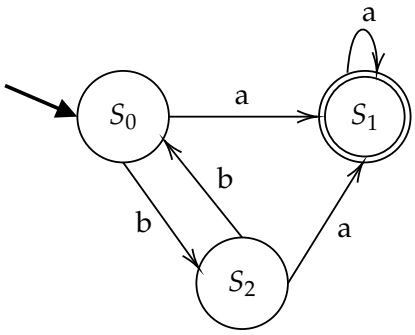
(a) Initial term t



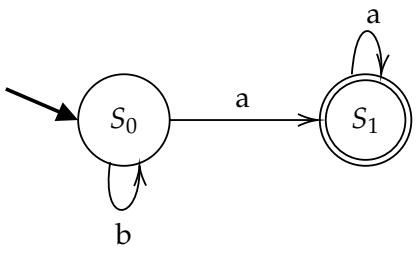
(b) $t_3 = R_3(t)$



(c) $t_0 = R_0(t)$



(d) $t_{30} = R_0(R_3(t))$



(e) $t_{03} = R_3(R_0(t))$

Fig. 5.8 – The two sequences of composition considering the (t, R_0, R_3) rewriting system

5.5.1.4 Overcame Challenges

Confluence-like assessment Confluence states if there is an order in which applying a set of rules, or if every composition equations eventually yield the same result. Thanks to the actions-based approach, the new extra-step allows us to check for such ordering issues. One can look for *conflicts* in the delta-set to-be-applied. We outlined that, even applied independently, the rules are conflicting, implying that all sequences of application may lead to a different result in this context.

Reducing Problem Space Our contribution enables the outlining of the conflicting black-box rewriting rules, and even why and where they are conflicting. By providing the conflicting deltas, linked to the original rule that created it, we drastically reduce the search space in case of issues. It enables human-in-the-loop interactions to let the user gain knowledge about the rules, fix them, or manually handle the conflict.

5.5.2 Detecting Semantic Issues

5.5.2.1 Description of the Rewriting System

We recall below the rewriting system of the running example: t, R_1, and R_2. As before, the rules composing the system are *black-box* rewriting rules; therefore, their definitions are not known. Thus, one can only analyze the result of their applications on t (FIG. 5.9). We see that the two sequences output different results (FIG. 5.12 and FIG. 5.13). Let us apply step-by-step our contribution to this example and use the *iso* operator.

5.5.2.2 Paradigm Shift

Our proposition is to reason on actions instead of rewriting rules. As described in SEC. 5.2.2 and depicted in FIG. 5.8, we compute the *diff* between the initial term t and the result of a given rule. As we use the *iso* operator, we perform the diffs between t and t_1, then on t and t_2. A partial view of these diffs is depicted in FIG. 5.14.

5.5.2.3 Syntactic Conflicts

As one of the delta produced (δ_2) is empty, syntactic conflict cannot occur. By definition, a bug-free rewriting rule cannot conflict with itself; therefore, the concatenated Δ is free of any syntactic conflicts.

5.5.2.4 Semantic Conflicts

As no syntactic conflict was yielded, the concatenated delta can be applied on I, to yield the final output t_{out}(FIG. 5.15).

Then, the term t_{out} can be checked against *all* the rules' postconditions χ . As a *domain-independent implementation* of postcondition checking, we decided to re-apply every rule, and each non-empty sequence of actions will consist of a postcondition violation. This will capture that the composition of the rules does not violate any post-condition violation, meaning that during the composition itself (*i.e.*, when not all the rules have been applied), a post-condition can be violated temporarly. This approach holds if the rules are convergent and meant to be applied only once.

As depicted in FIG. 5.16 R_2 produces a non-empty sequence, yielding a post-condition violation.

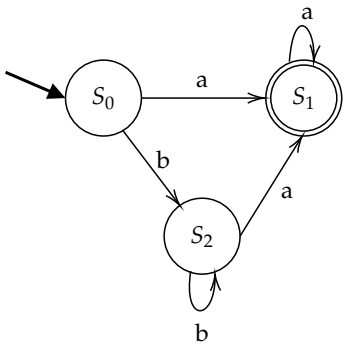


Fig. 5.9 – Initial term t

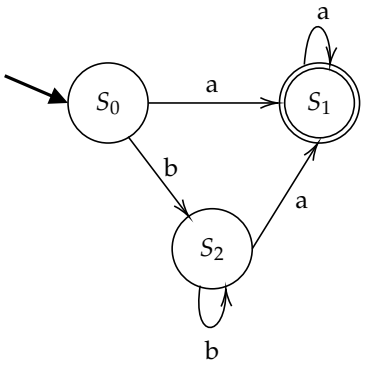


Fig. 5.10 – $t_2 = R_2(t)$

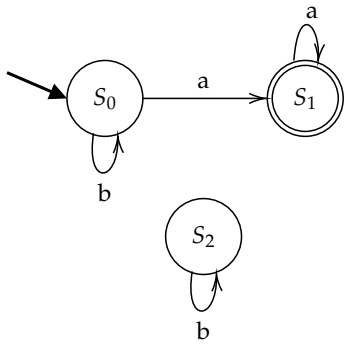


Fig. 5.11 – $t_1 = R_1(t)$

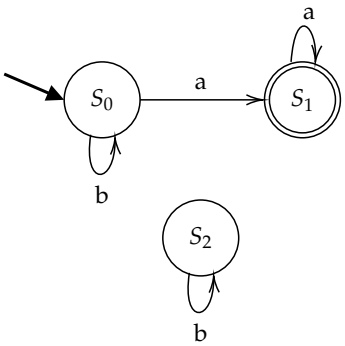


Fig. 5.12 – $t_{21} = R_1(R_2(t))$

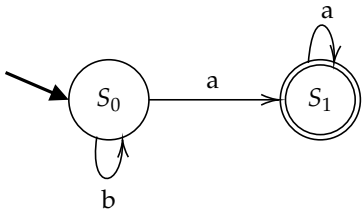


Fig. 5.13 – $t_{12} = R_2(R_1(t))$

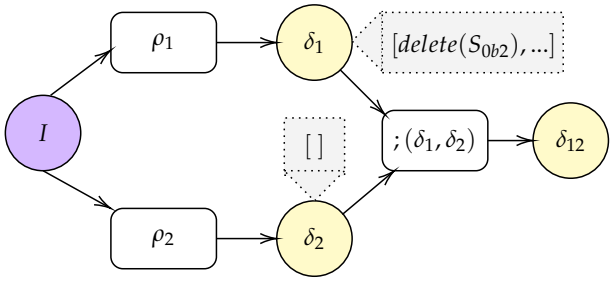


Fig. 5.14 – Application of *iso* to t, R_1 and R_2

5.5.2.5 Overcame Challenges

Confluence-like assessment Confluence states if there is an order in which applying a set of rules, or if every composition equations eventually yield the same result. We outlined that, even applied independently, the rules are not conflict-

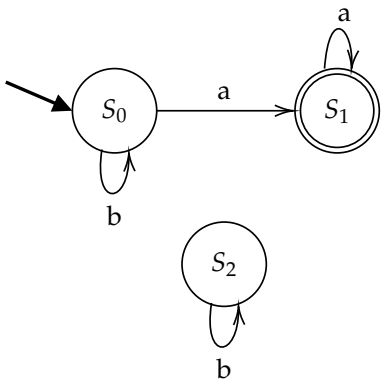


Fig. 5.15 – t_{out}

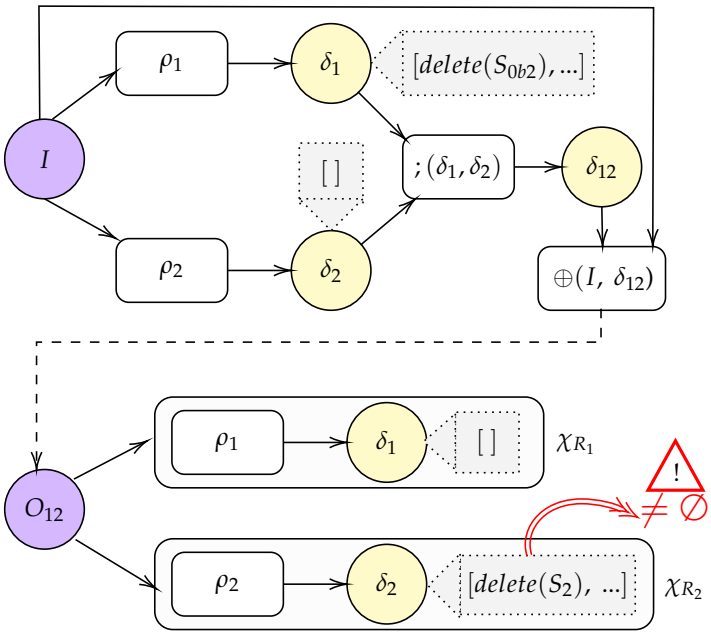


Fig. 5.16 – Application of χ_1 and χ_2 to t_{out}

ing syntactically but that they conflict semantically implying that a sequence of applications can lead to a valid (*i.e.*, non-conflicting) result.

Reducing Problem Space by yielding critical pairs Our contribution enables the outlining of the conflicting black-box rewriting rules, and even why and where they are conflicting. By providing the rule that has its postcondition violated, we reduce the search space in case of issues. By analyzing which model element has been modified lately by R_2, and comparing it with model elements modified by R_1, we can automatically deduce *(i)* the involved rules, *(ii)* a partial order, *(iii)* the involved elements, thus; highlighting critical pairs. This also reduces the search-space and provides stronger insights to the user.

Termination-like assessement We cannot ensure that a rewriting system will never terminate. However, using the *iso* operator, we can ensure that termination is reached and that no further rules need to be applied. Postcondition checks can serve as a termination flag. If all postconditions hold, then compute is over; if a

postcondition does not hold, we state that we do not know if the rewriting system will eventually terminate.

5.5.3 Domain-independence

This subsection assesses the domain-(in)dependence of our proposition and summarizes all the aspects involved. The reasoning on ordering issue checking is independent of the language of action and even independent of the application domain considered.

— The actions-based approach is domain-independent.

— The definition of the *diff* ($\ominus$) and *patch* ($\oplus$) operations are domain-independent. Of course, their implementations are domain-dependent, but we rely on their definitions only to shift to an action paradigm.

— The composition of actions is made using the ; concatenation operator which is domain-independent.

— Syntactic conflicts are defined using rules' types only, not what they manipulate. Therefore, syntactic conflict definition and detection is domain-independent.

— Semantic conflicts are defined using two domain-models yielding a boolean. Again, their implementations are domain-dependent, but their definitions are not. We even proposed a generic domain-independent way of checking postconditions. Therefore, semantic conflicts are defined in a domain-independent way.

Our contribution allows one to reason on *any* black-box rewriting rules, given the two hypotheses mentioned, regardless of its actual application domain, and safely compose them by detecting and yielding syntactic and semantic conflicts, highlighting the involved rules if needed.

5.6 Conclusion

In this chapter, we moved from black-box rewriting rules to actions. We shifted the paradigm to reasonings on an action-based approach instead of black-box rewriting rules and defined the hypotheses needed to perform such a shift: the existence of diff $\ominus$ and patch $\oplus$ operations in the addressed domain. We defined the state-of-practice composition operator $apply$, and *seq* and *iso* operators using this action-based formalism. This formalism allowed us to define reasonings that can be done on actions to detect syntactic or semantic conflicts that would have gone unnoticed previously. Finally, we defined and assessed black-box equivalents of white-box properties such as confluence, termination, and critical-pairs. We illustrated the whole chapter on the running example of this thesis, assessing (non-)conflicting rules, terminating black-box rewriting systems, and yielding critical pairs, reducing the search space for the user. All the formalisms and operations defined in this chapter are domain-independent. This allows our contribution to ensure the safe composition of black-box rewriting rules, in a domain-independent way.

However, we did not consider *(i)* if actual transformations are black-boxes, *(ii)* how our formalism can be actually applied in real-life use-cases, and *(iii)*

whether the outcome in real-life use-cases is as expected. In the next chapters, we validate our proposition on two real-life scenarios. We will check rules that automatically transform the Linux kernel, and rules that modify Android applications to reduce their energy consumption, validating both the relevance and applicability of our proposition and its outcome.

CHAPTER 6

Composing Black-box Rewriting Functions in a Controlled Environment

Content

6.1 Introduction

In the previous chapter we depicted our proposition and how it enables reasonings on black-box rewriting functions. Whereas state-of-practise approaches reason and ensure guarantees on top of white-box rewriting functions, our contribution makes no assumption on the content of these rewriters, neither on what they capture nor what they produce. We have shown the benefit of this approach on our running example, that serves as an illustration example. In this chapter we will validate the whole approach. This chapter will *(i)* validate the black-box assumption, assessing that this black-box context happens in real-life scenario, *(ii)* validate the hypotheses that issues exist when working in a black-box context, *(iii)* apply our proposition on real-life use-cases, validating its conformance to actual scenarios, and *(iv)* check that our proposition applied in real-life scenario enables reasonings capabilities that were not feasible in the initial context.

We chose to validate the points mentionned above in a heavily controlled environment. We selected to analyze the context of the Linux kernel, its development and maintenance. The Linux project is a huge project spanning several decades, developed by skillfull developers, heavily managed by a narrow team of highly competent and trusted developers. Such environment is clearly a non-friendly context to assess our hypotheses as such controlled environment should be conflicts-free and none of the issues that occur when working with black-boxes, described in the previous chapter, are expected to occur.

In this Linux use-case, we more specifically target the automated fixes that occur inside the Linux kernel to fix common issues or make use of new APIs. It is a robust eco-system, built by skillful developers and act as a *real-life* use-case. In the following, the term *developers* designate the developpers of the application, *i.e.*, the domain-developers. A separate team is in charge of developing the automated transformations that will modify the code base to fix different issues.

6.2 Coccinelle and the Linux kernel use-case

6.2.1 A tool to automatically rewrite the kernel

Linux is a family of open source operating systems based on the Linux kernel, an operating system kernel first released on September 17, 1991. It is also a massive code base of more than 20 millions lines of code, maintenained and developed by more than 21,900 contributors [1] via more than 856,000 contributions.

With this amount of contributions, spread over a huge set of contributors, developping Linux and conforming to best practices becomes challenging. Reviewing a contribution before its integration involves notably: conforming the code to Linux formatting guidelines (*e.g.*, naming, namespaces), check for common mistakes or issues, conforming the code to Linux conventions, review the contribution as a feature addition, etc. Integrating a contribution in such ecosystem involves a lot of human effort, and conformance checks must be done as automatically as possible.

1. `https://github.com/torvalds/linux/community_contributors`, in July 2019

In 2006, Muller et al. coined the term of collateral evolution to address the issues that appear when developing Linux drivers: the kernel libraries continuously evolve, and device-specific drivers must be ported to support the new APIs. To tame this challenge, they develop the `Coccinelle` tool [77], [81]. It is used to automatically fix bugs in the `C` code that implements the `Linux` kernel, as well as conforming to code guidelines, or even backporting device-specific drivers [53]. These activities are supported by allowing a software developer to define *semantic patches*.

A semantic patch contains *(i)* the declaration of free variables in a header identified by *at* symbols (`@@`), and *(ii)* the patterns to be matched in the `C` code coupled to the rewriting rule. Statements to remove from the code are prefixed by a minus symbol (-), statements to be added are prefixed by a plus symbol (+), and placeholders use the "`...`" wildcard.

6.2.2 Examples of Semantic Patches

The two examples below are direct excerpts of the examples available on the Coccinelle tool webpage [2].

