Structural Change Distilling of Ansible Roles

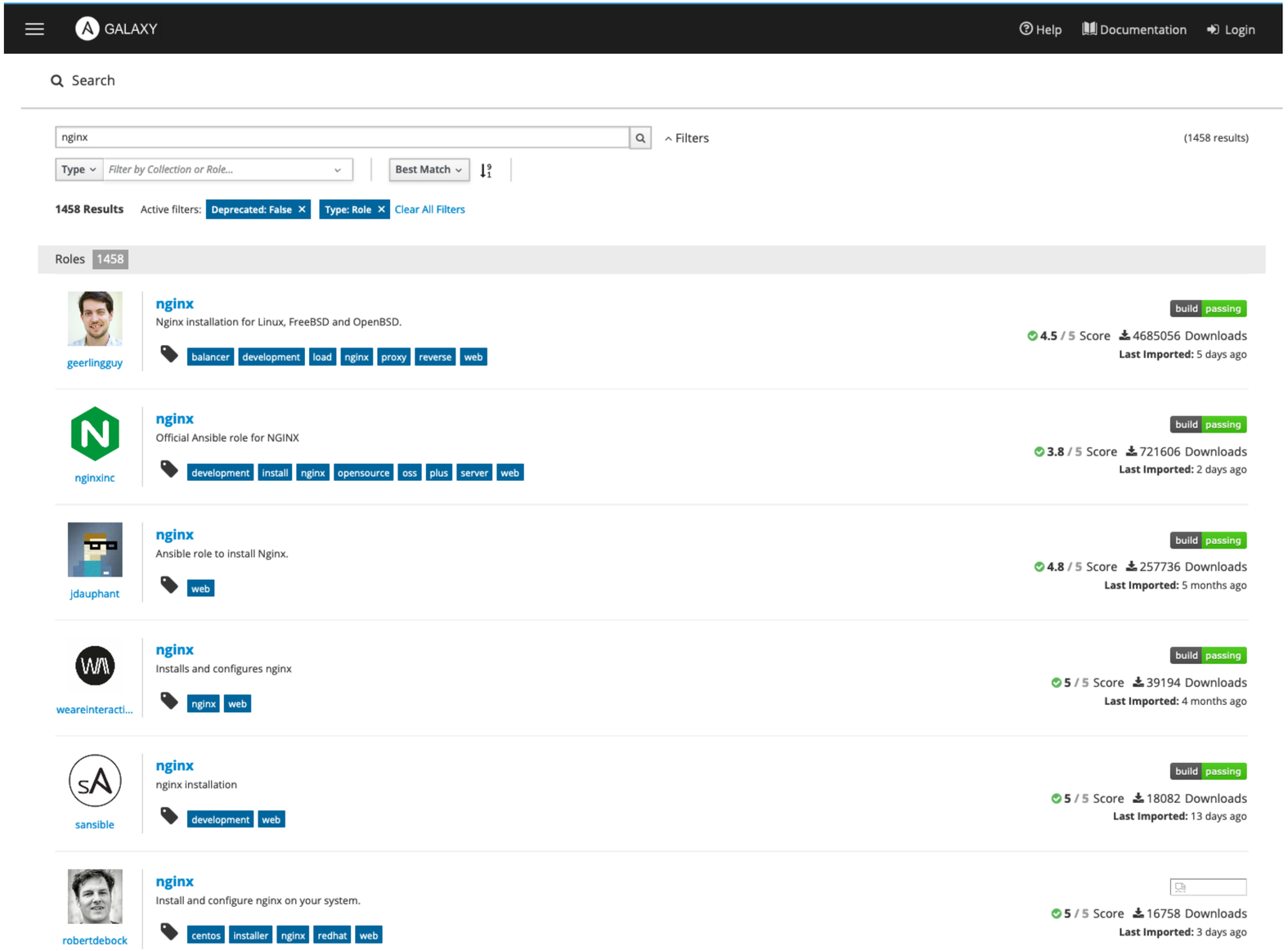
Presentation Abstract

Ruben Opdebeeck	*ropdebee@vub.be*
Ahmed Zerouali	*azeroual@vub.be*
Camilo Velázquez-Rodríguez	*cavelazq@vub.be*
Coen De Roover	*cderoove@vub.be*

VRIJE UNIVERSITEIT BRUSSEL

Software Languages Lab

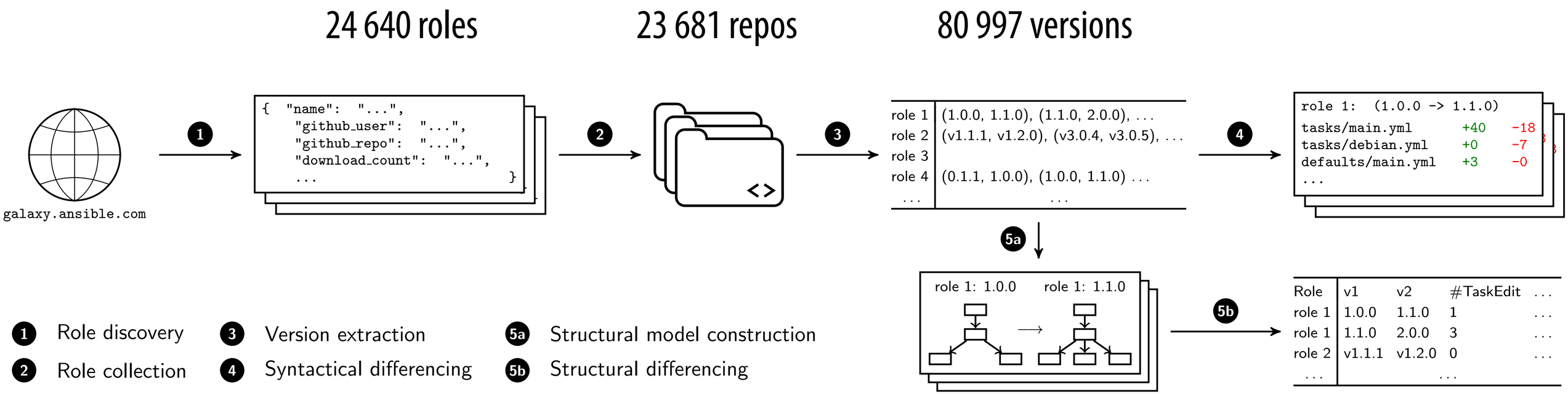
Ansible Galaxy



Ecosystem of roles
"Ansible's maven"

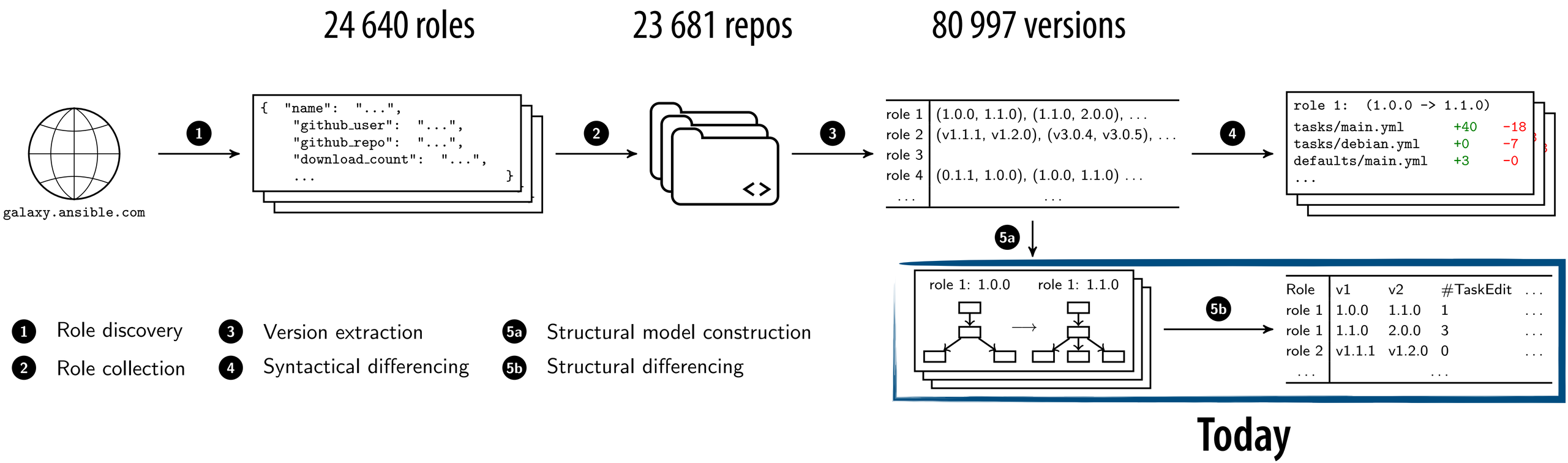
25k+ roles
~6k authors

Semantic Versioning in Ansible Roles



R. Opdebeeck, A. Zeraouli, C. Velázquez-Rodríguez, C. De Roover. "Does Infrastructure as Code Adhere to Semantic Versioning? An Analysis of Ansible Role Evolution", In Proc. 20th International Working Conference on Source Code Analysis and Manipulation, 2020.

Semantic Versioning in Ansible Roles



1 Role discovery
2 Role collection
3 Version extraction
4 Syntactical differencing
5a Structural model construction
5b Structural differencing

R. Opdebeeck, A. Zeraouli, C. Velázquez-Rodríguez, C. De Roover. "Does Infrastructure as Code Adhere to Semantic Versioning? An Analysis of Ansible Role Evolution", In Proc. 20th International Working Conference on Source Code Analysis and Manipulation, 2020.

