Challenges to Support Scalable Software Composition

Benjamin Benni, Sébastien Mosser and Michel Riveill

{benni,mosser,riveill}@i3s.unice.fr
Université Côte d'Azur, CNRS, I3S, France

Abstract. Software systems became so complex that the need to decompose them into simpler, more manageable pieces became crucial. Because of this, one has to compose every isolated pieces to build the expected system. Thus, composition is a mechanism used in many different domains developed from scratch through custom composition operators. Therefore, nowadays composition use-cases are developed in isolation from each other and do not reuse common mechanisms nor common abstractions. Do these composition mechanisms and operators can be shared and reused across (other) domains? Is there any common abstractions behind the composition in itself? This Ph.D. thesis, started in 2016, explores different composition examples, taken from real-life use-cases, to explore these questions.

1 Introduction

Ultra large-scale systems are the new de-facto standard for software developpers [3]. Those systems are so complex that the need to focus on specific parts becomes crucial. When facing such complexity, the approach remains the same: one has to cut a large problem into smaller ones until each sub-problem is known to be solvable. Then, one has to compose every sub-solution to obtain the complete, full and sound problem solution. Even though the composition paradigm is used in many different domains, this composition process is developed and implemented each time in a domain-specific way.

2 Background and Challenges

In order to stress out the scaling issues that the composition faces, we draw a parallel between composition and cloud computing. In cloud computing, a system must scale according to two dimensions: horizontally and vertically. The ability of a system to accept new machines to face and absorb a large set of client-requests is called the horizontal scaling. The vertical scaling is the ability to make a specific machine grow (in terms of memory, processing power, or space disk) to locally absorb a peak of load without interfering with other parts of the system. With respect to composition paradigm, horizontal scaling represents the ability to support the definition of new composition operators

in many different domains. Composition vertical scaling is represented by the ability for a application domain to reuse pre-defined abstractions, speeding up its development.

Two different approaches exist to support the implementation of software composition operators: a generic one, and domain-specific solutions. Both approaches do not scale for different reasons.

On one hand, the generic approach scales horizontally: by definition every domain can adopt it. However, it does not scale vertically since it usually involves non-polynomial algorithms that do not scale given the size of current real-life industrial use cases. For example, any model can be represented as a graph. When composing two models (*i.e* merging two graphs), one must check if there are identical nodes between the two input models. Checking if a model (*i.e.* a graph) is a subset of another one, is a sub-graph isomorphism problem, triggering scalability issue related to the model size.

On the other hand, a domain-specific solution uses tailored-made or heuristic based algorithms [5], [2] to scale vertically. When using such a solution, adding a new domain is not possible, and reuse is not common, thus this approach does not scale horizontally in terms of development and maintenance costs.

To quickly integrate composition operators of new domains, we need to heavily reuse previous efforts and share common composition-based components. Thus, we need to support domain-specific composition operators that scales in terms of development cost and response time (*c1*); and to be able to ease the addition of new application domains without interfering with other composition use-cases (*c2*). In the next section we will present two domains that use composition mechanisms. We will show the commonalities that can emerge when studying composition.

3 Explored Application Domains

M4S (for *Modeling for Scaling*) is a research project that aims at maintaining an international network of experts, from different communities, working with or defining software composition operators. In this section, we take two domains, part of the outcome of the M4S project: (i) building of artefacts in container-based applications and (ii) interactions in anti-pattern fixes. We deliberately took two examples as distant as possible from each other to underline that composition operators are not domain-dependent.

Docker is the industrial de-facto standard for software and service deployment. It is a platform that allows one to ship and run a custom service through the use of images and containers. An extension mechanism allows developers to extend a black-box artefact from another one. Conflicts can occur with such black-box extension, leading to anti-patterns or unexpected behaviors. For example, a developer can install the python executable in its `3.5` version, while its parent artefact relies on the incompatible `2.7` version. This version incompatibility leads the parent artefact to fail, leading to continuous deployment error.