Automating the use of new API. One of the evolution made in the Linux kernel is to add a new kernel function that replaces a set of functions usually called together. Therefore, they develop a Coccinelle patch R_k, that will make the codebase evolve (LIST. 6.1). It describes a semantic patch removing any call to the kernel memory-allocation function (`kmalloc`, *l.5*) that is initialized with 0 values (`memset`, *l.8*), and replacing it by an atomic call to `kzalloc` (*l.6*), doing allocation and initialization at the very same time. As depicted in *line 7*, wildcard patterns can define guards to prevent the application of the patch, for example here the patch cannot be applied if the allocated memory was changed in between (using the `when` keyword). This semantic patch allows to automatically perform an API evolution at large scale (*i.e.*, the whole Linux codebase) instead of retrieving and analyzing the codebase manually.

```
1 @@
2 type T;
3 expression x, E, E1,E2;
4 @@
5 - x = kmalloc(E1,E2);
6 + x = kzalloc(E1,E2);
7 ... when != \( x[...]=E; \| x=E;
      \)
8 - memset((T) x, 0, E1);
```

Listing 6.1 – `kmalloc` $\wedge$ `memset`$(0) \mapsto$ `kzalloc` (R_k)

Fixing common mistakes. A common issue when developping in C code, is that memory initialization (memset) is not done properly when using pointers (*e.g.*, array or structures). This kind of mistake can be hard to find and implies debugging and human effort. Therefore, they develop a semantic patch R_m, that

2. `http://coccinelle.lip6.fr/impact_linux.php`

will look for such context and fix it if necessary (LIST. 6.2). It will look for a type *T*, which is a pointer (*l.2,3*) that is used in a memset call (*l.7*) and replace its parameter by using pointers (*l.8*). This patch allows to automatically fix easy-to-miss mistakes at large scale, avoiding useless human effort.

```
1 @@
2 type T;
3 T *x;
4 expression E;
5 @@
6
7 - memset(x, E, sizeof(x))
8 + memset(x, E, sizeof(*x))
```

Listing 6.2 – Fix size in `memset` call (R_m)

6.2.3 Semantic Patches as Black-boxes

In the previous sections we described semantic patches as white-boxes and manually extracted easy and comprehensible portion of hand-picked semantic patches, *e.g.*, the actual kzalloc definition contains 275 lines. Actually, these patches can be really complex: they can consider a huge set of different cases; or their definitions can be complex on their own:

— Some patches take advantage of an import mechanism, allowing to capitalize on human effort [3] [4], other are build using dependency mechanisms [5] [6] that require a lot of extra-effort to have the "complete" semantic patch that will be actually executed,

— Coccinelle semantic patches can even run arbitrary scripts (usually in python) that can perform arbitrary operations [7] [8] [9], adding a huge extra layer of complexity to understand what the semantic patch will actually do and of what it actually consists,

— Finally, goto mechanism can be used *inside* a semantic patch definition [10] making even a local analysis complex.

Considering the complexity induced by the elements above, *we consider semantic patches as black-boxes.* In the Linux use-case, *a semantic patch is a black-box rewriting rules that operates over the Linux's codebase*. Again, applying a single semantic patch is a trivial operation as we consider bug-free implementation. Our goal is to ensure the composition of multiple semantic patches, when issues can arise.

3. http://coccinelle.lip6.fr/impact/round.html
4. http://coccinelle.lip6.fr/impact/array.html
5. http://coccinelle.lip6.fr/impact/jiffies.html
6. http://coccinelle.lip6.fr/impact/usbdata.html
7. http://coccinelle.lip6.fr/impact/countptr.html
8. http://coccinelle.lip6.fr/impact/notnull.html
9. http://coccinelle.lip6.fr/impact/sdhci.html
10. http://coccinelle.lip6.fr/impact/kmalloc8.html

6.3 Mapping to our proposition

In this section we map the abstractions and operations of the Linux kernel use-case, presented in the previous sections, to our proposition.

Rewriting rules ρ are implemented as the application of semantic patches to the `Linux` kernel. A `Coccinelle`'s patch is a black-box rewriting rule that will modify the Linux kernel source code.

The diff ($\ominus$) operation (*i.e.*, diff outputing Δ) is implemented by the *code diff* one can obtain as the output of `Coccinelle` usage.

Deltas are classic *diff/patch* actions operating at the *textual* level. The language of action is made of addition, deletion, or update of a given line of code.

The patch ($\oplus$) operation is classical *code patch* operation. It takes a *patch* - *i.e.*, a sequence of additions, deletions, and/or updates or lines of codes; and applies it on an input source code.

The postcondition χ is created to detect semantic conflict. We leverage the following assumption: a semantic patch whose intention is *respected*, yields an *empty* diff. Thus, reapplying the rewriting rule to the *rewritten* kernel must yield an empty diff when the postcondition is respected, or non-empty otherwise. Postconditions are implemented as an *empty-diff* check.

6.4 Example of Overlapping Applications of Semantic Patches

Considering a single semantic patch and applying it on the Linux kernel should not yield any issue. But what about when one applies multiple semantic patches on the Linux kernel? As Coccinelle proposes a set of 59 semantic patches, issues may arise when one applies, *i.e.*, composes, all these patches together, on the same version of Linux.

Let us consider the two semantic patches R_k (LIST. 6.1) and R_m (LIST. 6.2). The intention of applying the first one (R_k) is to make use of calls to `kzalloc` whenever possible in the source code, and the intention associated to the second one (R_m) is to fix bad memory allocation. In the state of practice, applying the two patches, in any order, does not yield any error. However, the application order matters.

```
1  struct Point {
2      double x;
3      double y;
4  };
5  typedef struct Point Point;
6
7  int main()
8  {
9      Point *a;
10     // ....
11     a = kmalloc(sizeof(*a), 0)
           ;
12     // not using a
13     memset(a, 0, sizeof(a));
14     // ...
15     return 0;
16 }
```

Listing 6.3 – Example of a C program (p_c)

For example, let us consider the sample program p_c described in LIST. 6.3 and apply the rules R_k and R_m to it. They can be applied following two sequences detailed below.

$p_{km} = R_m(R_k(p_c))$ Applying the rules in this order, R_k is applied first and doesn't modify the program p_c as its expression line 8 doesn't match with the expression line 13. Then, R_m is applied to the output of R_k (*i.e.*, unmodified p_c) and the erroneous memory allocations are fixed (*l.13*). The *memset* issue is fixed *after* the `kzalloc` merge. As a consequence, the sequence misses some of these `kzalloc` calls (*l.11*). This sequence may miss `kzalloc` call when it implies badly defined `memset`. The sequence $R_m(R_k(p_c))$ leads to a program p_{km} where the intention of R_k is not respected: the `kzalloc` method is not called whenever it is possible in the final program as depicted in LIST. 6.4.

```
1  int main()
2  {
3      Point *a;
4      // ....
5      a = kmalloc(sizeof(*a), 0);
6      // not using a
7      memset(a, 0, sizeof(*a));
8      // ...
9      return 0;
10 }
```

Listing 6.4 – Actual program: $p_{km} = R_m(R_k(P_c))$

$p_{mk} = R_k(R_m(p_c))$ The erroneous `memset` is fixed (LIST. 6.3, *l.13*) *first* and as a consequence the `kzalloc` optimization is also applied to the fixed `memset`, merging *l.11* and *l.13* into a single memory allocation call. In this order, the two initial intentions are respected in p_{km}: all the erroneous memory allocations are fixed, and the atomic function `kzalloc` is called whenever possible. This is the expected result depicted in LIST. 6.5.

```
1 int main()
2 {
3     Point *a;
4     // ....
5     a = kzalloc(sizeof(*a), 0);
6     // ...
7     return 0;
8 }
```

Listing 6.5 – Expected Program: $p_{km} = R_k(R_m(p_c))$

Comparison of yielded errors. Let us apply the three compositions operators *apply*, *seq* and *iso* on this small example. We summarize the results in Tab.6.1 showing the status of the different postconditions given a specific operation. The first two rows state that, when applying a rule, its postcondition holds (by definition). The next two rows use *apply*, that can only guarantee the application of *the last* rules, hence the silenced postcondition violation in the 4^{th} row. The *seq* operator takes advantage of assessing *all* postconditions, hence the yielded postcondition violation marked as a cross. Finally, the *iso* operator states that, even applied in parallel, these two rules are conflicting semantically.

Table 6.1 – Identifying semantic conflicts on the `Coccinelle` example. Elements in parentheses are not known by the user.

$p \in AST$	$p' \in AST$	$\chi_k(p, p')$	$\chi_m(p, p')$	Postconditions
p_c	$\varphi_k(p_c)$	✓	-	✓
p_c	$\varphi_m(p_c)$	-	✓	✓
$\varphi_m(p_c)$	$apply(p_c, [\rho_k, \rho_m])$	✓	(✓)	✓
$\varphi_k(p_c)$	$apply(p_c, [\rho_m, \rho_k])$	(✗)	✓	✓
p_c	$seq(p_c, [\rho_k, \rho_m])$	✓	✓	✓
p_c	$seq(p_c, [\rho_m, \rho_k])$	✗	✓	✗
p_c	$iso(p_c, \{\rho_k, \rho_m\})$	✗	✓	✗

Considering only these two simple and simplified patches, on a dedicated and hand-picked example, issues can arise. The Coccinelle's rules-set is made of 59 different rules, each one implementing many transformations, capturing different contexts to fix a specific issue. This validates *the need for automated reasonings* as no manual approach would scale, and that these semantic patches (*i.e.*, rewriting functions) are considered as black-boxes.

6.5 Ensuring Composition of Rewriting Rules in the Linux Kernel

The validation material for this section can be found on the dedicated repository [11] and has been built using the `Coccinnelle` tool, in the `Linux` ecosystem. Each rewriting rule is defined as a `Semantic patch` working at the code-level. The objective here is to show how the *iso* operator can be used *in practice* to identify conflicts among legacy rules and reduce the number of rules to order when necessary. The validation was performed on 19 different versions of the `Linux` kernel source-code, with an Intel Xeon E5-2637v2 (3,5GHz, 8cores) processor and 64GB of RAM DDR3. The data sample was made by randomly picking a version of Linux per month from january 2017 to july 2018, and analysing the applications and interactions of all semantic patches available in `Coccinnelle` for each kernel version.

6.5.1 State of practice (*apply*) does not provide guarantees

The `Linux` kernel represents around 20M lines of codes, and the `Coccinelle`'s rules set *applicable* to this source code contains 35 semantic patches. Without our proposition, the only way to ensure that no conflicts occured when applying the rules to the source code is to assess that each possible sequence of rule applications (here $35! \approx 10^{40}$) lead to the same result. The patterns matched by each semantic patch are not known as a semantic patch is a black-box. This prevents any analysis between semantic patches and renders the brute-force approach as the only one applicable. Considering one second to apply each sequence (an extremely optimistic guess), it leads to approximately 10^{32} years of computation, for a single commit (the `Linux` kernel versioning system contains more than 700K commits in 2018).

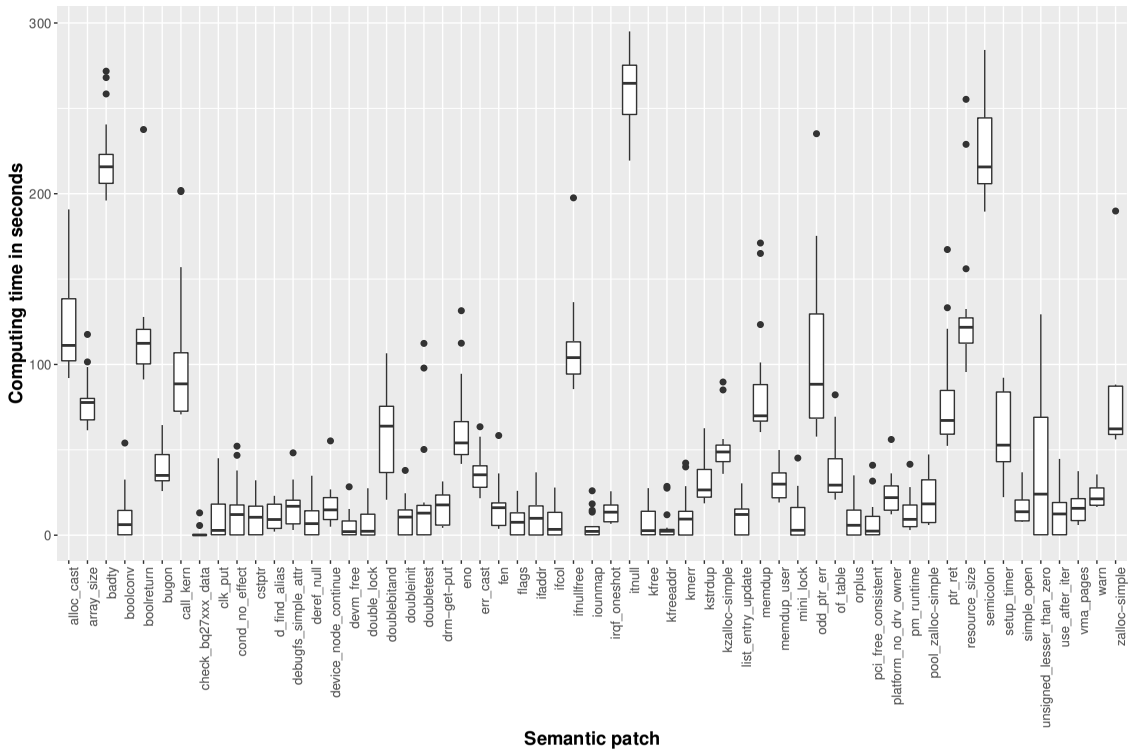
One can obviously argue that computation can be parrallelised to multiple computers. To assess this issue, we measured in FIG. 6.1 the average time taken by each semantic patch to analyse and rewrite the source code. One can notice that 75% of the rules are quick to execute (less than 5 minutes in average for each rule, FIG. 6.1a), and that a few semantic patches (25%, FIG. 6.1b) are complex and slow. In average, without taking into account the time to orchestrate the rule set, *it takes* $\approx$ 190 *minutes to execute a single sequence on the kernel*. It is clearly not reasonable to execute all of the sequences, even in a distributed and parallelized environment.

As a consequence, the current state of practice does not provide any guarantee on the generated source code with respect to rule interactions.

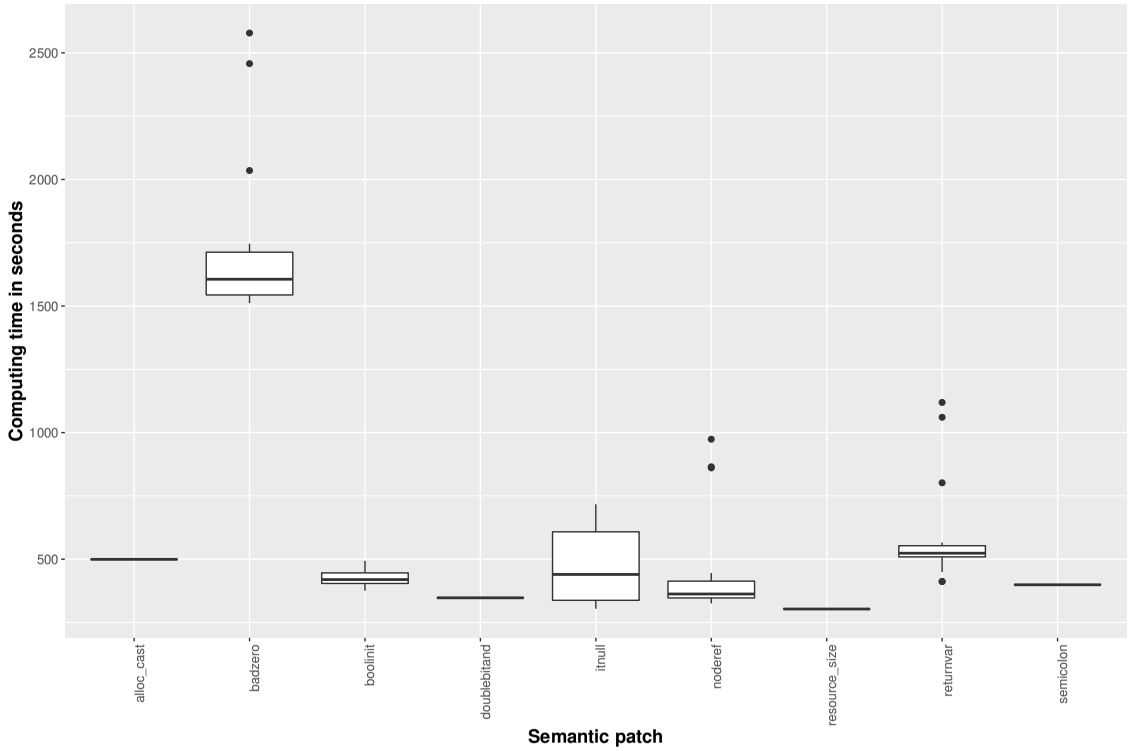
6.5.2 Applying contribution (*iso* operator)

In this section, we show how the *iso* operator described in the previous chapter is used to identify conflicts among rules in a reasonable amount of time, and how to map the proposed formal model to real-life artefacts.