Ansible Roles

Reusable Collections of Tasks

```yaml
# roles/nginx/tasks/main.yml
---
- name: Ensure nginx is installed
  apt:
    name: nginx
    state: present
- name: Ensure nginx configuration is present
  file:
    path: "{{ item }}"
    dest: "{{ nginx_dir }}"
  loop: "{{ conf_files }}"
```

```yaml
# roles/nginx/defaults/main.yml
---
conf_files:
  - files/nginx.conf
```

```yaml
# roles/nginx/vars/main.yml
---
nginx_conf_directory: /etc/nginx
```

```yaml
# roles/nginx/meta/main.yml
---
galaxy_info:
  role_name: nginx
  author: ROpdebee
  description: Installs nginx
  license: "GPL3"
  min_ansible_version: 2.4
  platforms:
    - name: Debian
      versions:
        - all
```

Ansible Roles

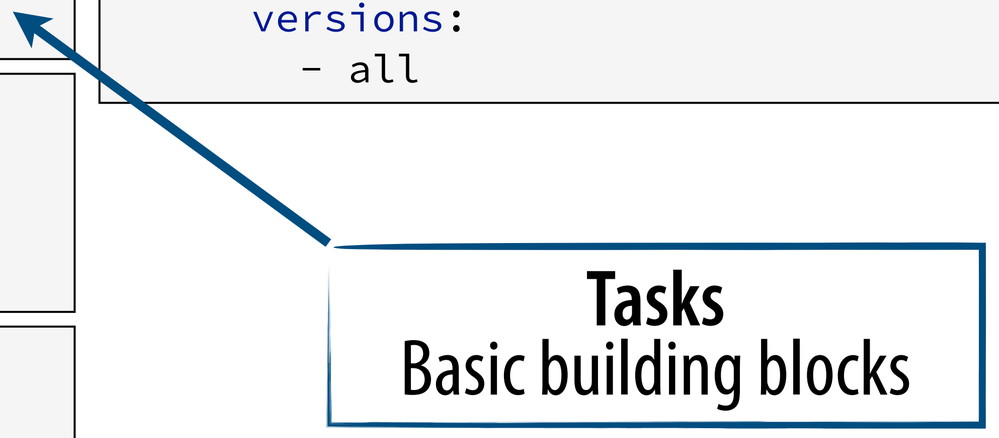
Reusable Collections of Tasks

```yaml
# roles/nginx/tasks/main.yml
---
- name: Ensure nginx is installed
  apt:
    name: nginx
    state: present
- name: Ensure nginx configuration is present
  file:
    path: "{{ item }}"
    dest: "{{ nginx_dir }}"
  loop: "{{ conf_files }}"
```

```yaml
# roles/nginx/defaults/main.yml
---
conf_files:
  - files/nginx.conf
```

```yaml
# roles/nginx/vars/main.yml
---
nginx_conf_directory: /etc/nginx
```

```yaml
# roles/nginx/meta/main.yml
---
galaxy_info:
  role_name: nginx
  author: ROpdebee
  description: Installs nginx
  license: "GPL3"
  min_ansible_version: 2.4
  platforms:
    - name: Debian
      versions:
        - all
```



Ansible Roles

Reusable Collections of Tasks

```yaml
# roles/nginx/tasks/main.yml
---
- name: Ensure nginx is installed
  apt:
    name: nginx
    state: present
- name: Ensure nginx configuration is present
  file:
    path: "{{ item }}"
    dest: "{{ nginx_dir }}"
  loop: "{{ conf_files }}"
```

```yaml
# roles/nginx/defaults/main.yml
---
conf_files:
  - files/nginx.conf
```

```yaml
# roles/nginx/vars/main.yml
---
nginx_conf_directory: /etc/nginx
```

```yaml
# roles/nginx/meta/main.yml
---
galaxy_info:
  role_name: nginx
  author: ROpdebee
  description: Installs nginx
  license: "GPL3"
  min_ansible_version: 2.4
  platforms:
    - name: Debian
      versions:
        - all
```

Ansible Roles

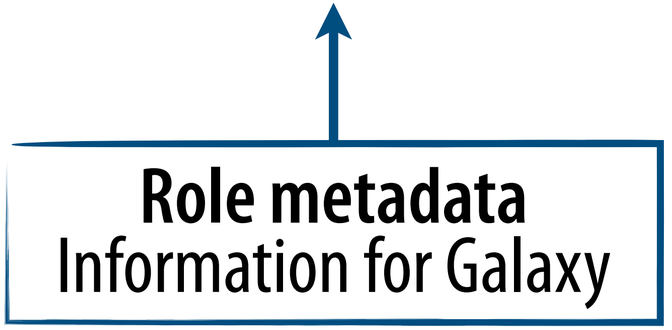
Reusable Collections of Tasks

```yaml
# roles/nginx/tasks/main.yml
---
- name: Ensure nginx is installed
  apt:
    name: nginx
    state: present
- name: Ensure nginx configuration is present
  file:
    path: "{{ item }}"
    dest: "{{ nginx_dir }}"
  loop: "{{ conf_files }}"
```

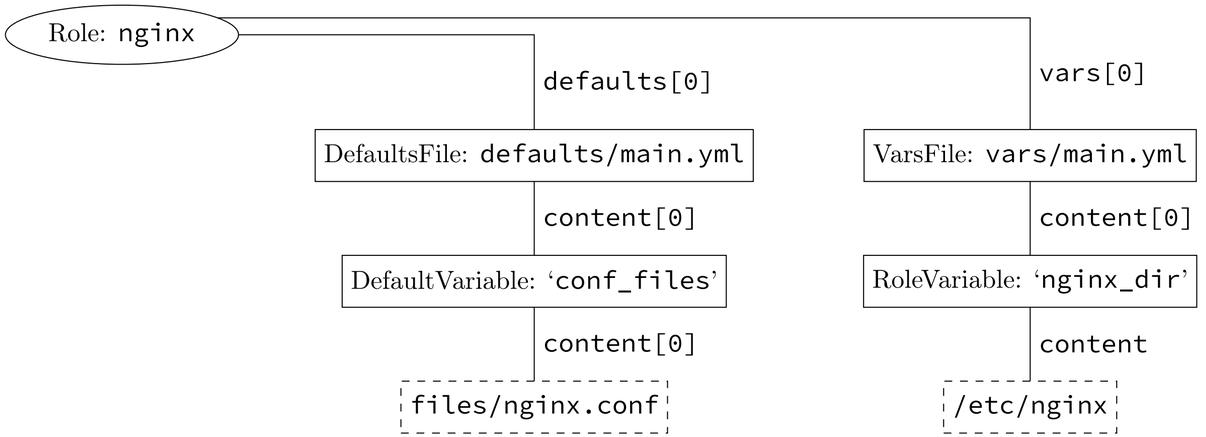
```yaml
# roles/nginx/defaults/main.yml
---
conf_files:
  - files/nginx.conf
```

```yaml
# roles/nginx/vars/main.yml
---
nginx_conf_directory: /etc/nginx
```

```yaml
# roles/nginx/meta/main.yml
---
galaxy_info:
  role_name: nginx
  author: ROpdebee
  description: Installs nginx
  license: "GPL3"
  min_ansible_version: 2.4
  platforms:
    - name: Debian
      versions:
        - all
```



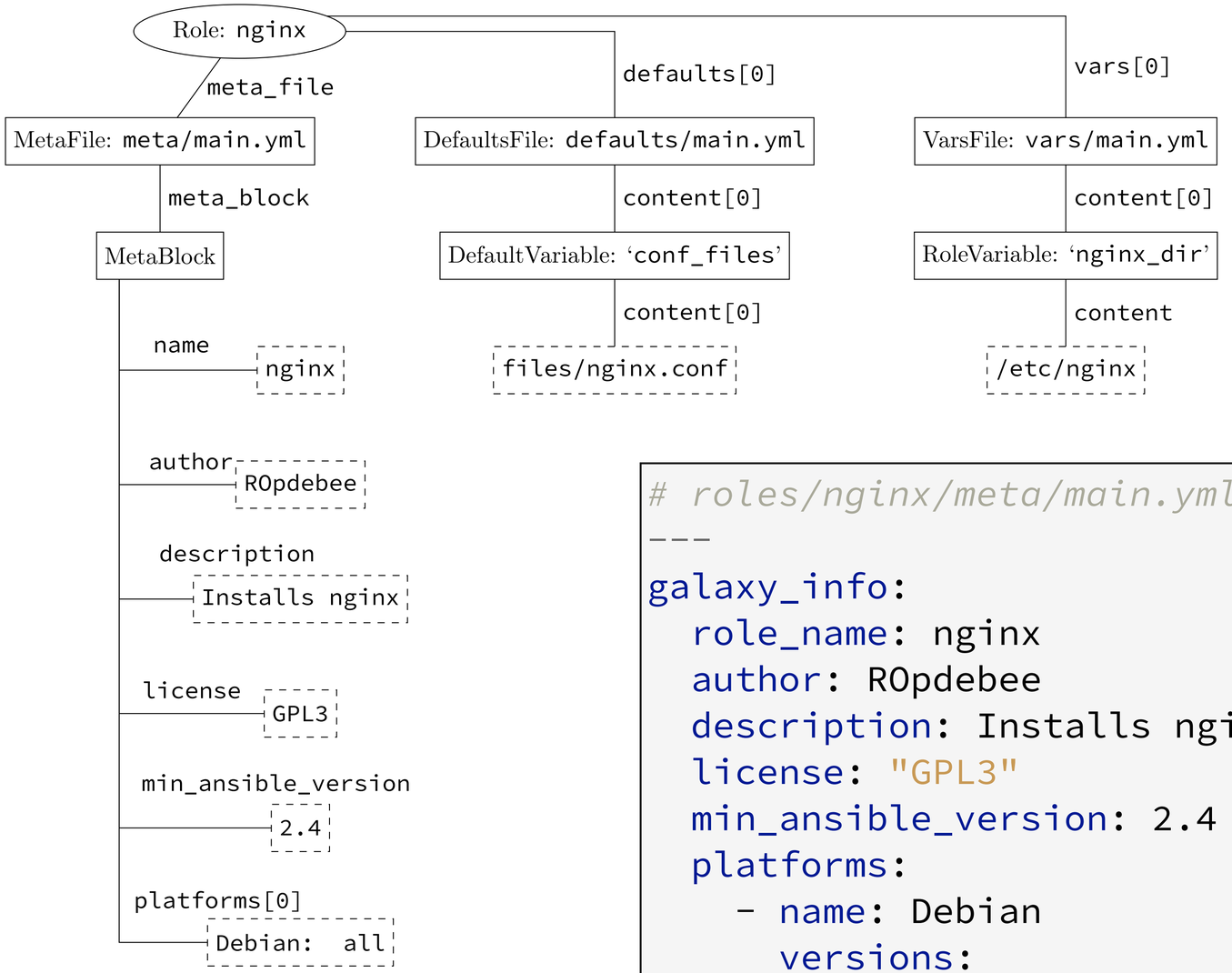
Structural Representation of Ansible Roles




```yaml
# roles/nginx/defaults/main.yml
---
conf_files:
  - files/nginx.conf
```