Android is a mobile operating system that runs on more than 1.5 billion of devices. This popularity arouses mobile-app developers attractiveness, thus generates a growing number of Android applications with an important need to shorten time-to-market. In this context, anti-patterns can occur and can be harmful to the final application. Anti-patterns are bad practices that affect maintenance, evolution, or power-consumption of a software. They can be automatically detected [4], finally producing a set of fixes to correct them. Each fix, correcting a specific anti-pattern, modify a portion of code. Fixes may be not independents as multiple anti-patterns can be detected at the very same place in the source code. In order to avoid non-deterministic correction, one needs to detect such concurrent modifications and to analyse fixes application.

In both example depicted above, an interference detection mechanism and a tracing system, among other things, would be useful. Both examples need to check interferences either between software installations or anti-patterns fixes. In case of detected conflict, a tracing system may be used to trace the original issue, (*e.g.* the two Docker artefacts installing incompatible software versions). Currently, these two needs would be implemented in isolation in two completely different domains and no sharing nor common abstraction would have emerged.

4 A Framework Approach

We have shown that at least two common needs emerged: interference detection and error tracing. The core idea of this Ph.D. thesis is to allow domain-experts to share development efforts by formally defining abstract components they can reuse, aligned to common needs in composition domain. This contribution targets developers of systems using composition operators (*e.g.* Docker or Android) and not the final user (*i.e.* developer actually using Docker or Android application).

The examples of composition given in the previous section are fully domain-specific, all the operations and abstractions are strongly linked to the targeted domain. In the given examples, interference detection and tracing system would be two domain-agnostic reusable off-the-shelf components that developers of Docker or Android platforms could use to support their work and help them focus on their domain of expertise (Fig.1). Such component-based approach needs to be based on common abstractions. These abstractions should represent either a Docker composition, an anti-pattern fixes or any other composition operator.

Docker is based on a sequence of operations that must be executed to build and run an artefact. Fixing an anti-pattern means to apply a sequence of actions to repair the initial source code. Thus, a common abstraction, *i.e.* a pivot-model, can emerge. We propose to use action-based modeling and reasoning techniques as pivot model. Action-based model, illustrated in Fig. 2, replaces any model (in the broadest sense of the term) by a sequence of actions that allows one to build it [1]. Scaling is supported, among other things, by incremental checking and parallelization allowed by action-based reasoning. Therefore, we can abstract docker extension mechanism and anti-patterns fixes as actions, and base the interference and tracing operators on it.

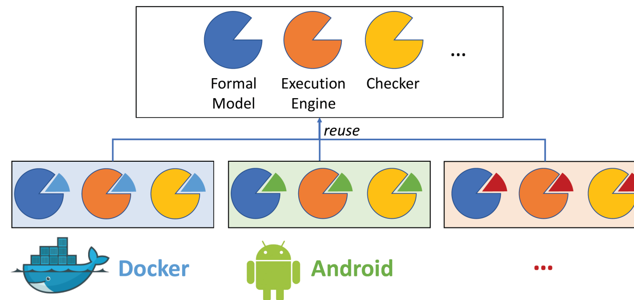


Fig. 1. Framework approach with reusable components

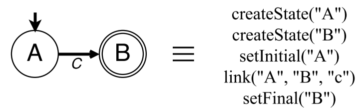


Fig. 2. Action-Based pivot model

Such component-based approach will allow one that wants to work with composition (*e.g.* developers of Docker or Android platforms) to easily reuse previous work. Thus, this approach match the horizontal and vertical scaling requirements. Because action-based model is known to scale well [1], using it as pivot-model will also support the vertical scaling requirement.

References

[1] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 511–520, New York, NY, USA, 2008. ACM.

[2] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

[3] Richard P. Gabriel, Linda Northrop, Douglas C. Schmidt, and Kevin Sullivan. Ultra-large-scale systems. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 632–634, New York, NY, USA, 2006. ACM.

[4] Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Detecting Antipatterns in Android Apps. Research Report RR-8693, INRIA Lille ; INRIA, March 2015.