11. `https://github.com/ttben/xp-jsep-linux`



(a) Fast patches (26), applied in less than 5 minutes



(b) Slow patches (9), applied in more than 5 minutes

Fig. 6.1 – Applying 35 semantic patches to 19 versions of the Linux kernel (execution time)

Experimental process. The experimental process applied to *each* version of the kernel is depicted in Fig. 6.2. It perfoms the following steps:

1. Checkout the `Linux` source code at a given version (*i.e.*, a given commit),
2. **Phase 1**: Use *iso* operator by applying *independently* all the 35 semantic patches with `Coccinnelle`. Each patch (*i.e.*, rewriting rule) produces code diffs (*i.e.*, Δs) to be applied to the kernel to rewrite it;
3. Detect any syntactical conflict among the generated patches. A syntactical conflict is detected when several code diffs target the same lines of `C` code in the kernel;
4. **Phase 2**: For conflict-free situations, *apply* the code diffs to the source code to obtain the rewritten kernel and *verify* the 35 postconditions on the obtained kernel.

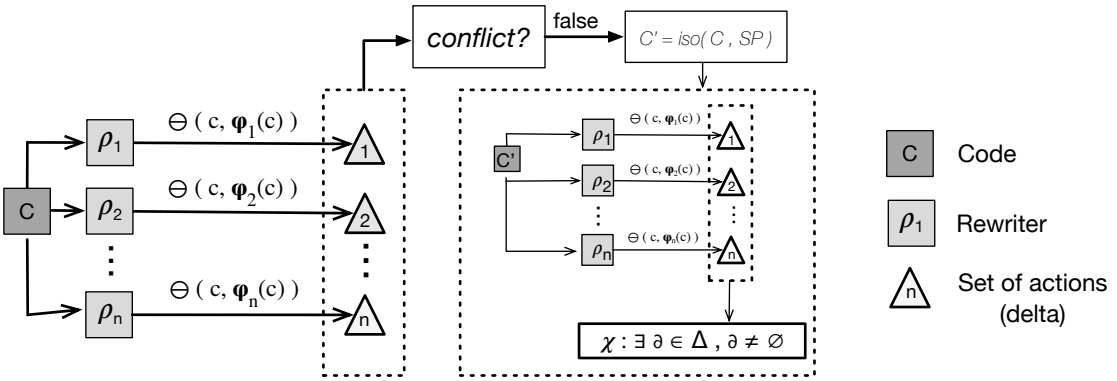


Fig. 6.2 – Experimental process of the linux use-case

Assessment duration. We depict in Fig. 6.3 the time consumed by our experimental setup for each of the 19 versions picked in this context. The average execution time is 190 minutes for a given version, with a low standard deviation. Even if "long", this time is more reasonable than the absence of guarantee identified in the previous paragraph and is compatible with continous build and integration pipelines that can be put in place for such large projects. We describe in the two following paragraphs how the conflict-detection step identified issues that were previously silenced.

6.5.3 Validating the absence of syntactical conflicts

The *conflict?* function, which looks for syntactical issues, is implemented as an intersection between generated deltas of phase 1. As the actions generated are code diffs, they basically consist of a set of *chunks* describing editions (addition or deletion) of the source code. We look then for pairs of *chunks* (c_1, c_2) such as c_1 will modify the same piece of code than c_2 in the codebase.

No syntactical conflicts were found between the 35 rewriters on the 19 different versions of Linux. That was *expected* given the high-level of expertise and the limited number of developers involved in the development of such rewriters.

This is a more important result than it might look like at first sight. It means that independently, each semantic patch behaves correctly. Moreover, the 35 patches written by experts do not target the same locations in such a large codebase and do not overlap at the syntactical level. This emphasizes the difficulty to identify interactions among rules in such a context.

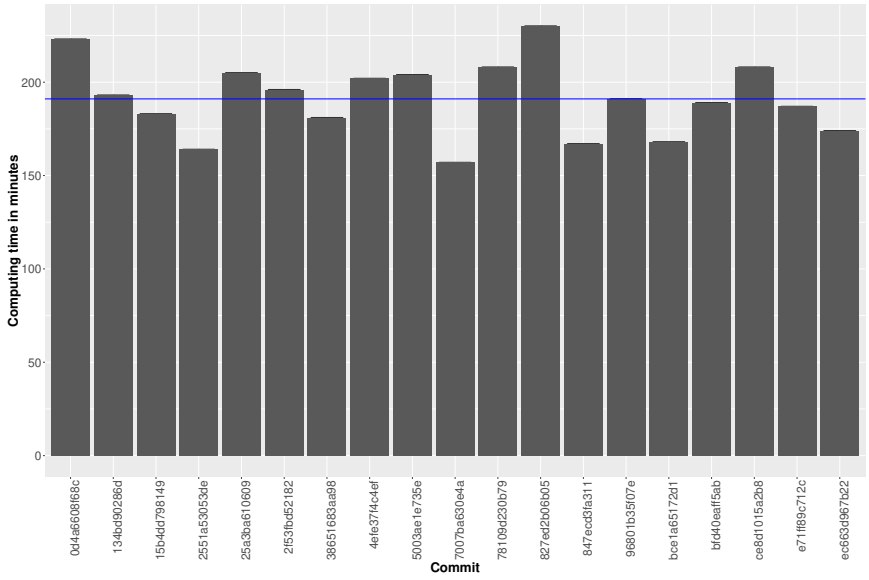


Fig. 6.3 – Execution time of our proposition in minutes (the line is the average time)

6.5.4 Yielding Previously Silenced Semantic Conflicts

Table 6.2 lists the interactions detected in the experimental dataset. Out of 19 versions, 7 ($>$ 36%) presented semantic conflicts that were not detected before. The table is ordered in chronological order, meaning that these interactions come-and-go, and are not solved once and for all. From a software engineer point of view, it is interesting to notice how the process helps to debug the rule set. Among 35 fully-functionning semantic patches available, now the developers only have to focus on two of them: `alloc_cast` and `memdup`. They also know the precise location(s) in their code where these two rules conflicts.

Table 6.2 – Table of interactions between pairs of semantic patches, on a given line of a specific code file.

commit.id	Rewriter #1	Rewriter #2	File with conflict	@line
38651683aa98	alloc_cast	memdup	.../sh_css_firmware.c	146
4efe37f4c4ef	alloc_cast	memdup	.../sh_css_firmware.c	146
b134bd90286d	alloc_cast	memdup	.../vega12_processpptables.c	292
25a3ba610609	alloc_cast	memdup	.../sh_css_firmware.c	133
bce1a65172d1	alloc_cast	memdup	.../vega12_processpptables.c	285
2551a53053de	alloc_cast	memdup	.../vega12_processpptables.c	285
bfd40eaff5ab	alloc_cast	memdup	.../vega12_processpptables.c	292

According to `Coccinelle` documentation, the `alloc_cast` semantic patch performs the following operation: *"Remove casting the values returned by memory allocation functions like `kmalloc`, `kzalloc`, `kmem_cache_alloc`, `kmem_cache_zalloc` etc."* [12]. The `memdup` patch is self-described in its implementation [13]. It avoids to reimplement the behavior of the `kmemdup` kernel-function at multiple locations in

12. `https://bottest.wiki.kernel.org/coccicheck`
13. `https://github.com/coccinelle/coccinellery/blob/master/memdup/memdup.cocci`

the kernel (which implies `kmalloc` and `kzalloc`). By reading the documentation, one might expect an interaction as both rules target the memory allocation, and it is interesting to notice how fine-grained the issue is. These two rules only conflict when applied to a very specific code subset in the kernel, even if their definitions might conflict in essence.

6.6 Conclusion : Overcoming Challenge C1

In Chapter 4.4 we identified challenge C1 that aims to ensure white-box properties in a black-box context. We identified that, by construction of these properties' assessment, none of these properties can be checked *as is* in a black-box context. Neverthless, developpers working in such black-box contexts still *need* to ensure *similar* properties.

In this chapter we validated our proposition to reason on generated deltas to enable reasonings on black-box rewriting functions. We applied our proposition on a real-life, non-friendly, use-case: the maintenance of the Linux kernel.

We outlined the outcome enabled by such an approach, providing reasonings that were not feasible in the state-of-practise. Our approach was capable of detecting two *overlapping black-box rewriting rules* among 39 of these ran against the Linux kernel and providing the whole context, the rule involved, the portion of the code in which it happened, and the two changes of each rules.

Therefore, we detected the *non-confluence-like* of the system, and *highlighted* the *critical pairs* involved.

We also implemented a *termination-like* assessment as a one-step lookahead that assesses if a rewriting rule still has something to do or not.

We also validated that our hypotheses, needed to apply our approach, respectively the presence of a $\ominus$ and $\oplus$ operator, hold in real-life use-cases.

CHAPTER 7

Composing Black-box Rewriting Functions in the Wild

Content

7.1 Introduction

In the previous chapter, we have quantitatively validated that black box rewriting rules may suffer ordering issues, that lead to undesired behavior, and this in the (heavily) controlled environment of the development of the Linux Kernel. We mapped and applied our contribution on this use-case and validated that, in such supposed error-free environment, our contribution was capable of highlighting semantic onflicts in the automated rewriting of the Linux Kernel, overcoming our challenge C1.

In this chapter, the goal is different and is focused on a vast and open ecosystem, opposed to the closed and controlled environment provided by the Linux kernel codebase. We will described two use-cases: the development of webservices in the context of Docker and the optimization of Android applications, respectively acting as an industrial use-case and an academic use-case. This chapter will depict the mapping between each of this use-cases and our contribution, and will validate three things: *(i)* the black-box assumption is not Linux specific, *(ii)* the hypotheses of the existence of *diff* and *patch* operators hold in various domains, and *(iii)* our proposition is not tied up to the Linux use-case (Challenge C2).

7.2 `SPOON`, `Paprika`, and the `Android` use-case

7.2.1 Context: Power-Consuming Practises in `Android` Applications

Smartphones. Smartphones are the most used embedded platform. Their battery requirements are critical and they keep growing overtime. Each manufacturer and kernel developers release guidelines on how to avoid useless battery consumption and how to reduce power consumption on Smartphones' applications [1]. Toolings have been developed to monitor applications in order to identify power-consuming portion of code [82]–[84].

Following guidelines. Software deployed on such platforms (*i.e.*, Apps) must take a specific care of their power consumption, more than classical desktop software. Unfortunately, good software development practises, from a software engineering point-of-view, can become bad practices in terms of power consumption as they may impact negatively the battery lifespan of the final device [30], [85]. Moreover, good practises from the power-consumption point-of-view can be platform specific, and even narrow to a specific version of this platform. Hence, software developpers are not familiar with these guidelines, and may not respect them, and these "counter-intuitive" rules are not applied.

`Paprika` **to yield issues.** To encounter this issue, `Paprika` has been developped [85]. It aims to formalize these power-consumption guidelines, and implement

1. `https://developer.android.com/topic/performance/power`

them as Paprika rules (*i.e.*, sniffing rules), that will find a specific issue, and *report* it. Fixing this issue is meant to reduce the power-consumption of the final application. Whereas Paprika *yields* issues, it does not actually fix them.

SPOON to fix issues. In this eco-system, the actual rewriting part is left to a tool that rewrites Java Abstract Syntax Tree (AST), named SPOON [31], [52]. SPOON is a tool defined on top of the Java language, which works at the *Abstract Syntax Tree* (AST) level. It provides the AST of a Java source code and lets the developer define her transformations. A SPOON rewriting rule is modeled as a Processor, which implements an AST to AST transformation. It is a Java class that analyses an AST by filtering portions of it (identified by a method named isToBeProcessed), and applies a process method to each filtered element, modifying this AST. SPOON reifies the AST through a meta-model where all classes are prefixed by Ct: a CtClass contains a set of CtMethods made of CtExpressions. In this context, a SPOON processor will find an issue yielded by Paprika, and fix it, to reduce the power-consumption of the final application. It applies to the most used smartphone platform: Android.

7.2.2 Example of SPOON processors

An example of processor is given in LIST. 7.1. This SPOON processor (*i.e.*, rewriting rule) will process every CtClass (*i.e.*, Java class) that has setter [2] methods in it (l.3-6). Then, for each setter method (l.12), it uses SPOON factories to build a not-null assessment statement, and wrap it around the initial body of the setter method. This processor is a rewriter used to protect setters from null pointer assignment by introducing a test that prevents an assignment to the *null* value to an instance variable in a class.

In this section we described SPOON processors as white-boxes and manually extracted easy and comprehensible portion of hand-picked Java code. Actually, these processors are really complex: they can consider a huge set of different cases; use part of Java reflexive code; relies on SPOON API to build arbitrary complex statements, etc. Considering the complexity induced by the elements above, *we consider SPOON processors as black-boxes.*

2. We use the classical definition of a setter, *i.e.*, *"a setter for a private attribute `x` is a method named `setX`, with a single parameter, and doing a single-line and type-compatible assignment from its parameter to `x`"*.

```
1 public class NPGuard extends AbstractProcessor<CtClass> {
2  @Override
3  public boolean isToBeProcessed(CtClass candidate) {
4     List<CtMethod> allMethods = getAllMethods(candidate);
5     settersToModify = keepSetters(allMethods);
6     return !settersToModify.isEmpty();
7  }
8
9  @Override
10 public void process(CtClass ctClass) {
11   List<CtMethod> setters = settersToModify;
12   for (CtExecutable currentSetterMethod : setters) {
13     if (isASetter(currentSetterMethod)) {
14       CtParameter parameter = (CtParameter) currentSetterMethod.
             getParameters().get(0);
15       CtIf ctIf = getFactory().createIf();
16       ctIf.setThenStatement(currentSetterMethod.getBody().clone());
17       String snippet = parameter.getSimpleName() + " != null";
18       ctIf.setCondition(getFactory().createCodeSnippetExpression(
             snippet);
19       currentSetterMethod.setBody(ctIf);
20     }
21    }
22 }
23 }
```

Listing 7.1 – Spoon: using processors to rewrite Java code (`NPGuard.java`, R_{np}))

7.2.3 Mapping to our proposition

In this section we map the abstractions and operations of the Android use-case, presented in the previous sections, to our proposition.

Rewriting rules ρ are implemented as a `SPOON` processor that operates over an `Java AST`. A `SPOON` processor is a black-box rewriting rule that will modify a portion of an Android application's source code.

The diff ($\ominus$) and patch ($\oplus$) operations. The *diff* can be implemented by the *code diff* one can obtain as the output of `classic diff`, or via toolings performing *diffs* at the `AST` level [86], or even via `SPOON` itself that keeps records of modifications made to an `AST`. The *patch* operation can be implemented in an analogous way.

Deltas are *diff/patch* actions operating at the *textual* level or `AST` level depending on the tool used. The language of action is made of addition, deletion, or update of a given line of code or node of `AST`.