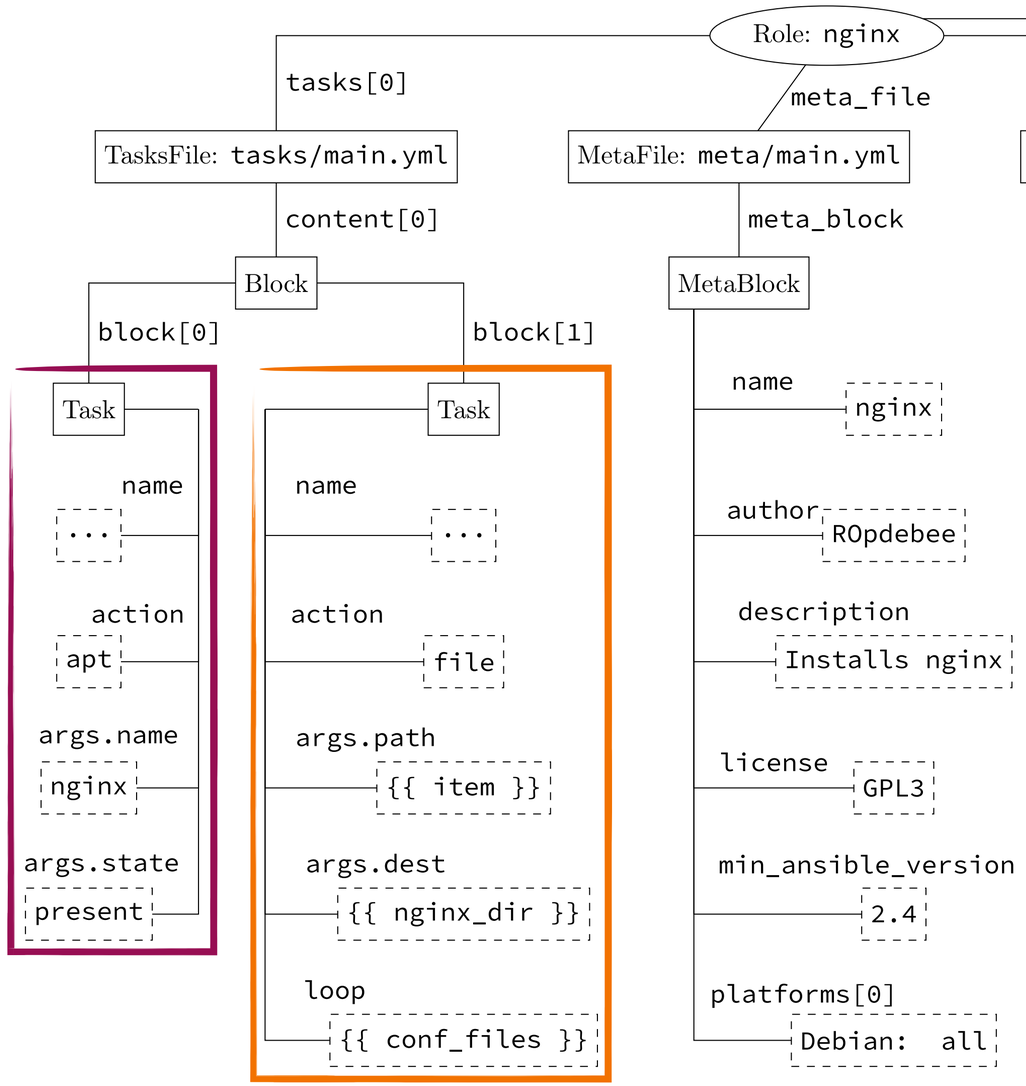
```yaml
# roles/nginx/vars/main.yml
---
nginx_conf_directory: /etc/nginx
```

Structural Representation of Ansible Roles




```yaml
# roles/nginx/meta/main.yml
---
galaxy_info:
  role_name: nginx
  author: ROpdebee
  description: Installs nginx
  license: "GPL3"
  min_ansible_version: 2.4
  platforms:
    - name: Debian
      versions:
        - all
```

Structural Representation of Ansible Roles




```yaml
# roles/nginx/tasks/main.yml
---
- name: Ensure nginx is installed
  apt:
    name: nginx
    state: present
- name: Ensure nginx configuration is present
  file:
    path: "{{ item }}"
    dest: "{{ nginx_dir }}"
  loop: "{{ conf_files }}"
```

Change Distilling

Extracting Changes After the Fact

Fine-grained and Accurate Source Code Differencing

Jean-Rémy Falleri
Univ. Bordeaux,
LaBRI, UMR 5800
F-33400, Talence, France
falleri@labri.fr

Floréal Morandat
Univ. Bordeaux,
LaBRI, UMR 5800
F-33400, Talence, France
fmoranda@labri.fr

Xavier Blanc
Univ. Bordeaux,
LaBRI, UMR 5800
F-33400, Talence, France
xblanc@labri.fr

Matias Martinez
INRIA and University of Lille, France
matias.martinez@inria.fr

Martin Monperrus
INRIA and University of Lille, France
martin.monperrus@inria.fr

ABSTRACT

At the heart of software evolution is a sequence of edit actions, called an *edit script*, made to a source code file. Since software systems are stored version by version, the edit script has to be computed from these versions, which is known as a complex task. Existing approaches usually compute edit scripts at the text granularity with only *add line* and *delete line* actions. However, inferring syntactic changes from such an edit script is hard. Since moving code is a frequent action performed when editing code and it should also be taken into account. In this paper, we tackle these issues by introducing an algorithm computing edit scripts at the abstract syntax tree granularity including move actions. Our objective is to compute edit scripts that are short and close to the original developer intent. Our algorithm is implemented in a freely-available and extensible tool that has been intensively validated.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding Tools and Techniques

General Terms: Algorithms, Experimentation

Keywords: Software evolution, Program comprehension, Tree differencing, AST.

1. INTRODUCTION

The first law of software evolution states that almost all software systems have to evolve to be satisfactory [19]. Since this law was formulated, many studies have been performed to better understand how software systems evolve, and forms what is called the *software evolution* research field [21].

There is global software evolution (e.g. evolution of requirements, of execution environments, ...) and local software evolution (evolution of source code files). In this paper, we focus on the latter, that is on understanding how source code files evolve. In particular, we focus on *edit scripts*, that are sequences of edit actions made to a source code file. Usually, since software is stored in version control systems, edit scripts are computed between two versions of a same file. The goal of an edit script is to accurately reflect the actual change that has been performed on a file.

Edit scripts are used by developers on a daily basis. For example, the Unix `diff` tool takes as input two versions of a source code file and performs the Myers algorithm [24] at the text line granularity and returns an edit script indicating which lines have been added or deleted. However, the limitations of `diff` are twofold. First, it only computes additions and deletions and does not consider other kinds of edit actions such as update and move. Second, it works at a granularity (the text line) that is both coarse grain and not aligned with the source code structure: the abstract syntax tree.

To overcome this main limitation, there are algorithms that can work at the abstract syntax tree (AST) level [13]. The main advantage in using the AST granularity is that the edit script directly refers to the structure of the code. For instance, if an edit action is the addition of a new `function` node, it clearly means that a new function has been added in the code. Despite several key contributions (e.g. [13]), the problem of computing AST edit scripts is still open, with two main challenges: handling move actions, and scaling to fine-grained ASTs with thousands of nodes[1]. This is where this paper makes a contribution.

To design our novel algorithm, we take the viewpoint of the developer: she is never interested in the theoretical shortest edit script. She is rather interested in having an edit script that reflects well the actual changes that happened. Thus our objective is not to find the shortest sequence of actions between two versions, but a sequence that reflects well the developer intent. Consequently, we devise an algorithm based on heuristics that contain pragmatic rules on what a good edit script is, and as importantly, that is efficient and scales to large ASTs. This algorithm has been implemented within a freely-available and extensible tool[2].

To sum up, our contributions are:

- a novel efficient AST differencing algorithm that takes into account move actions, and its implementation;

[1]The best known algorithm with add, delete and update actions has a $O(n^3)$ time complexity with n being the number of nodes of the AST [27]. Computing the minimum edit script that can include move node actions is known to be NP-hard [4]

[2]`github.com/jrfaller/gumtree`

ASE'14, September 15-19, 2014, Vasteras, Sweden.

ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642982.