[5] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring uml models. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, «UML» '01, pages 134–148, London, UK, UK, 2001. Springer-Verlag.

Scaling software composition

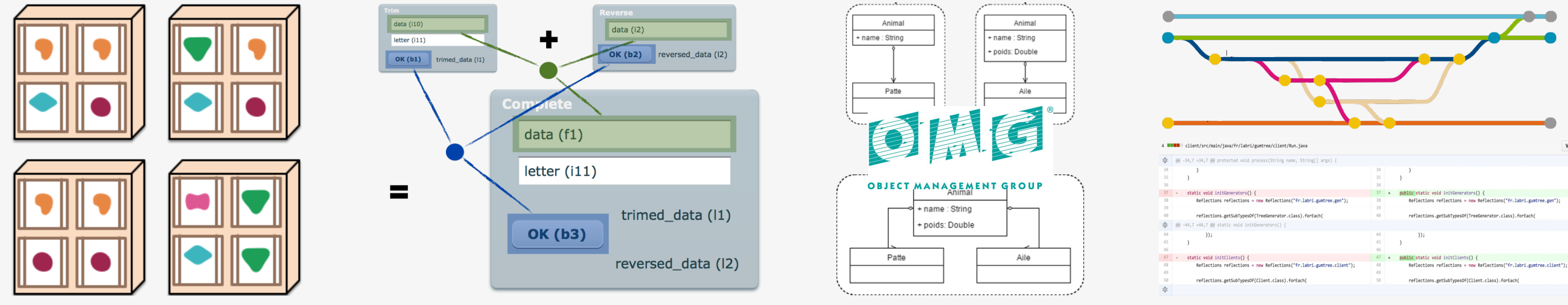
Benjamin benni www.i3s.unice.fr/~benni benni@i3s.unice.fr Summer 2017

Background

Ultra Large Scale Systems

Divide to conquer

Decompose to handle



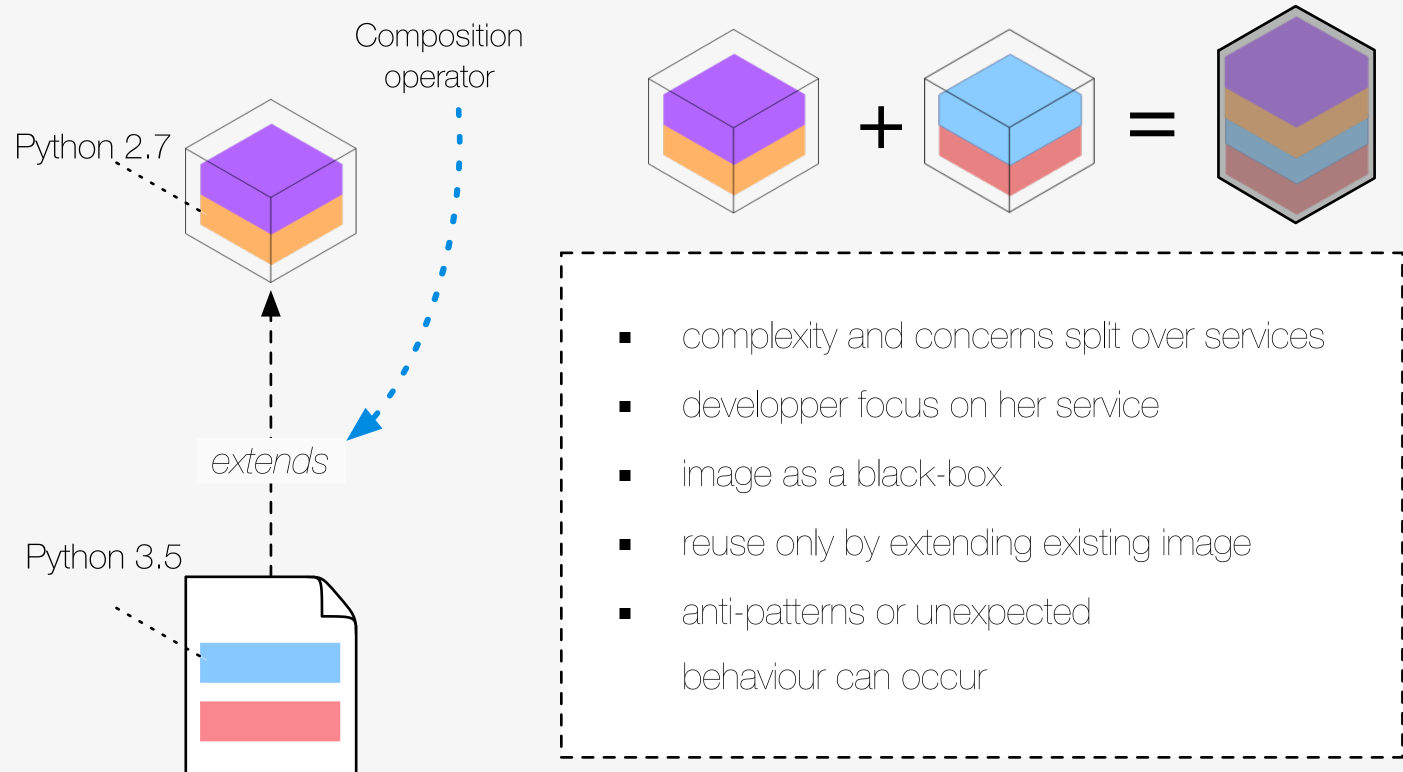
Different compositions
Domain-specific compositions
Common mechanisms
Common needs