The postcondition χ is created to detect semantic conflict. We leverage the following assumption: a semantic patch whose intention is *respected*, yields an *empty* diff. Thus, reapplying the rewriting rule to the *rewritten* app must yield an empty diff when the postcondition is respected, or non-empty otherwise. Postconditions are implemented as an *empty-diff* check.

Again, applying a single processor is a trivial operation as we consider bug-free implementation. Our goal is to ensure the composition of multiple processors, when issues can arise.

7.2.4 Example of Overlapping Applications of SPOON Processors

Applying a single SPOON processor is supposed to be bug free, so there is no issue in that. But what happens when applying (*i.e.*, composing) multiple processors?

We consider here two processors. The first one is depicted in file `NPGuard.java` (R_{np}, LIST. 7.1). The second one (`IGSInliner.java`, R_{igs})[3] implements a guideline provided by `Google` when developing mobile application in `Java` using the `Android` framework. It states that inside a given class, a developer should directly use an instance variable instead of accessing it through its own getter or setter (*Internal Getters Setters* anti-pattern). This avoids a useless function call. We do not depict its implementation as it is too complicated to be understandable and usable. This is one (among others) way to improve the energy efficiency of the developed application [87] with `Android`[4].

Like in the `Coccinelle` example, these two processors work well when applied to `Java` code, and always yield a result. However, order matters as there is an overlap between the addition of the *null* check in R_{np} and the inlining process implemented by R_{igs}.

We depict in FIG. 7.1 how these processors behave on a simple class p_j. The figure contains two columns representing sequences of functions calls. Figure 7.1a depicts the initial program, then FIG. 7.1b and FIG. 7.1d depict a possible sequence of application of rules R_{igs} and R_{np} whereas FIG. 7.1c and FIG. 7.1e depict the other possible sequence.

Inlining setters yields p_{igs} (FIG. 7.1b), where internal calls to the `setDt` method are replaced by the contents of the associated method (*line 10*). Then, introducing the *null* guard will modify the body of the setter method (FIG. 7.1d, line 6).

However, when introducing the *null* guard first the content of the `setDt` method is changed (FIG. 7.1e, *line 6*), which *prevents any upcoming inlining* as `setDt` is not considered as a setter based on its new implementation and the used definition. As a consequence, $R_{igs}(R_{np}(p_j)) = R_{np}(p_j)$, and R_{igs} is useless in this very context.

It is interesting to remark that, when considering R_{igs} and R_{np} to be applied to the very same program, one actually expects the result described in FIG. 7.1d: internal setters are inlined with the initial contents of `setDt`, and any external call to `setDt` is protected by the guard.

Comparison of yielded errors. Let us apply the three compositions operators *apply*, *seq* and *iso* on this small example. We summarize the results in Tab.7.1 showing the status of the different postconditions given a specific operation. The first two rows state that, when applying a rule, its postcondition holds (by definition). The next two rows use *apply*, that can only guarantee the application of *the last* rules, hence the postcondition violation in the 4^{th} row. The *seq* operator takex s advantage of assessing *all* postconditions, hence the yielded postcondition violation marked as a cross. Finally, the *iso* operator states that, applied in parallel, these two rules are syntactically and semantically *non-conflicting*.

3. `https://github.com/GeoffreyHecht/spoon-processors/blob/master/src/main/java/io/paprika/spoon/InvokMethodProcessor.java`

4. `http://stackoverflow.com/a/4930538`

```
1 public class C {
2
3  private String dt;
4
5  public String setDt(String s)
       {
6   this.dt = s;
7  }
8
9  public void doSomething() {
10  setDt(newValue)
11 }
12 }
```

(a) Example of a Java class (`C.java`, p_j)

```
1 public class C {
2
3  private String dt;
4
5  public String setDt(String s)
       {
6   this.dt = s;
7  }
8
9  public void doSomething() {
10  this.dt = newValue /* <<<<
       */
11 }
12 }
```

(b) $p_{igs} = R_{igs}(p_j)$

```
1 public class C {
2
3  private String dt;
4
5  public String setDt(String s)
       {
6   if (s != null) /* <<<< */
7    this.dt = s;
8  } /* <<<< */
9
10 public void doSomething() {
11  setDt(newValue)
12 }
13 }
```

(c) $p_{np} = R_{np}(p_j)$

```
1 public class C {
2
3  private String dt;
4
5  public String setDt(String s)
       {
6   if (s != null) /* <<<< */
7    this.dt = s;
8  } /* <<<< */
9
10 public void doSomething() {
11  this.dt = newValue /* <<<<
       */
12 }
13 }
```

(d) $p_{np \circ igs} = R_{np}(R_{igs}(p_j))$

```
1 public class C {
2
3  private String dt;
4
5  public String setDt(String s)
       {
6   if (s != null) /* <<<< */
7    this.dt = s;
8  } /* <<<< */
9
10 public void doSomething() {
11  setDt(newValue)
12 }
13 }
```

(e) $p_{igs \circ np} = R_{igs}(R_{np}(p_j))$

Fig. 7.1 – Spoon: applying processors to Java code

Here we depicted the smallest set of rules ran against a toy and handpicked example, thus ending with only 2 combinaisons possible. The number of automated transformations (around 10) grows and keep changing along versions of

Table 7.1 – Identifying semantic conflicts on the Spoon example

$p \in AST$	$p' \in AST$	$\chi_{igs}(p, p')$	$\chi_{np}(p, p')$	Postcondition
p_j	$\varphi_k(p_c)$	✓	-	✓
p_j	$\varphi_m(p_c)$	-	✓	✓
$\varphi_{np}(p_j)$	$apply(p_j, [\rho_{igs}, \rho_{np}])$	✗	(✓)	✗
$\varphi_{igs}(p_j)$	$apply(p_j, [\rho_{np}, \rho_{igs}])$	(✓)	✓	✓
p_j	$seq(p_c, [\rho_{igs}, \rho_{np}])$	✗	✓	✗
p_j	$seq(p_c, [\rho_{np}, \rho_{igs}])$	✓	✓	✓
p_j	$iso(p_c, \{\rho_{igs}, \rho_{np}\})$	✓	✓	✓

android. Thus, the combinatory approach, even automated, is not feasible and one needs to ensure the safe composition of SPOON processors.

7.2.5 Overlapping of Energy Anti-patterns in Android Applications

In this section, the objective is to focus on a vast and open ecosystem, opposed to the closed and controlled environment provided by the Linux kernel codebase (SEC. 6.5). We analyzed 22 different Android apps publicly available on GitHub and took 19 rules that detect and correct when possible energy-related anti-patterns in Android apps [88]. Our goal in this section *is not* to *quantitavely* validate that each rewriting rules conflict, but to show that issues described in the motivation section of this chapter (*e.g.*, overlaping rules) happen in real-life android applications, on a set of apps that we do not manage. We selected 22 public android applications that matched our technical requirements (*e.g.*, android version compatible with both Paprika, SPOON, and our in-house Fenrir tools), thus reducing the number of android applications analyzed.

We will first focus on the *characterization* of the overlap that exists among these rules before diving into a concrete example to see in practice how our contribution properly supports software developers.

The experiments were run on a mid-2015 MacBook Pro computer, with a 2,5 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 of RAM.

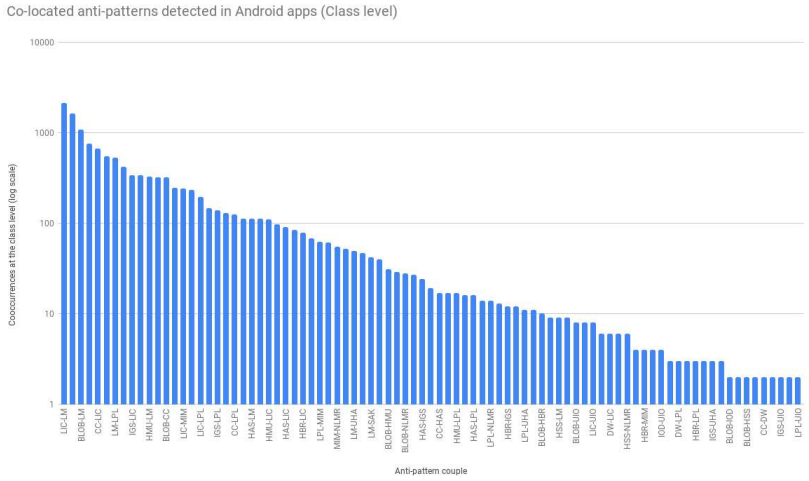
7.2.5.1 Overlapping Anti-patterns Detection

Experiment tooling. Paprika is a tool that allows one to analyse an Android application and detect anti-patterns, including the energy-related ones. Along with these anti-patterns, respective corrections are developed. According to the toolchain used at the implementation level, the rewriting rule is here a function that rewrites the AST of an Android application [87] using SPOON. Thus, fixing multiple anti-patterns at the same time can lead to postcondition violations, moreover if they happen at the very same location. We consider here the 22 energy anti-patterns available for detection in the Paprika toolsuite [5]. We use the visualisation tool associated with Paprika logs to identify pairs of co-located anti-patterns. This situation can happen at three different levels: *(i)* the class level,

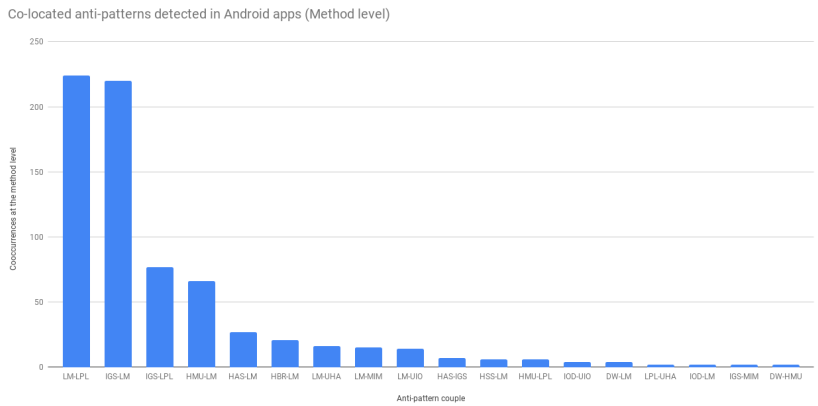
5. https://github.com/GeoffreyHecht/paprika

(ii) the method level and *(iii)* the instruction level. When several anti-patterns are colocated within the same scope, there is an high *probability* that repairing the overlapping patterns will interact.

First, to identify the order of magnitude of such interactions, we only consider overlapping pairs. We depict in FIG. 7.2 the result of analysing the 22 anti-pattern detection rules for the 19 apps of our dataset. At the class level (FIG. 7.2a), we detected up to 2,130 co-occurences of the *Leak Inner Class* (LIC [6]) and the *Long Method* (LM [7]) anti-patterns (FIG. 7.2b). We detected 44 pairs of overlapping anti-patterns at the *class* level, among the $\binom{22}{2} = 231$ pairs, meaning that almost 20% of the rules overlapped at this level in our dataset. At the *method* level (FIG. 7.2b), 18 pairs are (still) overlapping, representing 8% of the possible conflicts. Even at the *instruction* level, three anti-patterns interact together. These results strengthen the need to *automate* the detection of rule interaction issues on concrete examples, as it is not useful to analyse the 231 possible rule combinations but only a small subset of such set.



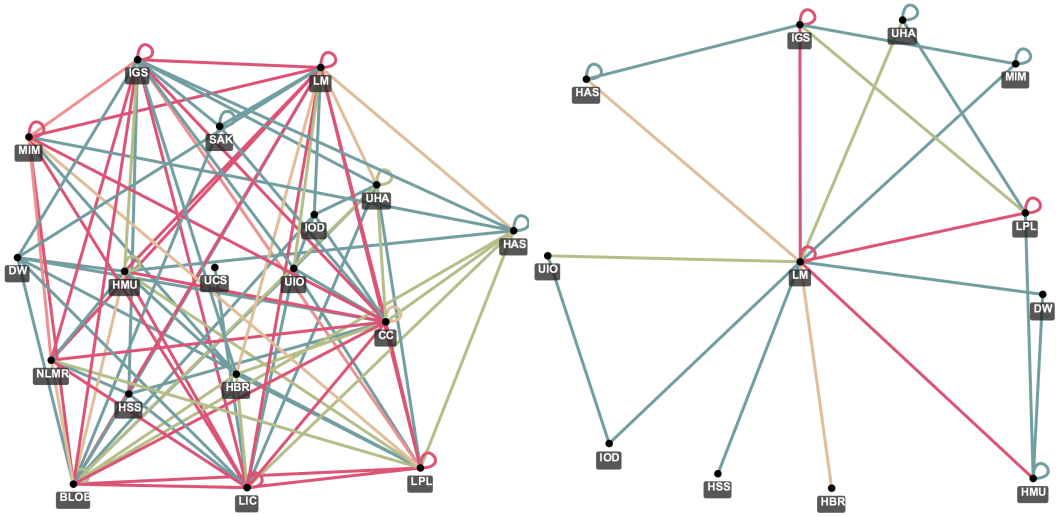
(a) Overlapping anti-patterns detected at the ***class*** level



(b) Overlapping anti-patterns detected at the ***method*** level

Fig. 7.2 – Identifying pairs of overlapping anti-patterns in 22 Android apps

6. Usage of a non-static, anonymous, inner class, leading to memory leak.
7. Method that has significantly more lines than the other, and can be splitted into smaller ones.



(a) Class level (b) Method level

Fig. 7.3 – Representing anti-patterns colocations

The previous analysis only considered *pairs* of anti-patterns. We used the Fenrir tool [8] to visualize at a coarse-grained level the *relationship* existing among multiple anti-patterns. We represent in FIG. 7.3 the relationship that exists among anti-patterns. Each node of the graph is an anti-pattern, and the existence of an edge between two nodes means that these two anti-patterns were detected at the very same place in the dataset.

7.2.5.2 Concrete Example

In the previous paragraphs, we validated the existence of overlaps between anti-patterns in existing applications, emphasizing the need for interaction detection mechanisms as the one described in the section motivation of this chapter. Unfortunately, it is very difficult to reproduce the build chain associated to these applications (even when the apps rely on tools such as Maven or Gradle), limitating the possibility to fix and rebuild all these apps in an automated way. To tame this challenge and validate the safe reuse of code rewriters (*i.e.*, the safe compositiong of rewriting rules) in the Android context, we made the choice to perform an in-depth analysis of a single application.

In the Java ecosystem, each rewriting rule is defined as a Spoon `Processor` working at the AST level, and we also used the same mechanism to implement the associated postcondition, as another `Processor` that identifies violations when relevant. To exemplify our contribution on a concrete application, we consider here as a validation example the development of a real Android application. Based on the collaborative catalogue *Android Open Source Apps* [9], we selected the *RunnerUp* [10] application. This application is developed by an external team, is open-source, has a large number of installations (between 10,000 and 50,000) and positive reviews in the Android Play Store. From a source code point of view, it has 316 stars on its GitHub repository (December 2017) and has involved

8. `https://github.com/FlorianBourniquel/Fenrir`
9. `https://github.com/pcqpcq/open-source-android-apps`
10. `https://github.com/jonasoreland/runnerup`

```
1    delete(2)
2    add(2, "this.column =
        dbColumn")
```

(a) Delta of the application of `IGS`

```
1    delete(11)
2    add(11, "if(name != null)
        {")
3    add(12, "this.column =
        name;")
4    add(13, "}")
```

(b) Delta of the application of `AddGuard`

```
1    import org.runnerup.export.format
2    class DataTypeField {
3        private String column = null;
4        public DataTypeField(String dbColumn) {
5            this.setColumn(dbColumn);
6        }
7        public void setColumn(String name) {
8            this.column = name;
9        }
10   }
```

(c) Initial program

```
1    import org.runnerup.export.format
2    class DataTypeField {
3        private String column = null;
4        public DataTypeField(String dbColumn) {
5            this.column = dbColumn;
6        }
7        public void setColumn(String name) {
8          if(name != null) {
9            this.column = name;
10         }
11       }
12   }
```

(d) Final program

Fig. 7.4 – Rewriting the RunnerUp Android application (excerpt)

28 contributors since December 2011. It defines 194 classes implemented in 53*k* lines of code. This application is dedicated to smartphones and smartwatches, thus its energy efficiency is very important.