GumTree

Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction

Beat Fluri, *Student Member, IEEE*, Michael Würsch, *Student Member, IEEE*,
Martin Pinzger, *Member, IEEE*, and Harald C. Gall, *Member, IEEE*

Abstract—A key issue in software evolution analysis is the identification of particular changes that occur across several versions of a program. We present *change distilling*, a tree differencing algorithm for fine-grained source code change extraction. For that, we have improved the existing algorithm by Chawathe et al. for extracting changes in hierarchically structured data [8]. Our algorithm extracts changes by finding both a match between the nodes of the compared two abstract syntax trees and a minimum edit script that can transform one tree into the other given the computed matching. As a result, we can identify fine-grained change types between program versions according to our *taxonomy of source code changes*. We evaluated our change distilling algorithm with a benchmark that we developed, which consists of 1,064 manually classified changes in 219 revisions of eight methods from three different open source projects. We achieved significant improvements in extracting types of source code changes: Our algorithm approximates the minimum edit script 45 percent better than the original change extraction approach by Chawathe et al. We are able to find all occurring changes and almost reach the minimum conforming edit script, that is, we reach a mean absolute percentage error of 34 percent, compared to the 79 percent reached by the original algorithm. The paper describes both our change distilling algorithm and the results of our evaluation.

Index Terms—Source code change extraction, tree-differencing algorithms, software repositories, software evolution analysis.

1 INTRODUCTION

SINCE Lehman's Laws of Program Evolution from the 1980s [25], it has been well understood that software has to be adapted to changing requirements and environments or it becomes progressively less useful. Change is broadly accepted as a crucial part of a software's life cycle. As a consequence, in recent years, several techniques and tools have been developed to aid software engineers in maintaining and evolving large complex software systems. For instance, Ying et al. or Zimmermann et al. developed approaches that guide programmers along related changes by telling them "programmers who changed these functions also changed..." [45], [47]. The Hipikat tool of Čubranić et al. used project history information to provide recommendations for a modification task [9]. Gall et al. detected possible maintainability hot spots by analyzing cochange relationships of modules [13].

We argue that such techniques and tools are valuable but suffer from the low quality of information available for changes. Typically, such information, in particular for source code, is stored by versioning systems (for example, CVS or Subversion). They keep track of changes by storing the text lines *added* and/or *deleted* from a particular file. Structural changes in the source code are not considered at all.

More sophisticated approaches are able to narrow down changes to the method level, but fail in further qualifying changes such as the addition of a method invocation in the else branch of an if-statement. Furthermore, a classification of changes according to their impact on other source code entities is missing. In particular, the latter information is important to improving the quality of software evolution results and, as a consequence, to providing better support for programmers and system analysts.

Since source code can be represented as abstract syntax trees (ASTs), tree differencing can be used to extract detailed change information. This approach is promising because exact information on each entity and statement is available in an AST. In our previous work [12], we built a *taxonomy of source code changes* that defines source code changes according to tree edit operations in the AST and classifies each change type with a *significance level*. The level expresses how strongly a change may impact other source code entities and whether a change may be functionality modifying or functionality preserving. In our taxonomy, we focus on object-oriented programming languages (OOPLs) and Java in particular. By adjusting the change type extraction, the taxonomy can also be used for other OOPLs. In total, our taxonomy defines 35 change types.

In this paper, we present *change distilling*, a tree-differencing algorithm for fine-grained source code change extraction. For that, we improved the existing algorithm for extracting changes in hierarchically structured data by Chawathe et al. [8]. This algorithm finds changes according to basic tree edit operations such as *insert*, *delete*, *move*, or *update* of tree nodes.

• *The authors are with the Department of Informatics, University of Zurich, Binzmühlestrasse 14, CH-8050 Zürich, Switzerland.*
E-mail: {fluri, wuersch, pinzger, gall}@ifi.uzh.ch.

Manuscript received 15 Jan. 2007; revised 13 July 2007; accepted 23 July 2007; published online 3 Aug. 2007.
Recommended for acceptance by H. Muller.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0012-0107.
Digital Object Identifier no. 10.1109/TSE.2007.70731.

0098-5589/07$25.00 © 2007 IEEE
Published by the IEEE Computer Society

ChangeDistiller

Empirical Software Engineering (2019) 24: 491–535
https://doi.org/10.1007/s10664-018-9644-3

CrossMark

Querying distilled code changes to extract executable transformations

Reinout Stevens[1] · Tim Molderez[2] · Coen De Roover[2]

Published online: 30 August 2018

Abstract
Change distilling algorithms compute a sequence of fine-grained changes that, when executed in order, transform a given source AST into a given target AST. The resulting change sequences are used in the field of mining software repositories to study source code evolution. Unfortunately, detecting and specifying source code evolutions in such a change sequence is cumbersome. We therefore introduce a tool-supported approach that identifies minimal executable subsequences in a sequence of distilled changes that implement a particular evolution pattern, specified in terms of intermediate states of the AST that undergoes each change. This enables users to describe the effect of multiple changes, irrespective of their execution order, while ensuring that different change sequences that implement the same code evolution are recalled. Correspondingly, our evaluation is two-fold. We show that our approach is able to recall different implementation variants of the same source code evolution in histories of different software projects. We also evaluate the expressiveness and ease-of-use of our approach in a user study.

Keywords Change distilling · Change querying · Logic meta-programming

1 Introduction

The use of a Version Control System (VCS) has become an industry best practice for developing software. Researchers in the field of mining software repositories (MSR) leverage the resulting revision histories to study the evolution of software systems. However, most VCSs

Communicated by: Gabriele Bavota and Andrian Marcus

✉ Tim Molderez
tim.molderez@vub.be

Reinout Stevens
reinout@reinoutstevens.be

Coen De Roover
coen.de.roover@vub.be

[1] Maxflow BVBA, Leuven, Belgium

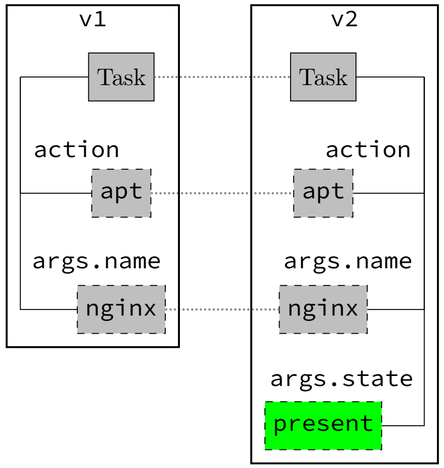
[2] Software Languages Lab, Vrije Universiteit Brussel, Ixelles, Belgium