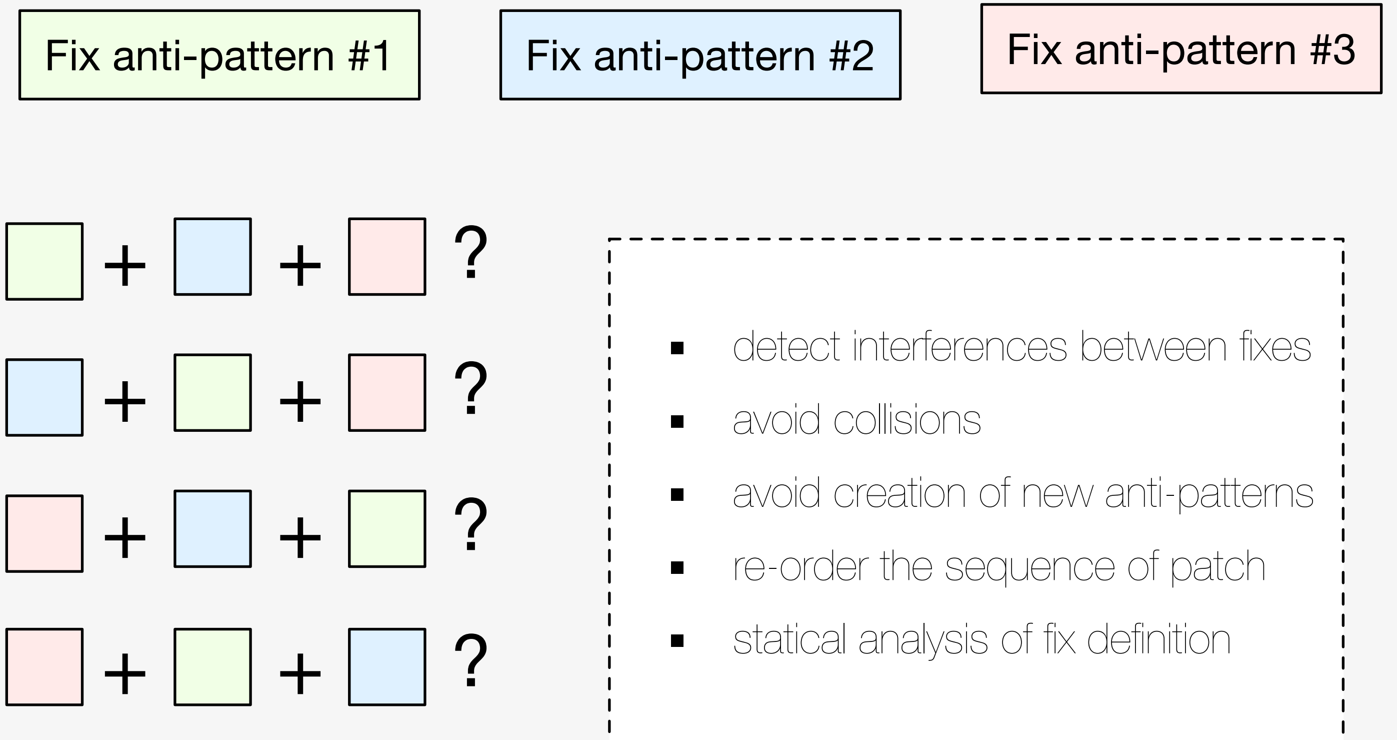
How common are composition mechanisms?