From the software rewriting point of view, we reused here four different rules. The first one, named R_λ, is used to migrate plain old iterations to the new λ-based API available since `Java 8`, helping the piece of software to stay up to date. The second one, named R_{np}, is used to introduce guards preventing *null* assignments (LIST. 7.1) in setters, introducing safety in the application. The two others are dedicated to energy consumption anti-pattern fixing: R_h replaces `HashMaps` in the code by a more efficient data structure (`ArrayMaps` are preferred in the Android context), and R_{igs} inlines internal calls to getters and setters (SEC. 7.2.4).

We act here as the maintainer of *RunnerUp*, who wants to reuse these four rules. As there is no evident dependencies between the rules, she decides to use the *iso* operator to automatically improve her current version of *RunnerUp*:

$p'_{ru} = iso(p_{ru}, \{R_{np}, R_{igs}, R_h, R_\lambda\})$.

Figures 7.4a and 7.4b show what has been modified by each rule (*i.e.*, the delta they produced). There is no interaction between those two sets of modifications. It happens that all the postconditions hold when applied to p_{ru} and p'_{ru}: *(i)* there is no call to internal getter/setter left in the final program p'_{ru} (Fig.7.4d), and *(ii)* there is no "un-guarded" modification of private field. Thus, the *iso* operator can be used in this case. The maintainer does not have to wonder about ordering issues *w.r.t.* this set of rules ($4! = 24$ different orders).

To validate the *seq* operator, we consider a *slightly different implementation* of the R_{igs} rule, named R'_{igs}. This rule rewrites a setter even if it does not contain a single line assignment, and expects as postcondition that the call to the setter is replaced by the contents of the method in the resulting program. With such a rule, $p'_{ru} = iso(p_{ru}, \{R_{np}, R'_{igs}, R_h, R_\lambda\})$ is not valid with respect to its postcondition, as $\chi'_{igs}(p_{ru}, p'_{ru})$ does not hold, as depicted in Fig. 7.4d. Actually, the yielded program contains calls to the initial contents of the setter, where the *guarded one is expected* according to this postcondition.

Considering this situation, the maintainer is aware that *(i)* isolated application is not possible when R'_{igs} is involved for p_{ru} and *(ii)* that the conflicting situation might involve this specific rule. She can yield a valid program by calling $iso(p_{ru}, \{R_{np}, R_h, R_\lambda\})$, meaning that these three rules do not interact together on p_{ru}, and thus an order involving R'_{igs} must be defined. The main advantage of the *seq* operator is to fail when a postcondition is violated, indicating an erroneous combination that violates the developers intention. Any call to the *seq* operator that does put R'_{igs} as the last rule will fail, thanks to a postcondition violation. Thus, among 24 different available orderings, the expected one is ensured by calling $p'_{ru} = seq(p_{ru}, [\ldots, R'_{igs}])$. The expected program is depicted in LIST. 7.2.

```
1 import org.runnerup.export.format
2 class DataTypeField {
3     private String column = null;
4     public DataTypeField(String dbColumn) {
5       if(name != null) {
6         this.column = dbColumn;
7       }
8     }
9     public void setColumn(String name) {
10      if(name != null) {
11        this.column = name;
12      }
13    }
14 }
```

Listing 7.2 – Final version of the RunnerUp excerpt using the *seq* operator

7.2.6 Conclusion

In this section, we identified the composition problem that exists when composing multiple rewriting rules using `Spoon`. We quantitatively validated on open-source Android applications that rewriting rules, aiming to fix energy consuming apps, overlap even at the instruction level. Then, we zoomed on an

external Android application, using four rewriting rules designed to identify and fix anti-patterns, following the latest guidelines from `Google` for Android development and outlined concretely how overlappings black-box rewriting rules badly affects the whole application.

7.3 Docker

7.3.1 Fast and Optimized Service Delivery

7.3.1.1 Context Description

Web Services. The *Service-Oriented Programming* (SOP) paradigm has recently evolved to the definition of microservices that cooperate together in a scalable way. Monolithic deployments used until then do not comply with the needs associated with such ecosystem [89]. As part of the microservice paradigm comes the idea of quickly propagating a change from development to production [90], according to a *DevOps* approach. *Development* and *Operations* are no longer separated in the service lifecycle, and a change in a given service can be automatically propagated to production servers through an automated delivery pipeline. In this context, it is up to the service developer to carefully describe how a microservice will be delivered, using dedicated technologies.

Among these technologies, the adoption of the container approach is tremendously increasing [91]. Containers ensure that a given microservice will run the same regardless of its environment, easing the repeatability of build, test, deployment and runtime executions [92], [93]. Containers are faster at runtime and boot-time, lighter than virtual machines, and scale better [94]–[97]. In the container field, the Docker engine quickly became the reference platform for builds and deployments of micro-services [98].

Building artefacts by reuse. Building a non-trivial container image is a difficult process. Describing an image is an imperative process, where a *service deployment descriptor* is written (*e.g.*, a *dockerfile* in the Docker ecosystem) to describe as shell commands how the microservice is installed and configured inside the container. Following an *off-the-shelf* approach, a container is defined on top of others (reused as black boxes). However, this implementation is not compliant with the open/-closed principle, as it is open for extensions (a descriptor extends another one), but not closed for modifications (a descriptor does not provide a clear interface about its contents, making reuse hazardous). By hiding the contents of an image as a blackbox, deployment instruction can conflict with the hidden one, *e.g.*, overriding executables, duplicating installation of the same piece of software in alternative versions, or shadowing executables. It leads to erroneous deployments, detected at runtime. Moreover, the technologies supporting microservice deployment evolve constantly, to make it more efficient or scalable. This evolution can drastically change the way the deployment engine is implemented, and abstraction leaks can occur (*i.e.*, an internal technological choice inside the deployment engine the final user must take into account when writing a service descriptor). It is up to the service developer to stay up to date with ever-changing guidelines that implements fixes to abstraction leaks.

Optimize artefact following official guidelines. As the service to be deployed must be deployed fastly and at scale, the quality of the artefact providing the service is critical. Properties such as its weight impact transfer time and deploy time. Moreover, as for all software artefacts, maintenance is an important aspect which is rendered difficult by the low-level approach of such service deployment descriptors. Their low level of abstraction combined with the fast change of the deployment technology makes maintenance a difficult and critical task.

To mitigate these issues, guidelines were created to develop service deployment descriptors. These guidelines can be provider-specific (*e.g.*, docker); team-specific (*e.g.*, CentOS team has its own guidelines); company specific (*e.g.*, the Amadeus company has its set of guidelines), etc.

7.3.1.2 Example of Overlapping Guidelines

One of the critical properties of software deployment artefacts such as Dockerfiles is their **weight**. Their weight is linked to the number of information units they embed, and in the Docker ecosystem, these units are stored in *layers*.

7.3.1.3 Guidelines Examples

Let us consider the service deployment descriptor described in LIST. 7.3. It is a `Dockerfile` that reuses the latest `alpine` image and installs packages for a `Javascript` and `C` applications.

```
1 FROM alpine:latest
2 MAINTAINER Ben <ben@super-nice.cc>
3 RUN apt-get install nodejs
4 RUN apt-get install npm
5 RUN apt-get install maven
6 RUN apt-get install coccinnelle autoconf
7 CMD ["nodejs"]
```

Listing 7.3 – Dockerfile *d*

Reducing size by merging instructions (R1). The more instructions your Dockerfile has, the more layers your Docker image will have, the heavier it will be, the longer will be the transfer and deployment times, and more expensive will be the storage. Thus, one of the official guidelines is to minimize the number of layers in the image. Actually, one can merge similar contiguous instructions to reduce the number of layers. For instance, contiguous `RUN` docker instructions using `apt-get` can be merged into a single one and their bodies concatenated. Thus, the result of applying R_1 on d is depicted in LIST. 7.4.

Install only what is needed (R2). Another way to reduce the footprint of an artefact is to simply install fewer packages. As one may not know, when installing a specific package, some package manager will also silently install additional packages that may be used later. This automatic addition, which is user-friendly in a standard context, becomes a bad practise in a context of building a deployable artefact. Thus, one can add `-no-install-recommend` option to avoid this on d (LIST. 7.5).

```
1 FROM alpine:latest
2 MAINTAINER Ben <ben@super-nice.cc>
3 RUN apt-get install nodejs npm  maven
    coccinnelle autoconf
4 CMD ["nodejs"]
```

Listing 7.4 – $R_1(d)$

```
1 FROM alpine:latest
2 MAINTAINER Ben <ben@super-nice.cc>
3 RUN apt-get install --no-install-
    recommend nodejs
4 RUN apt-get install --no-install-
    recommend npm
5 RUN apt-get install --no-install-
    recommend maven
6 RUN apt-get install --no-install-
    recommend coccinnelle autoconf
7 CMD ["nodejs"]
```

Listing 7.5 – $R_2(d)$

Sorting package installation (R3). As the number of packages needed to be installed in order to deploy a given service can become huge, it is a good practise to sort the packages alphabetically for a single $apt - get$ install command. This will ease the maintenance of the service descriptor and will speed up finding a specific package in the whole set of instructions. Thus, applying it on d will yield the Dockerfile described in LIST. 7.6.

```
1 FROM alpine:latest
2 MAINTAINER Ben <ben@super-nice.cc>
3 RUN apt-get install nodejs
4 RUN apt-get install npm
5 RUN apt-get install maven
6 RUN apt-get install autoconf coccinnelle
7 CMD ["nodejs"]
```

Listing 7.6 – $R_3(d)$

7.3.1.4 Context Example

If we consider the three rules R_1, R_2, R_3 described above, and apply them on this `Dockerfile`, overlappings and order-related issues occur. There exist 6 different combinations to apply these rules, and only two are valid w.r.t the intent of the rules. Sorting the packages first (R_3) will only impact the last `RUN` instructions (LIST. 7.6), ***but***, if one merges all the consecutive `RUN` instructions (R_1) and ***then*** sorts the packages (R_3), one will end up with a minimized Dockerfile, with sorted list of packages. Thus, applying $R_2(R_3(R_1(d)))$ or $R_3(R_2(R_1(d)))$ will yield the Dockerfile depicted in LIST. 7.7.

```
1 FROM alpine:latest
2 MAINTAINER Ben <ben@super-nice.cc>
3 RUN apt-get install --no-install-
    recommend autoconf coccinnelle maven
    nodejs npm
4 CMD ["nodejs"]
```

Listing 7.7 – Dockerfile *d*

7.3.2 Mapping to our proposition

In this section we map the abstractions and operations of the Docker use-case, presented in the previous sections, to our proposition.

Rewriting rules ρ are implemented as substitutions that operate over an textual Dockerfile. A substitution is a black-box rewriting rule that will modify a portion of a Dockerfile source code.

The diff ($\ominus$) and patch ($\oplus$) operations can be implemented as a *classic textual diff/patch*.

Deltas are *diff/patch* actions operating at the *textual* level The language of action consists of addition, deletion, or update of a given line of code.

The postcondition χ is made to detect semantic conflict. We leverage the following assumption: a semantic patch whose intention is *respected*, yields an *empty* diff. Thus, reapplying the rewriting rule to the *rewritten* dockerfile must yield an empty diff when the postcondition is respected, or non-empty otherwise. Postconditions are implemented as an *empty-diff* check.
Again, applying a single guideline is a trivial operation as we consider bug-free implementation. Our goal is to ensure the composition of multiple guidelines, when issues and overlappings can arise.

7.3.3 Validation: Issues and Overlaps

We described in the previous subsection, in a small and dedicated example, that overlappings and order-relating issues can occur. We did this at small scale for the sake of the example. In this subsection, we check that issues and overlappings can occur at large scale.

7.3.3.1 Dataset

Dockerfiles We built a dataset of *dockerfiles* available on GitHub, the community code-versioning reference platform. We collected an initial set of 24,357 deployment descriptors (*i.e.*, Dockerfiles). Details about this collection, the composition, and the building of the dataset are available in Appendix B. Over these *dockerfile*s, 5.8% (1,412) were considered as trivial (*i.e.*, having less than 3 instructions [11]) and were removed from the dataset. *The remaining* 22,945 dockerfiles *regroup* 178,088

11. A parent reference and a single command.

instructions and represent our experimental dataset, denoted DS. As issues can occur when analyzing a dockerfile or a dockerfile with its reused extension, we also analyze composed dockerfiles that we name *normalized* dockerfiles. A normalized dockerfile is the content of this very dockerfile, appended to the content of its parent dockerfile, again appended to its parent dockerfile, until reaching the root level. The normalized version of our dataset, denoted $\overline{DS}$, is made of also 22,945 *dockerfile*s but due to extension mechanism, it regroups 285,142 instructions. Isolated *dockerfile*s of DS contain between 3 and 202 instructions with 7.76 instructions per *dockerfile* on average; normalized *dockerfile*s of $\overline{DS}$ have between 3 and 202 instructions, with 14.37 on average. The smallest sizes are the same since it is our lower-threshold for trivial *dockerfile*. The highest size is a single *dockerfile* that is bigger than every normalized *dockerfile*s. The most interesting metric here is that normalized *dockerfile*s double in size on average. Normalized *dockerfile*s of $\overline{DS}$ have between 0 and 6 parent *dockerfile*s with 1.45 level of parents on average.

Guidelines. From the guidelines in the official Docker webpages [12], we identified and implemented 15 of them as they are both highly used by the community and general enough to be relevant for a wide number of *dockerfile*s. They are detailed Appendix A of this chapter.

7.3.3.2 Guideline violation (issues)

In this subsection, we focus on identifying guidelines violations to check that issues occur in real-life scenarios. Fig. 7.5 shows how many *dockerfile*s are detected as violation of a given guideline. The blue bars represent non-composed Dockerfiles, whereas yellow bars represent composed Dockerfiles, *i.e.*, we consider the content of the dockerfile and its parents' contents. We note that somes guidelines (*e.g.*, G_2, G_7, G_15) are violated by a lot of Dockerfiles (around 1,000). We also note that guidelines G_3, G_4, G_5, G_9, G_{10} and G_{11} are violated the same amount of times, which is very low. This is due to the fact that *(i)* those errors are rarely made and *(ii)* are more likely to be made by beginners (*i.e.*, at the bottom of the hierarchy). We also note that the 9 remaining guidelines are more violated when applying the normalized operator. This difference corresponds to guidelines violation that *cannot be detected* without taking the normalized descriptor into account, as our approach does.

Fig. 7.6 shows how many *instructions* are detected as violating a given guideline in isolated and normalized modes (using a logarithmic scale). This figure focus on *instructions*, so a Dockerfile can violate a guideline multiple times. This gives insights about how many times a *dockerfile* violates a given guideline and therefore that a high-level *dockerfile* has a given flaw. For instance, guideline #2 is violated by a very small amount of instructions, which impacts a lot of *dockerfile*s. The amount of instructions involved in the violation of guideline 2, in the isolated dataset, is a bit smaller than the amount of instructions involved in the violation of guideline 2, but in the normalized dataset. This means that fixing a small amount of instructions, in parents dockerfile, may actually fix a lot of *dockerfile*s.

12. `https://docs.docker.com/engine/reference/builder/`, `https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/`.