Springer

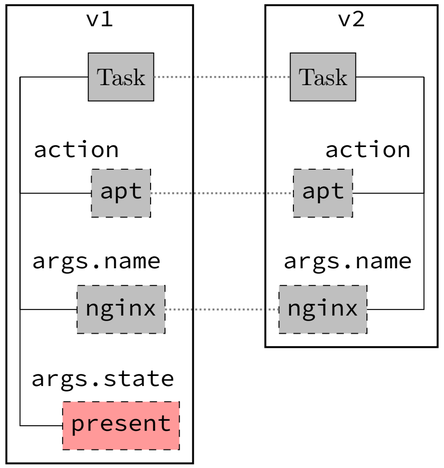
ChangeNodes

Change Distilling of Ansible Roles

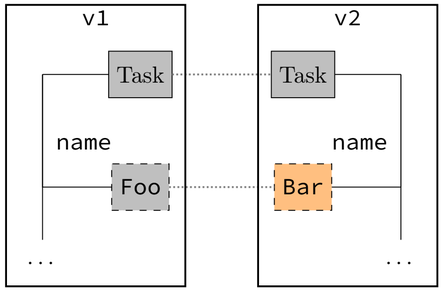
Additions



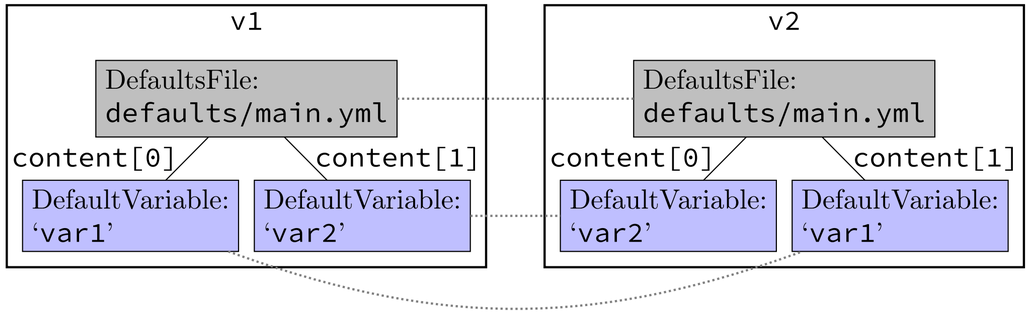
Removals



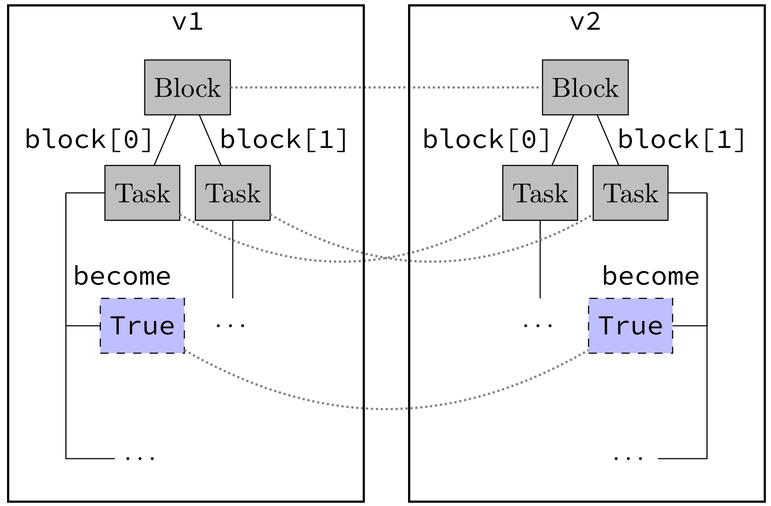
Updates



Local relocation

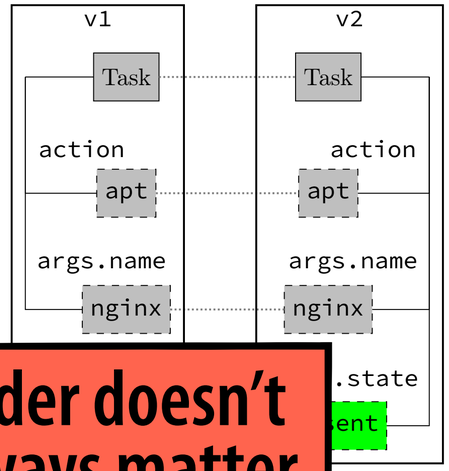


Global relocation



Change Distilling of Ansible Roles

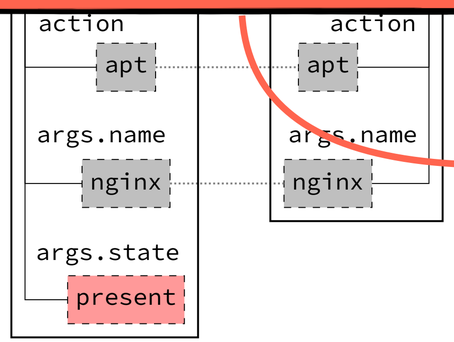
Additions



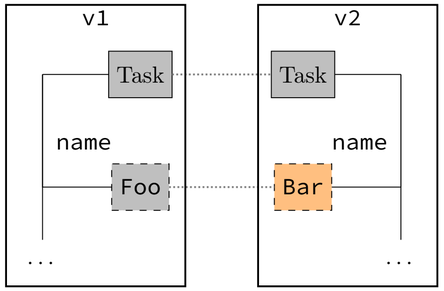
Order doesn't always matter

Removals

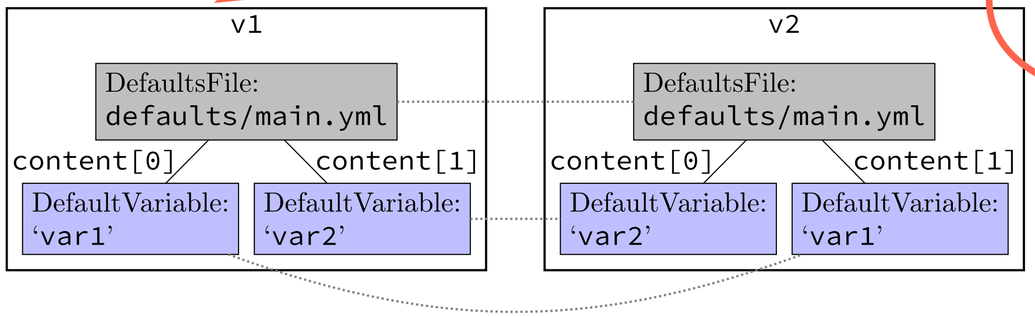
Name has no semantic relevance
Update shouldn't matter



Updates

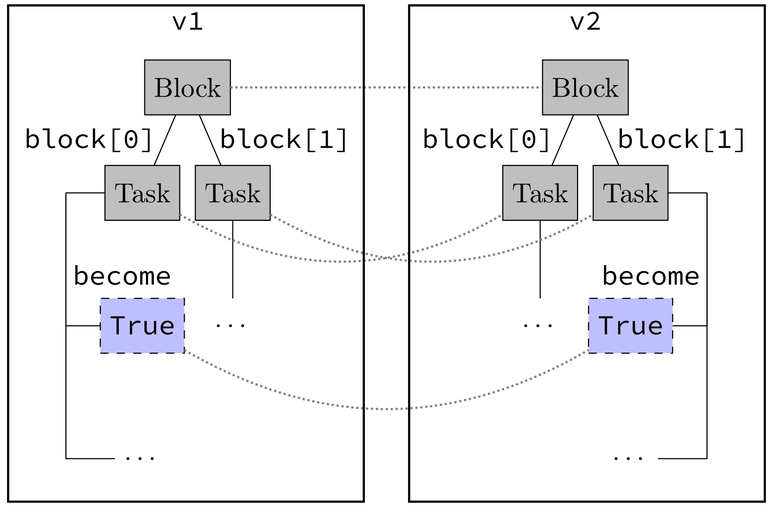


Local relocation

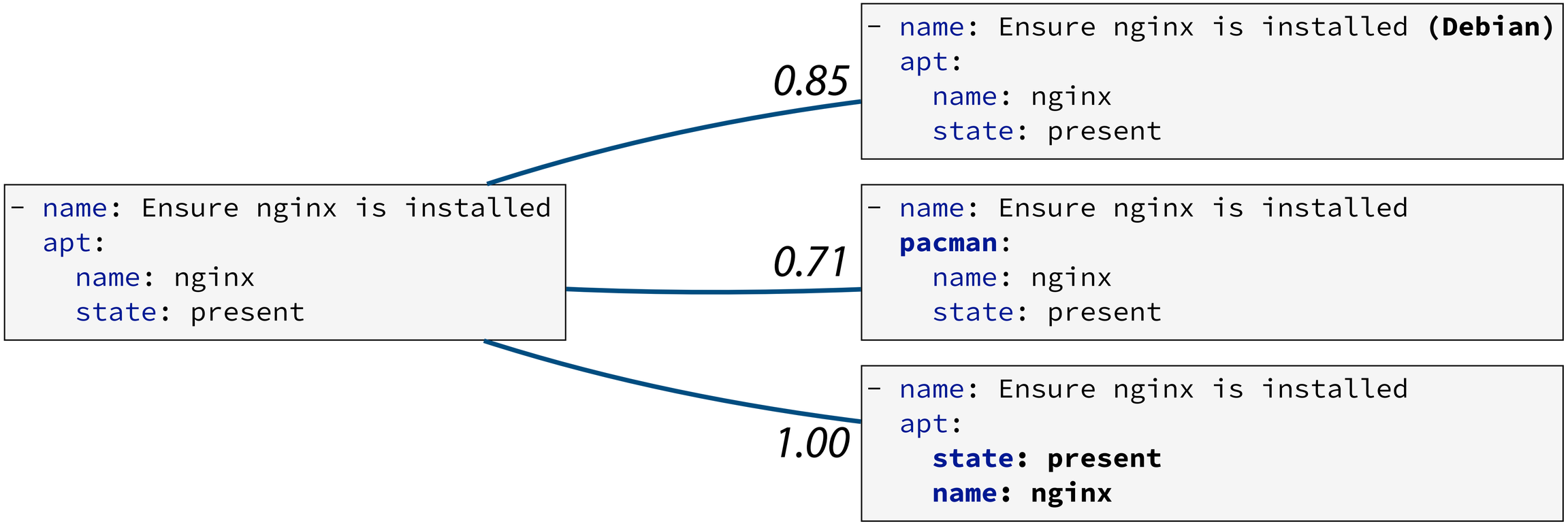


Too fine-grained

Global relocation



Task Similarity

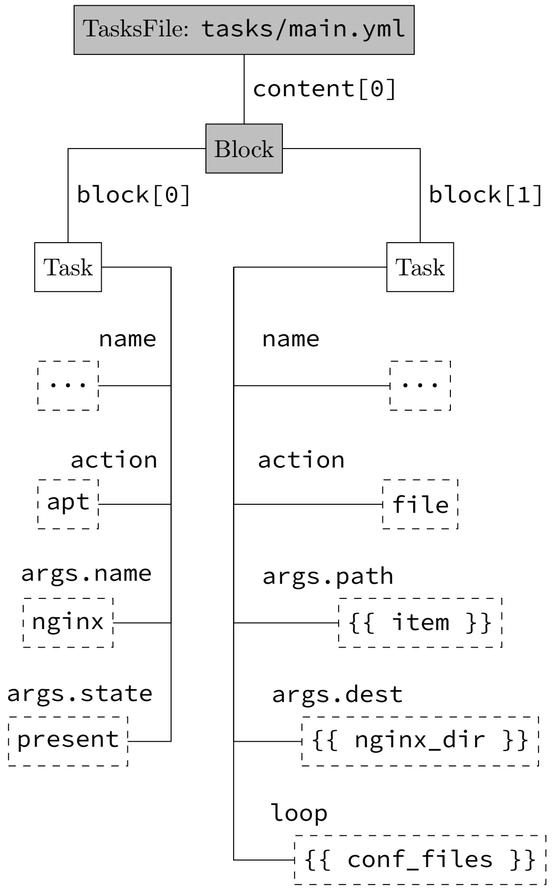


$$sim_{task}(T_1, T_2) = \frac{\sum_{kw \in T_1 \cap T_2} w(kw)}{\sum_{kw \in T_1 \cup T_2} w(kw)}$$

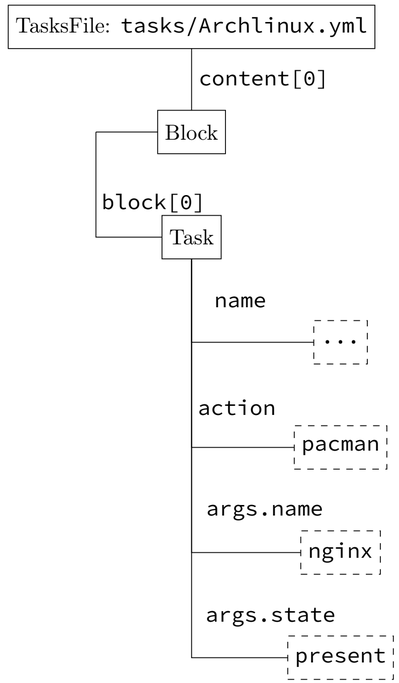
$$w(kw) = \begin{cases} 1, & \text{if } kw \in \{\texttt{action}, \texttt{args}, \texttt{when}, \texttt{loop}, \texttt{loop\_control}\} \\ 0.5, & \text{otherwise} \end{cases}$$