Is there any common abstraction?

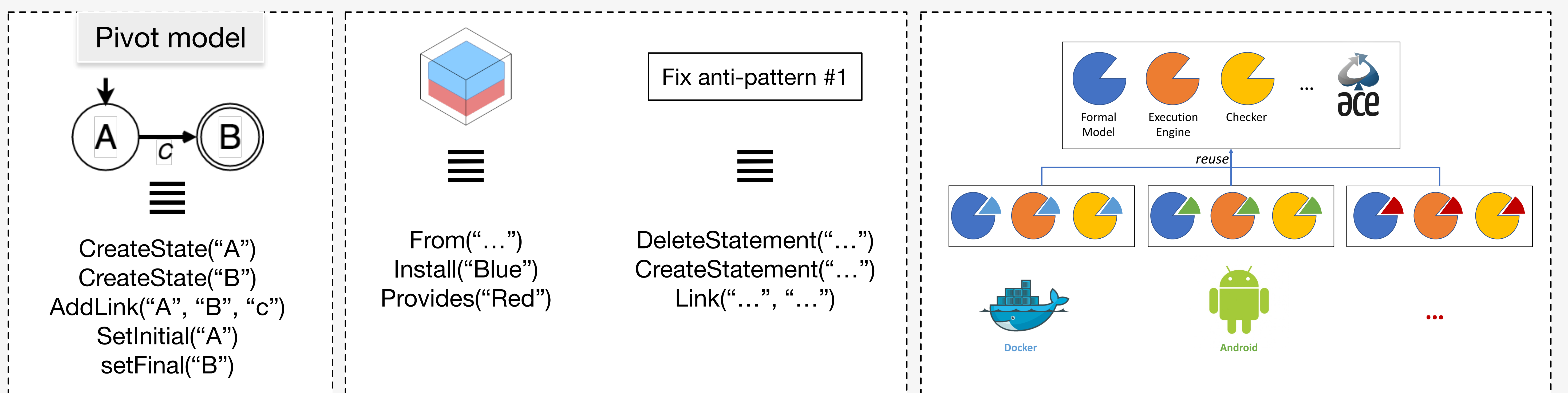
Is there any common operation?

Use-cases





Challenges



Related work

Ultra-Large-Scale Systems: The Software Challenge of the Future – 2006 report.Sponsored by the United States Department of Defense

Operation-based merging - Ernst Lippe and Norbert van Oosterom. 1992. In Proceedings of the fifth ACM SIGSOFT symposium on Software development environments (SDE 5). ACM, New York, NY, USA

Detecting model inconsistency through operation-based model construction - Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. 2008. In Proceedings of the 30th international conference on Software engineering (ICSE '08). ACM, New York, NY, USA, 511-520.

Impact on vertical scaling
Infer abstractions
Infer common operator

Laboratoire i3s, Université Côte d'Azur

Directed by Michel Riveill
Supervised by Sébastien Mosser

i3s sophia antipolis
UNIVERSITÉ CÔTE D'AZUR
cnrs