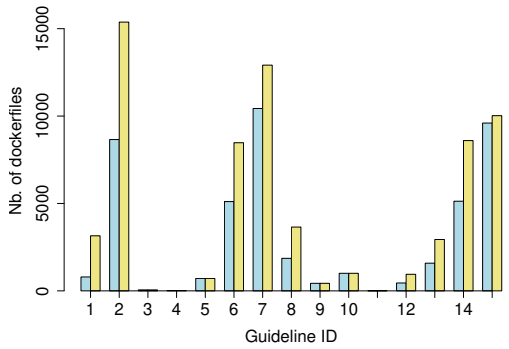


Fig. 7.5 – Number of *dockerfile*s violating a given guideline

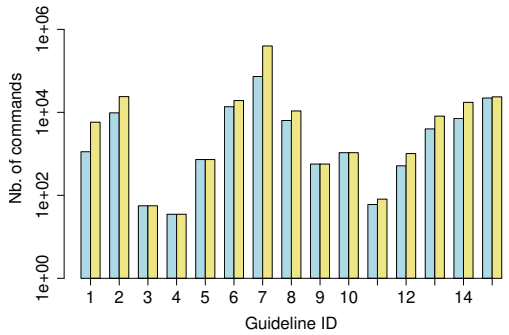


Fig. 7.6 – Number of *instruction*s violating a given guideline

7.3.3.3 Overlappings

In the previous section we checked that issues (*i.e.*, guideline violations) occur in real-life scenario, on public and community-driven Dockerfiles. In this section, we focus on the overlappings that exists between the instructions involved in these issues. Do issues occur on the same portion of code or do they occur at different location; *i.e.*, their resolution may be problematic or not?

Some guidelines are going to conflict with each other, by construction (*e.g.*, updating before installing, and adding specific arguments to *aptget* command). Half of the extracted guidelines target *RUN* commands, hence are more likely to be conflicting. An interference matrix of conflicting guidelines can then be built.

Table 7.2 shows the number of *dockerfile*s that present *potential* conflicts for each guideline pair. We represent only the upper right part of the matrix, as it is symmetric by construction. There is a potential conflict when two guidelines are violated on the same *dockerfile d* and target the same kind k of commands. For example, 8,492 *dockerfile*s violate guideline 6 and guideline 7, whereas only 124 *dockerfile*s violated guideline 1 and guideline 5. Two issues can occur in a single *dockerfile* but on different instructions and therefore create no new conflict, but this information is not available on Table 7.2.

Table 7.3 shows the number of *instructions* that are *really* conflicting, *i.e.*, conflicts occurring on the same instructions of the same *dockerfile* and producing different results. We note that some guidelines pairs (*e.g.*, G_1 and G_{15}) are violated in

Table 7.2 – *Dockerfile*s containing guidelines violation pairs

$G_{i,j}$	1	5	6	7	13	14	15
1	–	124	2,212	2,901	1,731	2,810	1,816
5	–	–	524	685	176	436	629
6	–	–	–	8,492	2,110	6,331	6,531
7	–	–	–	–	2,601	8,690	10,223
13	–	–	–	–	–	2,351	1,861
14	–	–	–	–	–	–	5,965
15	–	–	–	–	–	–	–

many *dockerfile*s (1816) but that *only* 671 instructions are really in conflict; whereas others (*e.g.*, G_6 and G_7) are violated in around 8,500 *dockerfile*s, and that more that 20,000 instructions are really conflicting. This result shows that these guidelines *are often violated together*, on *a lot* of instructions, exposing an understanding problem of the platform by the service designers.

Table 7.3 – Instructions containing guidelines violation pairs (*i.e.*, real conflicts)

$G_{i,j}$	1	5	6	7	13	14	15
1	–	15	1,795	5,445	4,100	3,509	671
5	–	–	9	360	12	174	314
6	–	–	–	20,094	1,211	9,155	14,655
7	–	–	–	–	5,239	19,474	50,256
13	–	–	–	–	–	1,815	1,211
14	–	–	–	–	–	–	8,680
15	–	–	–	–	–	–	–

7.3.4 Conclusion

In this section, we identified the composition problem that exists when composing multiple rewriting rules using `Docker guidelines`. We zoomed in on an external Dockerfile, using three rewriting rules designed to identify and fix bad practises that add useless weight to the final artifact, or violate good practises. Then, we quantitatively validate on open-source Dockerfiles that rewriting rules, aimed at lightening dockerfiles and following good maintenance practices, overlap even at the instruction level.

7.4 Conclusion: Overcoming C2

In this chapter we described two use-cases, Android and Docker, that use black-box rewriting rules, and mapped our proposition to them. Contrarily to the previous chapter that performed a quantitative validation of syntactic and semantic conflict in a controlled environment (the Linux Kernel), this chapter

depicted a quantitative validation of overlapping rewriting rules in two different domains in *uncontrolled* environments.

We showed that rules (*i.e.*, `SPOON` processors) that aim to decrease the power consumption of android apps overlap at the instruction levels ; and that rules (*i.e.*, Docker guidelines) that aim to lighten and optimize docker images overlap at the instruction levels too. This partially validates challenge C1 as our proposition was able to outline the conflicts and the involved rules.

These two very different domains have the same issues. We mapped both of these contexts to our proposition, showing how each domain is captured and handled by our contribution. Therefore, this chapter overcomes challenge C2, aiming for a domain-independant approach, and we validated the applicability of our proposition in real industrial and academic use-cases.

CHAPTER 8

Conclusions and Perspectives

"There is no real ending. It's just the place where you stop the story."

Frank Herbert [99]

Content

8.1 Conclusion

Composing things comes from the fact that we decompose them to handle their complexity. Decomposing without proper recomposition is a meaningless, error-prone, and time-consuming task.

Whereas state-of-the-art acknowledges the issue and provides powerful ways of (re)composing things in a model-driven fashion, state-of-practice does not. The state-of-the-art approaches and techniques rely on the assumption that the composition, along with the elements to be composed are *known* or *formalized* in a specific way, enabling powerful reasoning mechanisms, ensuring the whole composition. The state-of-practice cannot always have such characteristics or does not work that way, preventing any automated safety checks on the composition and its result.

There is a need, from the field of practice, for insurance, guarantees, and reasonings, on their non-well-formalized compositions. These composition operators are designated as "black-boxes" in opposition as the well-formalized "white-boxes" of the state-of-the-art.

This thesis proposed a delta-oriented approach to enable reasonings to ensure properties on black-box compositions. The chosen approach relies on the assumption that two operators, respectively $\ominus$ (diff) and $\oplus$ (patch), exist in the addressed application domain. Our approach bridged state-of-the-art techniques and formalisms with state-of-practice assumptions and context. The former $\ominus$ operator allows us to shift reasonings from an artifacts-based reasonings to a modifications-based one. These modifications were introduced in the state-of-the-art and used as a base in this thesis. We brought the composition issue from an open-world composition to the (re)conciliation of known deltas. Our contribution enabled the capture of various black-box composition operators, both from the industrial and academic worlds. By shifting from domain-specific models to domain-independent actions, we successfully formalized state-of-practice composition operators.

Moreover, this shift allowed our approach to extend the existing compositions by enabling reordering in compositions and a dedicated reasoning step. Finally, over this reasoning step, we formalized confluence-like and termination properties, allowing our contribution to detect and yield conflicts, in a domain-independent way. Our delta-based approach enabled syntactic and semantic conflicts detection, a step forward ensuring white-box properties in a black-box context.

The latter $\oplus$ operator allows one to apply our proposition in real-world use-cases. This operator takes conflict-free modifications and uses a domain-dependent operator to apply them on an arbitrary artifact, allowing our proposition to be operationalized in various domains. The implementation of this operator is domain-dependent, but its definition as we used it in our formalism is not. We validated the applicability of the existing delta-based reasonings, along with the hypotheses made, on real use-cases representing widely different domains. We validated that our contribution enables one to assess conflicts-free situations, or yield the rules and the portion of code involved in the conflict to the final user. This validation was made in the context of the Linux kernel maintenance where rules modifying the Linux codebase are black-boxes. No guarantee was ensured regarding

the conflicts between rules and our contribution ensured conflicts-free situations and yielded scoped conflicts when needed. We also validated our contribution in less-controlled environments such as Android and Docker, where rules aim to optimize their respective artifacts given arbitrary guidelines. We *quantitatively* validated that composition-issues happen in those contexts, and described how our contribution detect such overlapping rules that would have badly affected the final artifact.

8.2 Perspectives

This section gives an overview of short-term to long-term perspectives and visions. The first perspective is a short-term goal and targets merge conflicts in a code versioning context. Then, the second and third perspectives are mid-term ones. One focuses on characterizing black-box composition operators, assessing their algebraic properties and measuring non-functional metrics. The third perspective acts as a support to build efficient, coherent, and optimized machine learning pipelines. Finally, the last perspective is a long-term one. It aims to perform multi-criteria optimization on equations of composition and ensuring that the optimizations are applied safely.

8.2.1 Make Git merge smarter

8.2.1.1 Context

When building a software system, multiple teams are involved and work in parallel with developing the same system. Using a code versioning solution, such as Git, they can diverge from a version of a codebase, develop features in parallel, then reunite their work, speeding up the whole development process. This reunification is a *code merge* operation [100]. This idea of diff and merge is quite old [58], [59], [101]–[103] and in practice, developers still use line-based unstructured tools to handle merges. Nowadays, this merge is done via textual *diff* and textual merge and does not exploit language-dependent information such as structure, or order sensitivity of some elements. More or less recent research works have been done to improve the precision of the merge operation in specific domains [104], [105] while others try to reach a sweet-spot between better merge and execution time added by adding the extra computation [106]–[108]. Merging code is known to be a complex and time-consuming task, and the reasons for why a merge may automatically fail are not fully known [109]–[112], but developers try to avoid them as much as possible, and research tries even to predict merge conflict beforehand [113], [114].

8.2.1.2 Proposed approach

We propose to apply the proposition and contribution made in this thesis to the context of automatic merge conflict detection and resolution in code versioning to improve the smartness, precision, and recall of the merge process. We will exploit already existing sources of *diffs* such as textual diff, or smart diffs working at the AST level [86], [108] and reconcile them in the same approach depicted in ModelBus [51]. Again, we need to compose the various differences safely, and

yield conflicts, the same way nowadays merge work, but with various sources of diff. However, we can go a step further: we can use off-the-shelf transformations that will try to solve yielded conflicts. These transformations are still to be defined and are part of the early results described in the next section. The transformations will capture conflicts, *e.g.*, concurrent addition of statements, and will transform these conflicting actions into a new one that solves the issue. Again, these off-the-shelf transformations that fix conflicts need to be composed safely, at the meta-level this time. The whole point is that, *again*, our contribution can be applied as-is, even at the meta-level of transformations, modifying actions that represent transformations at the AST-level.

8.2.1.3 Early results

To start this work, we first analyzed the conflicts that occur in real-life scenarios. We use recent findings classifying merge conflicts [115] to describe conflicts that happen the most, to formalize them as off-the-shelf transformations. Thus, each transformation is effectively applied onto the dataset to measure and check its "performance". We listed state-of-the-art tools that perform code merge and automated merge-resolutions and measure how many merge scenarios were fixed automatically, and how many are not. The early conclusion of this study is that state-of-the-art tools do not automatically fix merge scenarios correctly most of the time, leaving room from improvements.

8.2.2 Characterize black-box composition operators

8.2.2.1 Context

In this Ph.D. thesis, we validated composition operators, measuring their execution time, assessing their pros and cons, comparing multiple implementations of composition operators. A lot of the work done was repeated across experiments, tailoring the development to match the context of the experiment. This repetition brought us to take a step back and ask what are the reusable abstractions and operations between experiments? What does it mean and imply to perform experiments on composition operators? As seen in the motivation chapter of this Ph.D., targetted composition operators are black-boxes, performing paradigm-shifts, via non-deterministic procedures sometimes. This is the case for a black-box composition that we did not study, develop or use in this thesis: Familiar. Familiar is a tool suite that allows one to manipulate feature models for many purposes, *e.g.*, building a catalog from product descriptions [116]. It proposes many compositions operators [32], with varying semantics, is developed in Java, uses reflexivity, performs paradigm shift to perform the composition, and use non-deterministic algorithms to move between different representations.

8.2.2.2 Proposed Approach

End-users of FAMILIAR highlighted issues raising questions regarding the algebraic properties of the operator:

> *"From a set of N descriptions of products, can I merge N/2 descriptions twice, then merge the two results?"*

> *"I use Familiar in a distributed context, can I merge the same description multiple times without changing the semantic of the result?"*

We chose to map these questions to algebraic questions at the developer-level, *i.e.*, the developer of FAMILIAR has to assess if its operator is, for instance, associative, commutative, or idempotent. Following the overall approach of this thesis where operators are *black-boxes*, assessing commutativity (or any other algebraic properties) is not a trivial thing to do. It is not feasible following formal methods as no information about the definition of the operator is available.

Moreover, algebraic properties (*e.g.*, commutativity) hold regardless of the input passed as operands. Classically, if an operator is commutative, this implies that it respects the commutativity property on any inputs. As no such strong result can be made with black-box composition operator, one has to find a middle-ground. Of course, practitioners and engineers would want to know that their composition operator is *always* commutative, but how one can guarantee that a reflexive and arbitrarily complex Java code, calling many external libraries, composing two automata, is commutative? Of course, one would want to guarantee 100% of the time, but as this seems to not be feasible in practise, a best-I-can-do approach is acceptable.

As recently pointed out in the literature [117], "*Conducting technology-oriented experiments [...] without proper tool support is often a time-consuming and highly error-prone task.*". Conducting experiments in the composition domain, moreover in a black-box context is even more challenging. Yet, the need for proper tooling has been identified, and solutions fitted for composition use-cases are still needed. We propose a composition-operator-oriented framework that allows one to reason and benchmark her own composition operator.

To answer this question, we list below the challenges that need to be overcome:

— Formalizing of a composition operator from a benchmark point-of-view,

— Defining a composition-oriented experiment, its goal, and its measurements,

— Ensuring the quality of the measurements,

— Ensuring the reproducibility of the experiment,

— Efficiently navigate the search space, as a brute force approach is not feasible in practice,

— How to give confidence in algebraic assessments?

8.2.3 Building proper and efficient machine learning pipelines

8.2.3.1 Context

Machine learning (ML) pipelines aim to create knowledge from user-data. An ML pipeline is composed of different steps (*e.g.*, filtering data, transforming data),

each step constraining the possibilities in the next steps. As ML is an active community, a lot of new steps appear every day; thus, creating an ML pipeline to answer a specific question is a difficult task. Assessing if newer preprocessing can be attached to older ones or to the previous algorithm is a difficult task that cannot be operationalized at a human scale, because the number of possible pipelines rises exponentially.

8.2.3.2 Proposed approach

Machine learning pipelines are made by concatenating preprocessing and algorithms. Our approach would be to take the lense of composition and consider the building of ML pipelines as a composition issue. The concatenation-as-composition approach would help developers of ML pipeline by reducing the search space they are confronted to. By formalizing preprocessors' expectations on their input data, and properties of their output data, along with algoritm's requirements, an automatic composition may be triggered to help the developer in his task. Then, by applying the proposition of this thesis, considering preprocessors as black-box composition operators, we would be able to gain knowledge of what the preprocessing steps actually do, feeding the preprocessors' expectations and properties database.

8.2.4 Using algebraic properties to optimize a composition equation

8.2.4.1 Context

We have seen in this Ph.D. thesis that applying multiple composition operators on an input can lead to ordering issues. As they may be many ways to arrange different composition operators, this leads to a *composition of composition* challenge. One arrangement may be different from another in terms of the semantic of their outputs, again, as seen in this thesis. For instance, an arrangement may be invalid from a domain point-of-view. This often happens when some concerns must be applied last, *e.g.*, security, logging. Thus, all arrangement not applying these last are wrong from a domain point-of-view.