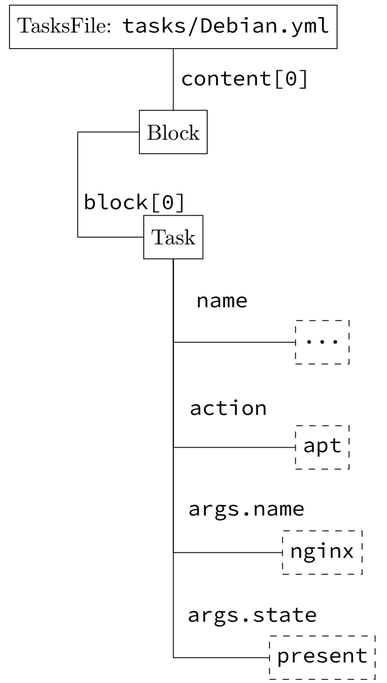
“Weighted Jaccard”

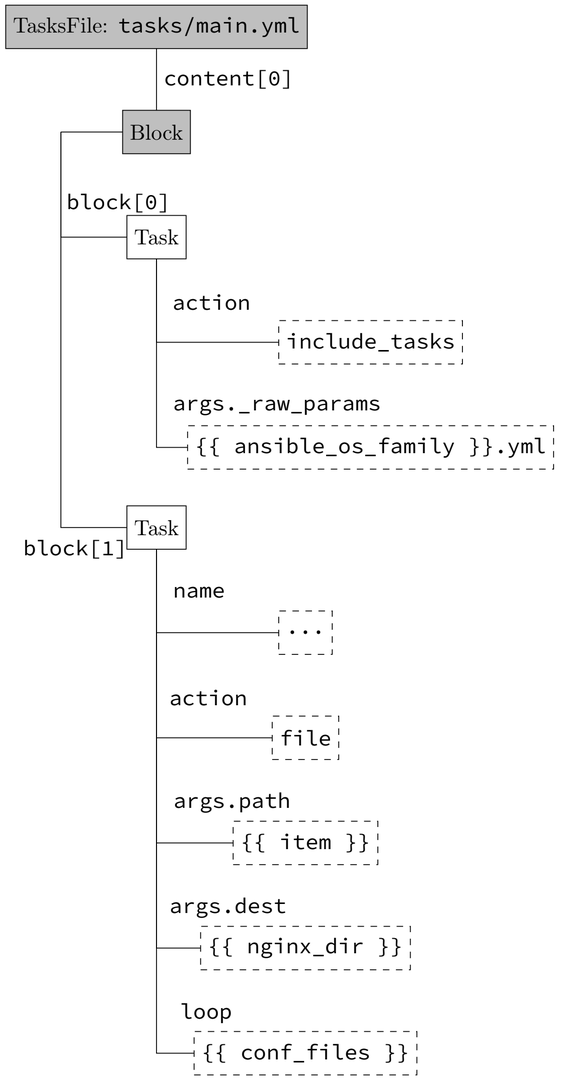
Task Matching



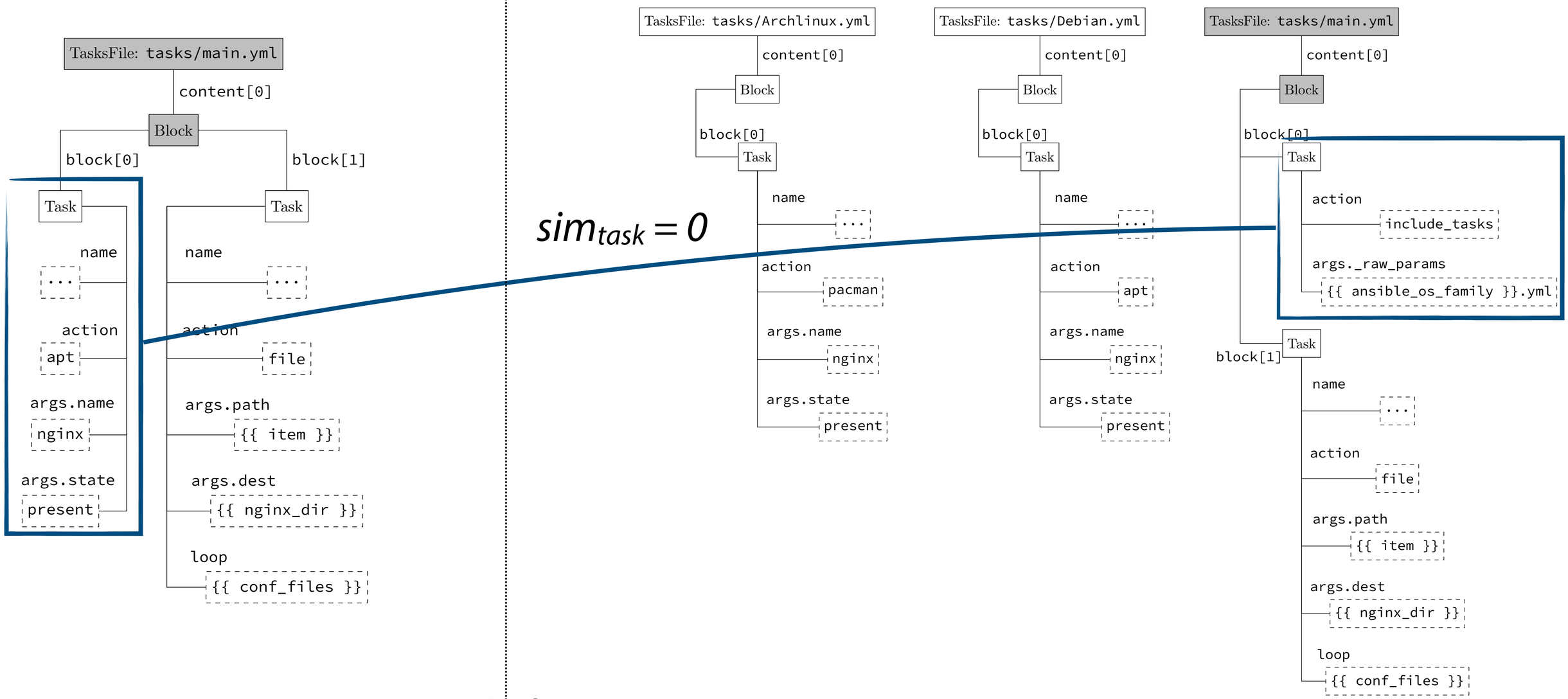
v1 v2





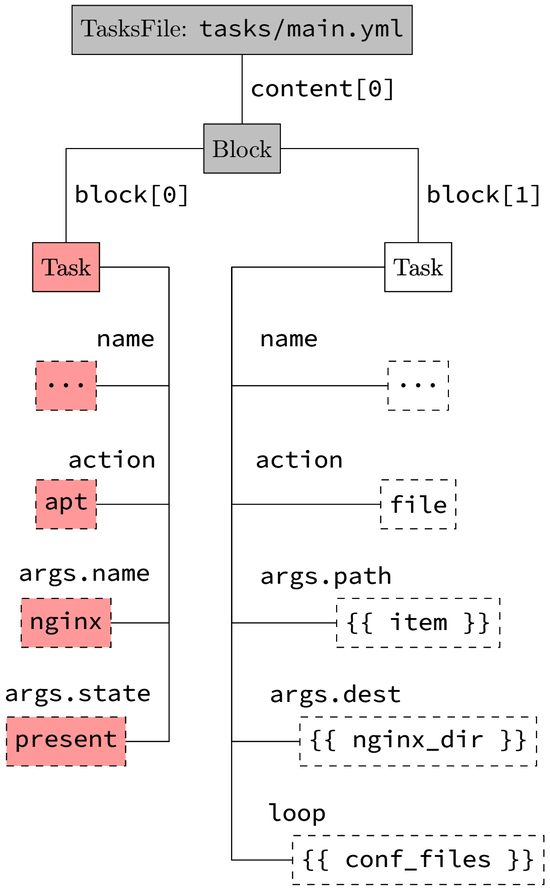


Task Matching

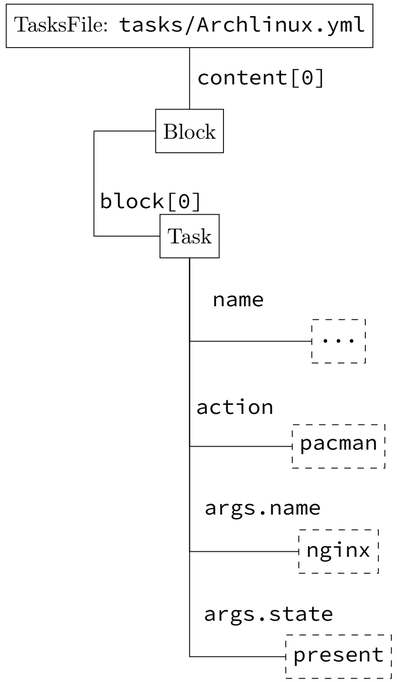


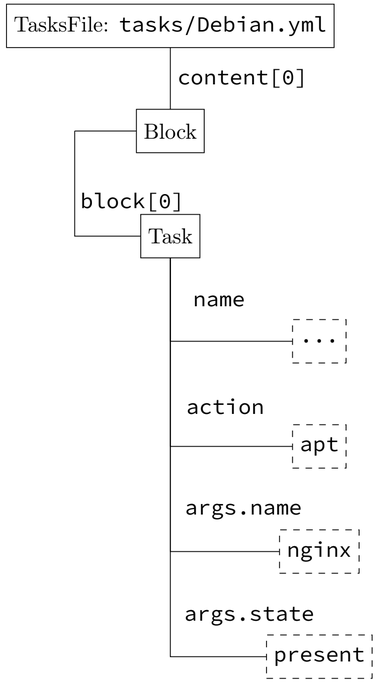
v1 v2