Moreover, even if all arrangements of composition operators end up in the same result (*i.e.*, the considered system is strongly confluent), every arrangement may not be stricly equivalent regarding functionnal and non-functionnal metrics. For instance, an arrangement may be quicker to compute than another, hence optimizing a non-functionnal metric (*i.e.*, execution-time). Let us consider the simple arrangement described in EQ. 8.1.

$$((A \cup B) \cup C) \tag{8.1}$$

If the $\cup$ operation copies all the information from its left operand to its right operand, one should always put the smallest of the operands as left-hand side of the $\cup$ operation. Thus, if A is bigger than B, and C is smaller than A and B, the following arrangement is supposedly more efficient, from an execution time point-of-view that the one depicted in EQ. 8.2.

$$(C \cup (B \cup A)) \tag{8.2}$$

The concern of optimizing non-functional metrics such as execution time is of crucial importance in reactive systems and where dynamic compositions is used. Moreover, when such compositions are applied at large-scale (*e.g.*, a huge UML model, a massive codebase), non-functional metrics are critical. The feasibility of the whole composition approach relies of the capability of the composition to be executed in a reasonable amount of time. In a context where millions of lines of code have to be composed together (*e.g.*, massive merge scenarios), memory usage and execution time are of primary concerns.

8.2.4.2 Challenges

— How to optimize non-functionnal metrics of a composition of compositions without interfering with its functional requirements?

— How to optimize non-functionnal metrics of a composition of compositions ensuring that the result will remain equivalent?

— How to find the most optimized arrangement of compositions, if any?

— How to safely compose multiple optimization to be applied on a composition equations?

8.2.4.3 Proposed approach

A sequence of compositions (*i.e.*, an arrangement) can be formalized in an algebraic context, considering a composition operator as an algebraic operator manipulating models. An arrangement will be formalized as an *equation* of compositions that will represent the sequence in which composition operators are applied. Algebraic properties attached to a composition operator will ensure semantic equivalence of the result in case of transformations. For instance, the two equations depicted in EQ. 8.1 and EQ. 8.2 are equivalent **if** the $\cup$ operator is known to be commutative. Then, to formalize such transformations of equations, one can again formalize modifications made by algebraic properties as meta-transformations, working directly at the equation-level.

APPENDIX A

Official Docker guidelines

1. FROM command first: the FROM command must be the first to appear in a dockerfile
2. RUN Exec form: RUN commands have two syntaxes, one with brackets and one without. Interpretation of arguments differ from the two syntaxes. The one with brackets must be used.
3. Multiple CMD: CMD commands allows one to start a service when booting up a container. Docker allows only a single service to be specified, therefore multiple CMD are useless since only the last one will be run.
4. Provides default to CMD: One has to provide default parameter via CMD to start a service. If an EntryPoint command is specified, CMD and EntryPoint commands should be specified in JSON format.
5. Variables in exec form of CMD: Variables used in CMD commands in its exec form are not interpreted. CMD [“echo”, “$HOME”] won’t output the $HOME variable value.
6. Merge LABEL commands: When possible, merge labels commands.
7. Avoid apt-get upgrade: You should avoid RUN apt-get upgrade or dist-upgrade, as many of the “essential” packages from the base images will not upgrade inside an unprivileged container
8. Combine install with update: Always combine RUN apt-get update with apt-get install in the same RUN statement. Ommiting this can lead to unexpected behaviour since apt-get update can be not run.
9. Packages, version pinning: Always fully specify the version of the package to install.
10. FROM, version pinning: Always fully specify the version of the parent dockerfile to use (i.e., latest tag is therefore not permitted).
11. CMD exec form: CMD commands have two syntaxes, one with brackets and one without. Interpretation of arguments differ from the two syntaxes. The one with brackets must be used if parameters are specified.
12. Prefer COPY: Although ADD and COPY are functionally similar, generally speaking, COPY is preferred.

13. ADD <http> discouraged: Because image size matters, using ADD to fetch packages from remote URLs is strongly discouraged; you should use curl or wget.
14. User root discouraged: You should avoid installing or using sudo since it has unpredictable. TTY and signal-forwarding behavior that can cause more problems than it solves. If you absolutely need functionality similar to sudo (e.g., initializing the daemon as root but running it as non-root), you may be able to use gosu.
15. As few USER commands as possible: To reduce layers and complexity, avoid switching USER back and forth frequently.
16. WORKDIR must have absolute path: For clarity and reliability, you should always use absolute paths for your WORKDIR.
17. cd in RUN should be avoided: Do not use cd in RUN commands, use WORKDIR instead.
18. Sort installation alphanumerically: Installation of multiple softwares must be written in alphanumerical order.
19. Add –no-install-recommend: Add –no-install-recommend when installing with apt-get, this will avoid installation not explicitly specified.

Guidelines can be found in a our repository [1]. Guidelines 4 and 11 are not implemented since they are too domain-specific.

1. https://github.com/ttben/dockerconflict/tree/master/src/main/java/fr/unice/i3s/sparks/docker/core/guidelines

APPENDIX B

Collecting Dockerfiles

Our main requirement was to avoid **downloading** images since (i) a docker image can easily weight more than 500MB (the official version 3.5 of python image weights 680MB, the official java image weights 640MB and node 655MB) the amount of data to store would be too large, and (ii) a docker image does not contain all the information originally written by the user.

We first targeted the largest collections of docker files we known: the DockerHub. This hub hosts both official images (around 120 images) [1] and open non-official repositories (around 150 000 repositories [2]). This hub is based on a registry that lists all available images (and therefore, dockerfiles) [3] through a *catalogue* endpoint. This specification is currently not implemented by the docker company itself [4] therefore we can not list all available images or dockerfiles in the hub.

The second biggest source of dockerfiles was GitHub. We decided to crawl a set of dockerfiles from GitHub platform. Again, GitHub does not provide an API endpoint to list all files by type. We had to web-crawl dockerfiles as a physical user would do. Due to GitHub restrictions, one can not look for a specific type of file and has to specify information about the content of the dockerfile. This communal platform allowed us to perform more or less specific requests on the content of the *dockerfiles* and gave us a random sample of it. This has to be done by crawling, too. We use a chrome-extension to crawl github content. We perform requests on those kind of URLs to be able to find Dockerfile that, at least, contains a FROM code inside.

```
https://github.com/search?p=100&q=language%3ADockerfile+FROM
        &ref=searchresults&type=Code&utf8=%E2%9C%93
```

We then filter the result to delete duplicates since the API returned a random sample. In order to have a fair set of files, we iteratively looked for docker commands from the docker DSL. This way, we had a homogeneous amount of each docker commands and let statistics do the remaining work. Moreover, the parent-child

1. http://www.slideshare.net/Docker/dockercon-16-general-session-day-2-63497745
2. https://www.ctl.io/developers/blog/post/docker-hub-top-10/
3. http://54.71.194.30:4014/reference/api/docker-io_api/
4. https://github.com/docker/distribution/pull/653

relationship still needs to be established since the layer does not store explicitly the parent image ID. We manually retrieve parent's dockerfiles to cover half of our dataset [4].

Bibliography

[1] J. D. Cook. (Feb. 2011). LEGO blocks and organ transplants, [Online]. Available: https://www.johndcook.com/blog/2011/02/03/lego-blocks-and-organ-transplants/.

[2] « ISO/IEC/IEEE 24765:2017: Systems and software engineering — Vocabulary », International Organization for Standardization, Geneva, CH, Standard, 2017. [Online]. Available: https://www.iso.org/standard/71952.html.

[3] B. Benni, S. Mosser, N. Moha, and M. Riveill, « A delta-oriented approach to support the safe reuse of black-box code rewriters », *Journal of Software: Evolution and Process*, vol. 31, no. 8, e2208, 2019, e2208 smr.2208. DOI: 10.1002/smr.2208. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2208. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2208.

[4] B. Benni, S. Mosser, P. Collet, and M. Riveill, « Supporting Micro-services Deployment in a Safer Way: a Static Analysis and Automated Rewriting Approach », in *Symposium on applied Computing*, Pau, France, Apr. 2018. DOI: 10.1145/3167132.3167314. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01659776.

[5] B. Benni, S. Mosser, N. Moha, and M. Riveill, « A Delta-oriented Approach to Support the Safe Reuse of Black-box Code Rewriters », in *17th International Conference on Software Reuse (ICSR'18)*, Madrid, France, 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01722040.

[6] B. Benni, P. Collet, G. Molines, S. Mosser, and A.-M. Pinna-Déry, « Teaching DevOps at the Graduate Level: A report from Polytech Nice Sophia », in *First international workshop on software engineering aspects of continuous development and new paradigms of software production and deployment*, LASER foundation, Villebrumier, France, Mar. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01792773.

[7] H. D. Mills, « Structured Programming-Retrospect and Prospect », 1986.

[8] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, *et al.*, « Ultra-large-scale systems: The software challenge of the future », CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., 2006.

[9] E. W. Dijkstra, « On the role of scientific thought », in *Selected writings on computing: a personal perspective*, Springer, 1982, pp. 60–66.

[10] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004, ISBN: 0321245628.

[11] A. Zito, Z. Diskin, and J. Dingel, « Package merge in uml 2: Practice vs. theory? », in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2006, pp. 185–199.

[12] M. J. Rochkind, « The source code control system », *IEEE transactions on Software Engineering*, no. 4, pp. 364–370, 1975.

[13] D. L. Parnas, « On the criteria to be used in decomposing systems into modules », *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[14] ——, « Software engineering or methods for the multi-person construction of multi-version programs », in *IBM Germany Scientific Symposium Series*, Springer, 1974, pp. 225–235.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, « An overview of AspectJ », in *European Conference on Object-Oriented Programming*, Springer, Sep. 2001, pp. 327–354. DOI: 10.1007/3-540-45337-7_18.

[16] C. Cecchinel, « DEPOSIT : an approach to model and deploy data collection policies on heterogeneous and shared sensor networks », Theses, Université Côte d'Azur, Nov. 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01703857.

[17] J. C. Martin, *Introduction to Languages and the Theory of Computation*. McGraw-Hill NY, 1991, vol. 4.

[18] M. Sipser *et al.*, *Introduction to the Theory of Computation*. Thomson Course Technology Boston, 2006, vol. 2.

[19] J. Berstel, L. Boasson, O. Carton, and I. Fagnot, *Minimization of Automata*, 2010. eprint: arXiv:1010.5318. [Online]. Available: https://arxiv.org/abs/1010.5318v3.

[20] (2019). DFA minimization, [Online]. Available: https://en.wikipedia.org/wiki/DFA_minimization (visited on 09/20/2019).

[21] Docker. (2019). Docker Official website, [Online]. Available: https://www.docker.com/ (visited on 09/01/2019).

[22] ——, (2019). Docker Official Numbers, [Online]. Available: https://www.docker.com/company (visited on 09/10/2019).

[23] Git. (2019). Git Official Numbers, [Online]. Available: https://git-scm.com/ (visited on 09/10/2019).

[24] VentureBeat. (2018). Github Passes 100M Repositories, [Online]. Available: https://venturebeat.com/2018/11/08/github-passes-100-million-repositories/ (visited on 09/10/2019).

[25] Github. (2019). Github, State of the Octoverse, [Online]. Available: https://octoverse.github.com/ (visited on 09/10/2019).

[26] L. Foundation. (2017). 2017 Linux Kernel Report Highlights Developers' Roles and Accelerating Pace of Change, [Online]. Available: https://www.linuxfoundation.org/blog/2017/10/2017-linux-kernel-report-highlights-developers-roles-accelerating-pace-change/ (visited on 09/10/2019).

[27] L. Torvald. (2019). Linux Official Repository, [Online]. Available: https://github.com/torvalds/linux (visited on 09/10/2019).

[28] T. Lee. (2017). Android Now Has 2 Billion Monthly Active Users, [Online]. Available: https://www.ubergizmo.com/2017/05/android-2-billion-monthly-users/ (visited on 09/10/2019).

[29] D. Bohn. (2018). ANDROID AT 10: THE WORLD'S MOST DOMINANT TECHNOLOGY, [Online]. Available: https://www.theverge.com/2018/9/26/17903788/google-android-history-dominance-marketshare-apple (visited on 09/10/2019).

[30] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, « Tracking the Software Quality of Android Applications Along Their Evolution (T) », in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 236–247. DOI: 10.1109/ASE.2015.46.

[31] R. Pawlak, C. Noguera, and N. Petitprez, « Spoon: Program Analysis and Transformation in Java », Inria, Research Report RR-5901, 2006. [Online]. Available: https://hal.inria.fr/inria-00071366.

[32] M. Acher, P. Collet, P. Lahire, and R. France, « Composing Feature Models », in *Software Language Engineering*, M. van den Brand, D. Gašević, and J. Gray, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 62–81, ISBN: 978-3-642-12107-4. DOI: 10.1007/978-3-642-12107-4_6.

[33] L. C. Megginson, « Lessons from Europe for American business », *The Southwestern Social Science Quarterly*, pp. 3–13, 1963.

[34] I. Kurtev, « State of the art of QVT: A model transformation language standard », in *International Symposium on Applications of Graph Transformations with Industrial Relevance*, Springer, 2007, pp. 377–393.

[35] O. MOFM2T, « OMG MOF Model to Text Transformation Language (OMG MOFM2T) Version 1.0 », *Object Management Group. http://www. omg. org/spec/MOFM2T/1.0*, 2008.

[36] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez, « ATL: a QVT-like transformation language », in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ACM, 2006, pp. 719–720.

[37] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro, « VIATRA - visual automated transformations for formal verification and validation of UML models », in *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, Sep. 2002, pp. 267–270. DOI: 10.1109/ASE.2002.1115027.

[38] M. Lawley and J. Steel, « Practical declarative model transformation with Tefkat », in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2005, pp. 139–150.

[39] G. Mussbacher, D. Amyot, J. Araújo, and A. Moreira, « Requirements modeling with the aspect-oriented user requirements notation (AoURN): a case study », in *Transactions on aspect-oriented software development VII*, Springer, 2010, pp. 23–68.

[40] J. Klein, L. Hélouët, and J.-M. Jézéquel, « Semantic-based weaving of scenarios », in *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, R. E. Filman, Ed., Bonn, Germany: ACM, Mar. 2006, pp. 27–38. DOI: 10.1145/1119655.1119662. [Online]. Available: https://hal.inria.fr/hal-00921480.

[41] E. Merks, R. Eliersick, T. Grose, F. Budinsky, and D. Steinberg, « The eclipse modeling framework », *retrieved from, total*, p. 37, 2003.

[42] F. Fleurey, R. Reddy, R. France, B. Baudry, S. Ghosh, and M. Clavreul, *Kompose: a generic model composition tool*, 2005.

[43] A. Jackson, J. Klein, B. Baudry, and S. Clarke, « Executable Aspect Oriented Models for Improved Model Testing », in *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.*, Bilbao, Spain, Spain, 2006. [Online]. Available: https://hal.inria.fr/inria-00512544.

[44] E. Baniassad and S. Clarke, « Theme: an approach for aspect-oriented analysis and design », in *Proceedings. 26th International Conference on Software Engineering*, May 2004, pp. 158–167. DOI: 10.1109/ICSE.2004.1317438.

[45] D. Kolovos, R. Paige, and F. Polack, « Merging Models with the Epsilon Merging Language (EML) », vol. 4199, Oct. 2006, pp. 215–229. DOI: 10.1007/11880240_16.

[46] M. Schöttle, O. Alam, J. Kienzle, and G. Mussbacher, « On the modularization provided by concern-oriented reuse », in *MODULARITY*, 2016.

[47] B. Morin, J. Klein, O. Barais, and J.-M. Jézéquel, « A Generic Weaver for Supporting Product Lines », in *International Workshop on Early Aspects at ICSE'08*, Leipzig, Germany, Germany, 2008. [Online]. Available: https://hal.inria.fr/inria-00456485.

[48] M. Kramer, J. Klein, J. Steel, B. Morin, J. Kienzle, O. Barais, and J. Jézéquel, « On the formalisation of GeKo: A generic aspect models weaver », Technical Report, University of Luxembourg, Tech. Rep., 2012.

[49] J. Whittle and P. Jayaraman, « MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation », in *Models in Software Engineering*, H. Giese, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 16–27, ISBN: 978-3-540-69073-3.

[50] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg, « Models@runtime to support dynamic adaptation », *Computer*, vol. 42, no. 10, pp. 44–51, 2009.

[51] P. Sriplakich, X. Blanc, and M.-P. Gervais, « Collaborative Software Engineering on Large-scale models: Requirements and Experience in ModelBus », in *23rd Annual ACM Symposium on Applied Computing (SAC'08)*, Fortaleza, Ceará, Brazil: ACM, Mar. 2008, pp. 674–681. DOI: 10.1145/1363686.1363849. [Online]. Available: https://hal.inria.fr/hal-00668912.

[52] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, « SPOON: A library for implementing analyses and transformations of Java source code », *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, 2016. DOI: 10.1002/spe.2346. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2346. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2346.

[53] L. R. Rodriguez and J. Lawall, « Increasing Automation in the Backporting of Linux Drivers Using Coccinelle », ser. 11th European Dependable Computing Conference - Dependability in Practice, https://hal.inria.fr/hal-01213912, Paris, France, Nov. 2015.

[54] V. Matena, B. Stearns, and L. Demichiel, *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform*, 2nd ed. Pearson Education, 2003, ISBN: 0201914662.

[55] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin, « JAC : A Flexible and Efficient Framework for AOP in Java », in *Reflection*, The Third International Conference on Metalevel Architectures and Separation of Crosscuting Concerns, Kyoto, Japon, sept 2001, X, France, Jan. 2001. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01124645.

[56] (Oct. 2019). 2to3 - Automated Python 2 to 3 code translation, [Online]. Available: https://docs.python.org/2/library/2to3.html.

[57] F. Santacroce, *Git Essentials: Create, merge, and distribute code with Git, the most powerful and flexible versioning system available*. Packt Publishing Ltd, 2017.

[58] V. Berzins, « Software merge: Models and methods for combining changes to programs », in *ESEC '91*, A. van Lamsweerde and A. Fugetta, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 229–250, ISBN: 978-3-540-46446-4.

[59] V. Berzins and D. A. Dampier, « Software merge: Combining changes to decompositions », *Journal of Systems Integration*, vol. 6, no. 1, pp. 135–150, Mar. 1996, ISSN: 1573-8787. DOI: 10.1007/BF02262754. [Online]. Available: https://doi.org/10.1007/BF02262754.

[60] (2019). Patch, [Online]. Available: http://man7.org/linux/man-pages/man1/patch.1.html (visited on 09/30/2019).

[61] S. Mimram and C. D. Giusto, *A Categorical Theory of Patches*, 2013. eprint: 1311.3903v1. [Online]. Available: https://arxiv.org/abs/1311.3903v1.

[62] R. M. Stallman and Z. Weinberg, « The C preprocessor », *Free Software Foundation*, 1987.

[63] (2019). Patch, [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/ (visited on 09/30/2019).

[64] M. D. Ernst, G. J. Badros, and D. Notkin, « An empirical analysis of C preprocessor use », *IEEE Transactions on Software Engineering*, vol. 28, no. 12, pp. 1146–1170, 2002.

[65] E. W. Dijkstra *et al.*, *Notes on structured programming*, 1970.

[66] F. Baader and T. Nipkow, *Term rewriting and all that*. Cambridge university press, 1999.

[67] J. W. Klop and J. Klop, *Term rewriting systems*. Centrum voor Wiskunde en Informatica, 1990.

[68] P. Mishra and U. S. Reddy, « Declaration-free Type Checking », in *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '85, New Orleans, Louisiana, USA: ACM, 1985, pp. 7–21, ISBN: 0-89791-147-4. DOI: 10.1145/318593.318603. [Online]. Available: http://doi.acm.org/10.1145/318593.318603.

[69] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann, « Type Checking with Open Type Functions », *SIGPLAN Not.*, vol. 43, no. 9, pp. 51–62, Sep. 2008, ISSN: 0362-1340. DOI: 10.1145/1411203.1411215. [Online]. Available: http://doi.acm.org/10.1145/1411203.1411215.

[70] M. G. J. van den Brand, P. Klint, and J. J. Vinju, « Term Rewriting with Traversal Functions », *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 2, pp. 152–190, Apr. 2003, ISSN: 1049-331X. DOI: 10.1145/941566.941568. [Online]. Available: http://doi.acm.org/10.1145/941566.941568.

[71] G. Huet, « Conflunt reductions: Abstract properties and applications to term rewriting systems », in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, Oct. 1977, pp. 30–45. DOI: 10.1109/SFCS.1977.9.

[72] T. Aoto, J. Yoshida, and Y. Toyama, « Proving Confluence of Term Rewriting Systems Automatically », in *Rewriting Techniques and Applications*, R. Treinen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 93–102.

[73] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke, « Automated Termination Proofs with AProVE », in *Rewriting Techniques and Applications*, V. van Oostrom, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 210–220.

[74] N. DERSHOWITZ and J.-P. JOUANNAUD, « CHAPTER 6 - Rewrite Systems », in *Formal Models and Semantics*, ser. Handbook of Theoretical Computer Science, J. V. LEEUWEN, Ed., Amsterdam: Elsevier, 1990, pp. 243–320, ISBN: 978-0-444-88074-1. DOI: 10.1016/B978-0-444-88074-1.50011-1. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780444880741500111.

[75] T. Arts and J. Giesl, « Termination of term rewriting using dependency pairs », *Theoretical Computer Science*, vol. 236, no. 1, pp. 133–178, 2000, ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(99)00207-8. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397599002078.

[76] C. S. Lee, N. D. Jones, and A. M. Ben-Amram, « The Size-change Principle for Program Termination », in *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '01, London, United Kingdom: ACM, 2001, pp. 81–92, ISBN: 1-58113-336-7. DOI: 10.1145/360204.360210. [Online]. Available: http://doi.acm.org/10.1145/360204.360210.

[77] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, « Documenting and Automating Collateral Evolutions in Linux Device Drivers », *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 4, pp. 247–260, Apr. 2008, ISSN: 0163-5980. DOI: 10.1145/1357010.1352618. [Online]. Available: http://doi.acm.org/10.1145/1357010.1352618.

[78] G. Sittampalam *et al.*, « Some properties of darcs patch theory », *Available fr om http://urchin. earth. li/darcs/ganesh/darcs-patch-theory/theory/formal. pdf*, 2005.

[79] J. Dagit, « Type-correct changes—a safe approach to version control implementation », 2009.

[80] X. Blanc, I. Mounier, A. Mougenot, and T. Mens, « Detecting model inconsistency through operation-based model construction », in *2008 ACM/IEEE 30th International Conference on Software Engineering*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds., ACM, May 2008, pp. 511–520, ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368158. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368158.

[81] Y. Padioleau, J. L. Lawall, and G. Muller, « Understanding Collateral Evolution in Linux Device Drivers », *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 59–71, Apr. 2006, ISSN: 0163-5980. DOI: 10.1145/1218063.1217942. [Online]. Available: http://doi.acm.org/10.1145/1218063.1217942.

[82] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, « Runtime Monitoring of Software Energy Hotspots », in *ASE - The 27th IEEE/ACM International Conference on Automated Software Engineering - 2012*, Essen, Germany, Sep. 2012, pp. 160–169. DOI: 10.1145/2351676.2351699. [Online]. Available: https://hal.inria.fr/hal-00715331.

[83] A. Noureddine, R. Rouvoy, and L. Seinturier, « Monitoring Energy Hotspots in Software », *Journal of Automated Software Engineering*, vol. 22, no. 3, pp. 291–332, Sep. 2015. [Online]. Available: https://hal.inria.fr/hal-01069142.

[84] A. Bourdon, A. Noureddine, R. Rouvoy, and L. Seinturier, « PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level », *ERCIM News*, Special Theme: Smart Energy Systems, vol. 92, ERCIM, Ed., pp. 43–44, Jan. 2013. [Online]. Available: https://hal.inria.fr/hal-00772454.

[85] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien, « Detecting Antipatterns in Android Apps », in *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '15, Florence, Italy: IEEE Press, 2015, pp. 148–149, ISBN: 978-1-4799-1934-5. [Online]. Available: http://dl.acm.org/citation.cfm?id=2825041.2825078.

[86] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, « Fine-grained and accurate source code differencing », in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. DOI: 10.1145/2642937.2642982. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982.

[87] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, « Investigating the energy impact of Android smells », in *IEEE 24th Int. Conf. on Software Analysis, Evolution and Reengineering, Klagenfurt, Austria, February 20-24*, S. 2017, Ed., 2017, pp. 115–126. DOI: 10.1109/SANER.2017.7884614. [Online]. Available: https://doi.org/10.1109/SANER.2017.7884614.

[88] G. Hecht, N. Moha, and R. Rouvoy, « An empirical study of the performance impacts of Android code smells », in *Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14-22, 2016*, ser. MOBILESoft '16, Austin, Texas: ACM, 2016, pp. 59–69, ISBN: 978-1-4503-4178-3. DOI: 10.1145/2897073.2897100. [Online]. Available: http://doi.acm.org/10.1145/2897073.2897100.

[89] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. " O'Reilly Media, Inc.", 2016.

[90] A. Balalaie, A. Heydarnoori, and P. Jamshidi, « Microservices architecture enables DevOps: migration to a cloud-native architecture », *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[91] DevOps.com and ClusterHQ, *Container market adoption - Survey 2016*, https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf, Jun. 2016.

[92] C. Boettiger, « An Introduction to Docker for Reproducible Research », *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015, ISSN: 0163-5980. DOI: 10.1145/2723872.2723882. [Online]. Available: http://doi.acm.org/10.1145/2723872.2723882.

[93] D. Merkel, « Docker: Lightweight Linux Containers for Consistent Development and Deployment », *Linux J.*, vol. 2014, no. 239, Mar. 2014, ISSN: 1075-3583. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[94] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, « Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments », in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb. 2013, pp. 233–240. DOI: 10.1109/PDP.2013.41.

[95] R. Peinl, F. Holzschuher, and F. Pfitzer, « Docker Cluster Management for the Cloud - Survey Results and Own Solution », *Journal of Grid Computing*, vol. 14, no. 2, pp. 265–282, 2016, ISSN: 1572-9184. DOI: 10.1007/s10723-016-9366-y. [Online]. Available: http://dx.doi.org/10.1007/s10723-016-9366-y.

[96] R. Morabito, J. Kjällman, and M. Komu, « Hypervisors vs. Lightweight Virtualization: A Performance Comparison », in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, Mar. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[97] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, « An updated performance comparison of virtual machines and Linux containers », in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[98] G. Rushgrove, *DockerCon16 - The Dockerfile Explosion and the Need for Higher Level Tools by Gareth Rushgrove*, https://goo.gl/86XPrq, Jun. 2016.

[99] W. McNelly. (1969). Interview of Frank Herbert: 'Herbert's science fiction novels, "Dune" and "Dune Messiah"', [Online]. Available: http://www.sinanvural.com/seksek/inien/tvd/tvd2.htm.

[100] T. Mens, « A state-of-the-art survey on software merging », *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, May 2002. DOI: 10.1109/TSE.2002.1000449.

[101] V. Berzins, « Software Merge: Semantics of Combining Changes to Programs », *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1875–1903, Nov. 1994, ISSN: 0164-0925. DOI: 10.1145/197320.197403. [Online]. Available: http://doi.acm.org/10.1145/197320.197403.

[102] Jackson and Ladd, « Semantic Diff: a tool for summarizing the effects of modifications », in *Proceedings 1994 International Conference on Software Maintenance*, Sep. 1994, pp. 243–252. DOI: 10.1109/ICSM.1994.336770.

[103] J. Buffenbarger, « Syntactic software merging », in *Software Configuration Management*, J. Estublier, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 153–172, ISBN: 978-3-540-47768-6.

[104] T. Apiwattanapong, A. Orso, and M. J. Harrold, « JDiff: A differencing technique and tool for object-oriented programs », *Automated Software Engineering*, vol. 14, no. 1, pp. 3–36, Mar. 2007, ISSN: 1573-7535. DOI: 10.1007/s10515-006-0002-0. [Online]. Available: https://doi.org/10.1007/s10515-006-0002-0.

[105] N. Niu, S. Easterbrook, and M. Sabetzadeh, « A category-theoretic approach to syntactic software merging », in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Sep. 2005, pp. 197–206. DOI: 10.1109/ICSM.2005.6.

[106] O. Leßenich, S. Apel, and C. Lengauer, « Balancing precision and performance in structured merge », *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, Sep. 2015, ISSN: 1573-7535. DOI: 10.1007/s10515-014-0151-5. [Online]. Available: https://doi.org/10.1007/s10515-014-0151-5.

[107] G. Cavalcanti, P. Borba, and P. Accioly, « Evaluating and Improving Semistructured Merge », *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 59:1–59:27, Oct. 2017, ISSN: 2475-1421. DOI: 10.1145/3133883. [Online]. Available: http://doi.acm.org/10.1145/3133883.

[108] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, « Semistructured Merge: Rethinking Merge in Revision Control Systems », in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, Szeged, Hungary: ACM, 2011, pp. 190–200, ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025141. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025141.

[109] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen, « Indicators for merge conflicts in the wild: survey and empirical study », *Automated Software Engineering*, vol. 25, no. 2, pp. 279–313, Jun. 2018, ISSN: 1573-7535. DOI: 10.1007/s10515-017-0227-0. [Online]. Available: https://doi.org/10.1007/s10515-017-0227-0.

[110] J. Eyolfson, L. Tan, and P. Lam, « Do Time of Day and Developer Experience Affect Commit Bugginess? », in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11, Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 153–162, ISBN: 978-1-4503-0574-7. DOI: 10.1145/1985441.1985464. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985464.

[111] P. Accioly, P. Borba, and G. Cavalcanti, « Understanding semi-structured merge conflict characteristics in open-source Java projects », *Empirical Software Engineering*, vol. 23, no. 4, pp. 2051–2085, Aug. 2018, ISSN: 1573-7616. DOI: 10.1007/s10664-017-9586-1. [Online]. Available: https://doi.org/10.1007/s10664-017-9586-1.

[112] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, « How Do Centralized and Distributed Version Control Systems Impact Software Changes? », in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: ACM, 2014, pp. 322–333, ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568322. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568322.

[113] M. L. Guimarães and A. R. Silva, « Improving early detection of software merge conflicts », in *2012 34th International Conference on Software Engineering (ICSE)*, Jun. 2012, pp. 342–352. DOI: 10.1109/ICSE.2012.6227180.

[114] T. Ziegler, « GITCoP: A Machine Learning Based Approach to Predicting Merge Conflicts from Repository Metadata », PhD thesis, University of Passau, 2017.

[115] G. G. L. Menezes, L. G. P. Murta, M. O. Barros, and A. Van Der Hoek, « On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub », *IEEE Transactions on Software Engineering*, pp. 1–1, 2018. DOI: 10.1109/TSE.2018.2871083.

[116] M. Acher, P. Collet, P. Lahire, and R. B. France, « Familiar: A domain-specific language for large scale management of feature models », *Science of Computer Programming*, vol. 78, no. 6, pp. 657–681, 2013.

[117] E. Silva, A. Leite, V. Alves, and S. Apel, « ExpRunA : a domain-specific approach for technology-oriented experiments », *Software & Systems Modeling*, Aug. 2019, ISSN: 1619-1374. DOI: 10.1007/s10270-019-00749-6. [Online]. Available: https://doi.org/10.1007/s10270-019-00749-6.