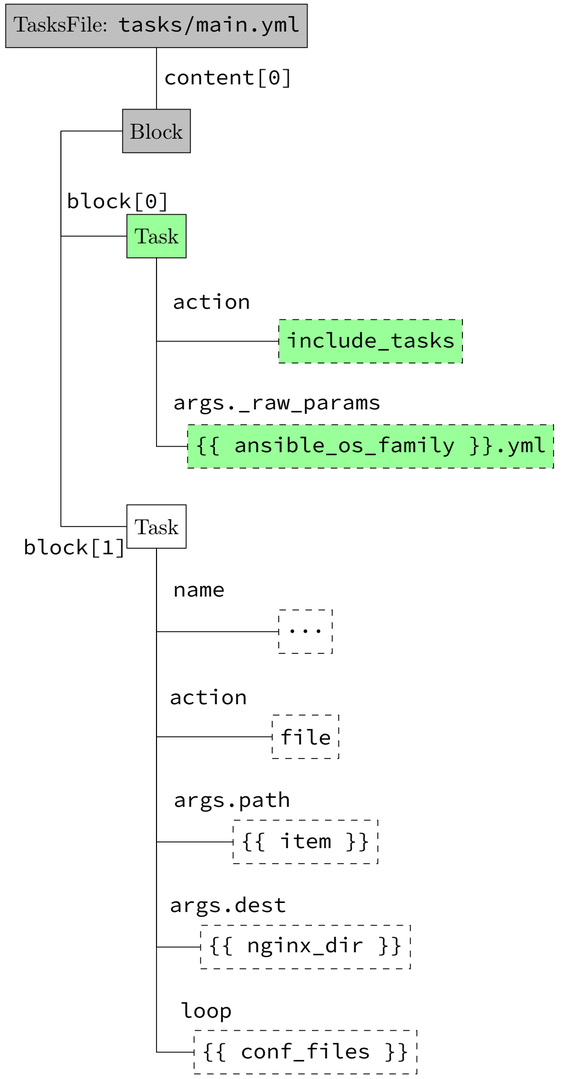
Task Matching



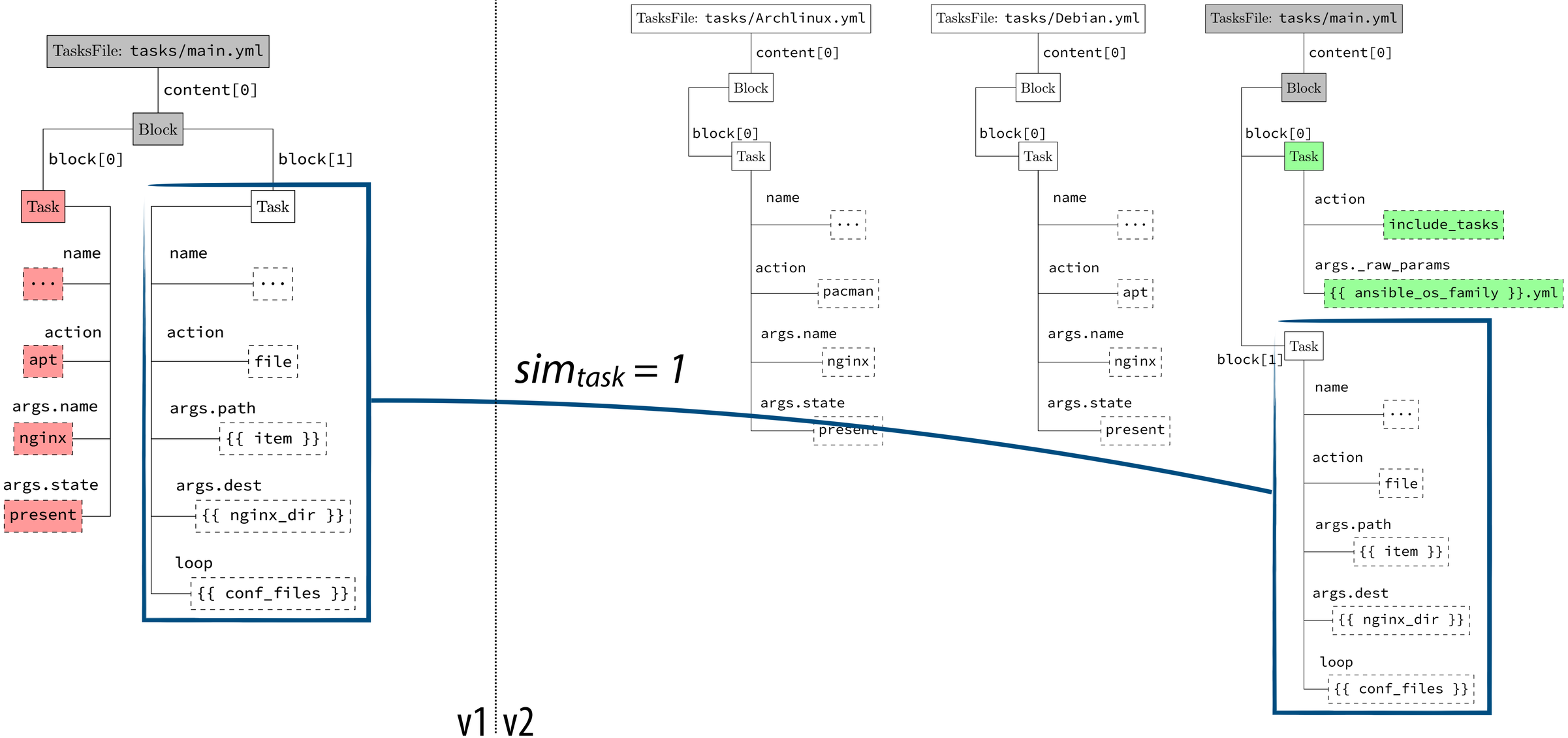
v1 v2



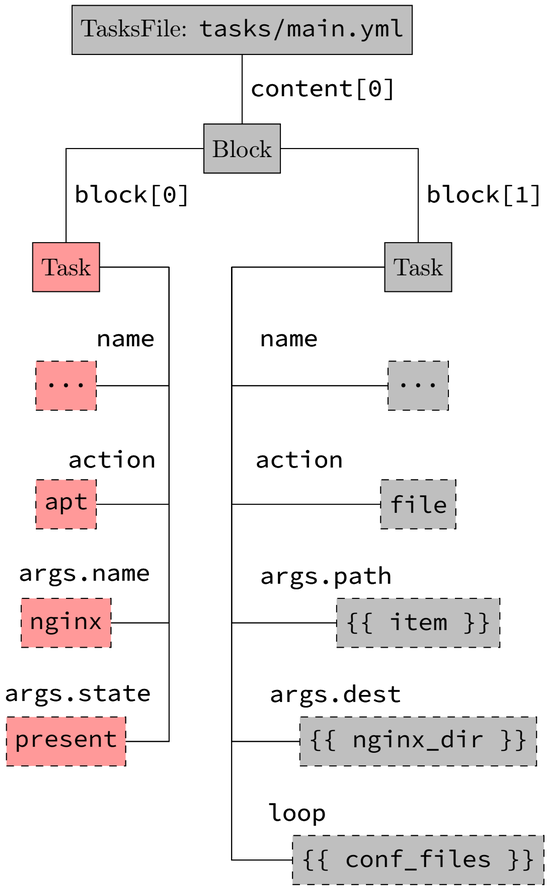




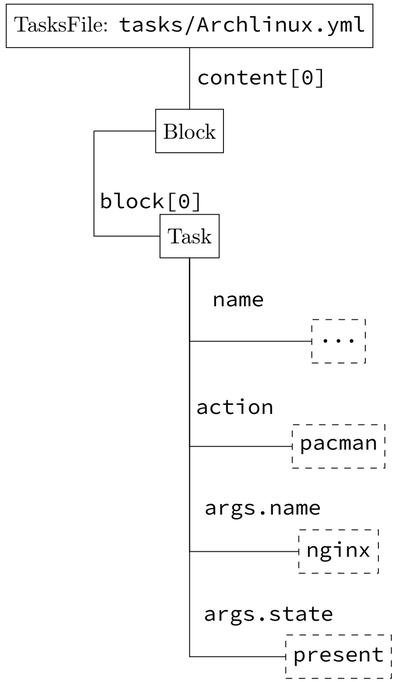
Task Matching

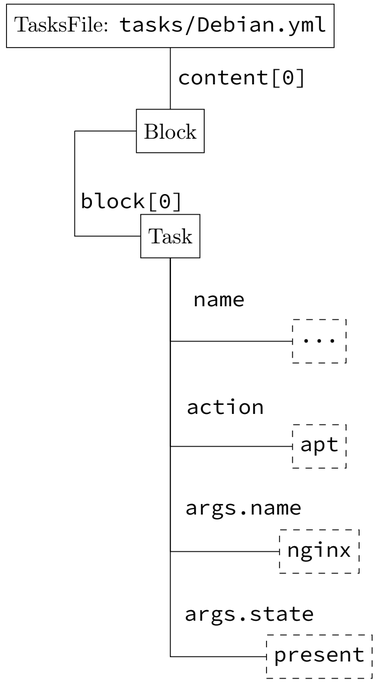


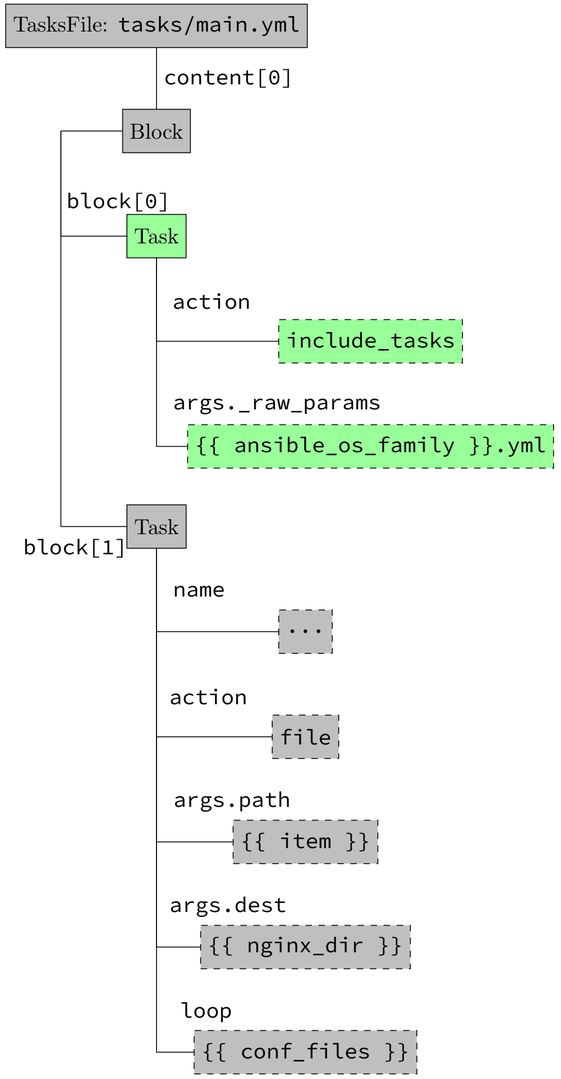
Task Matching



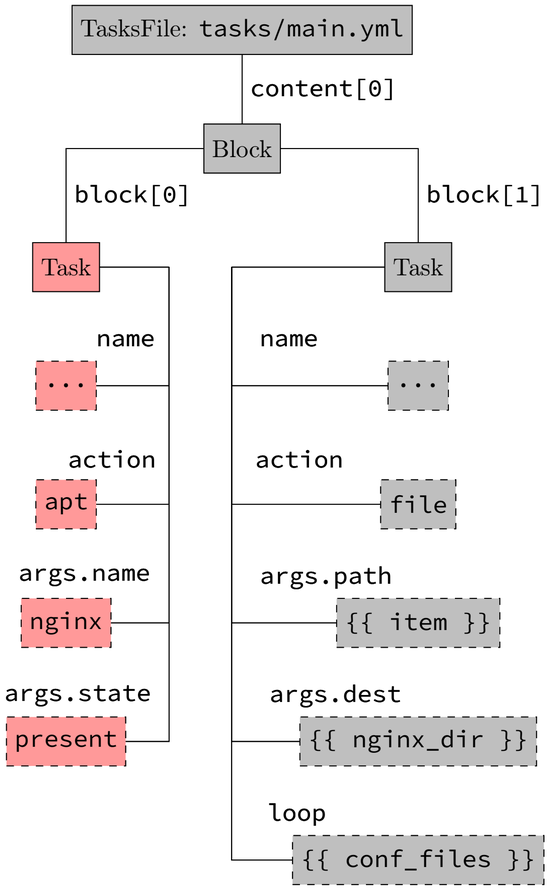
v1 v2



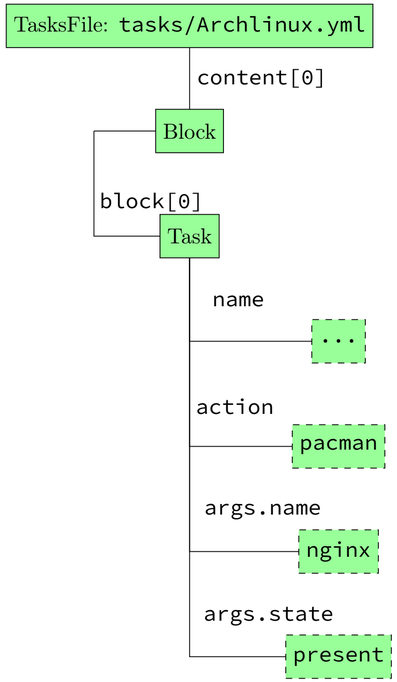


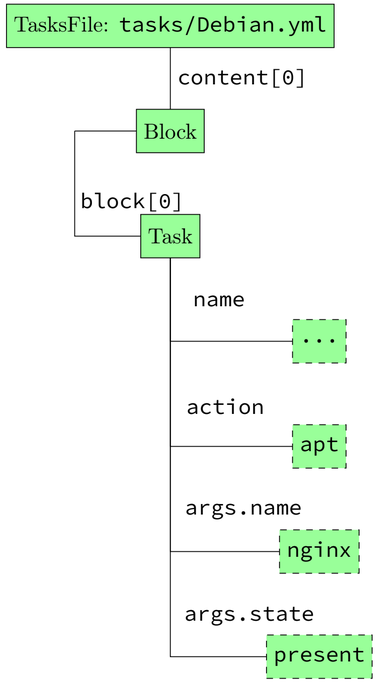


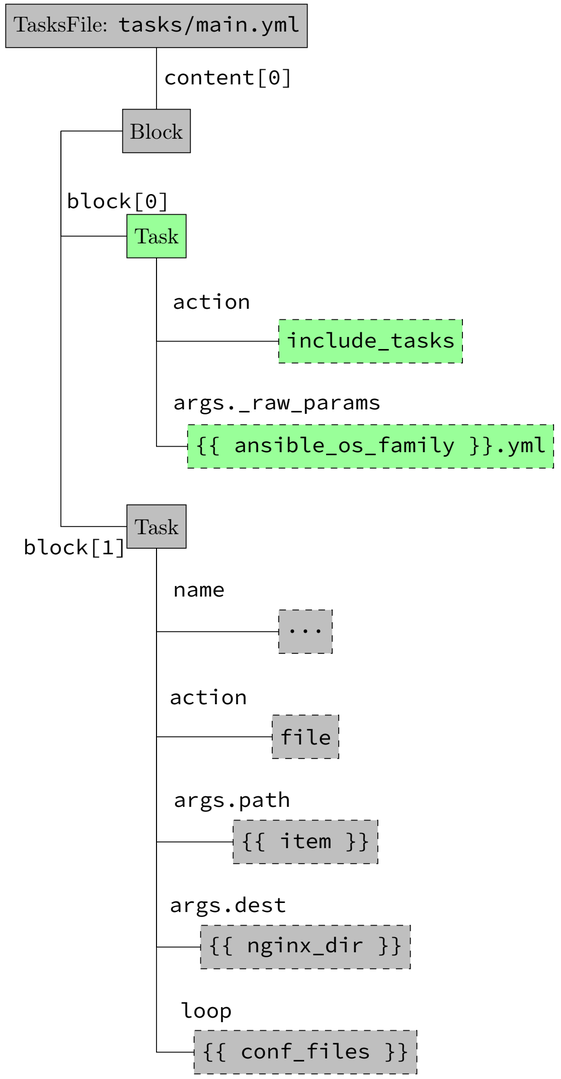
Task Matching



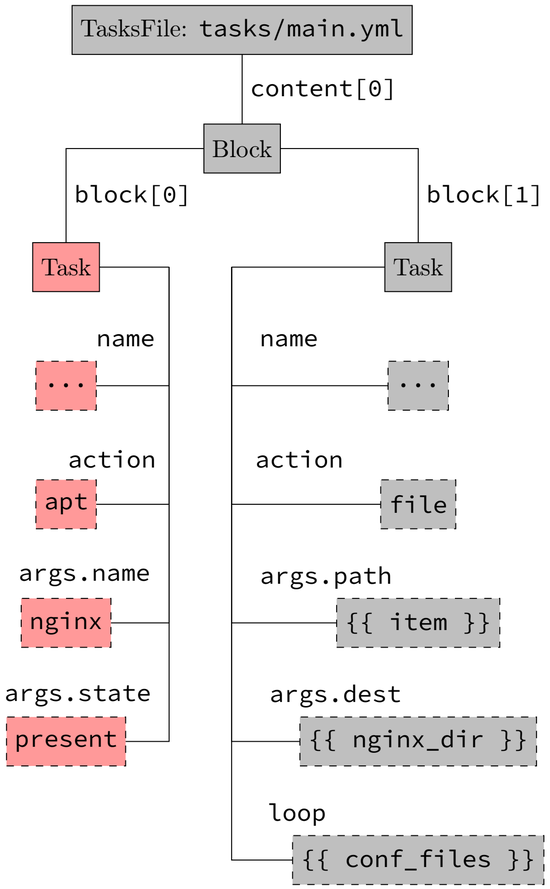
v1 v2



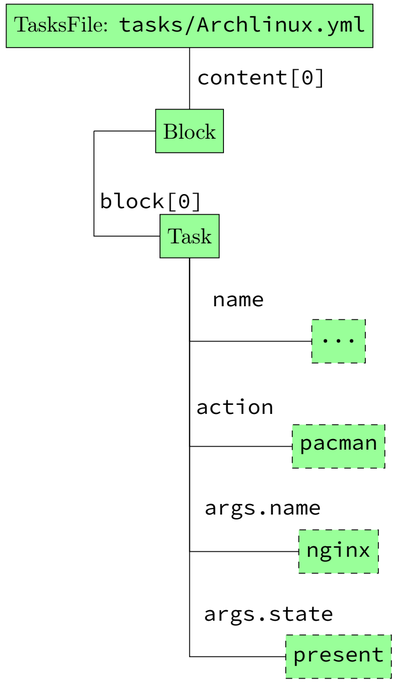


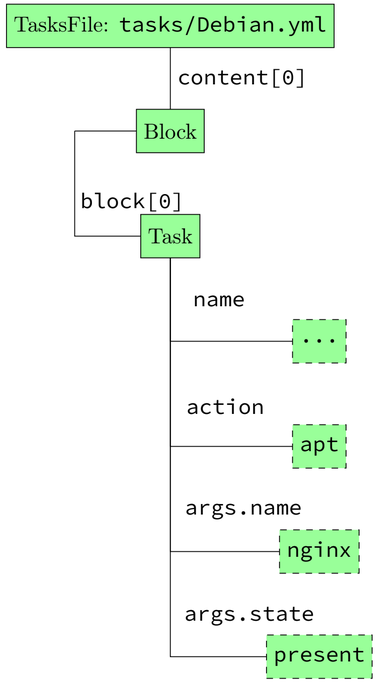


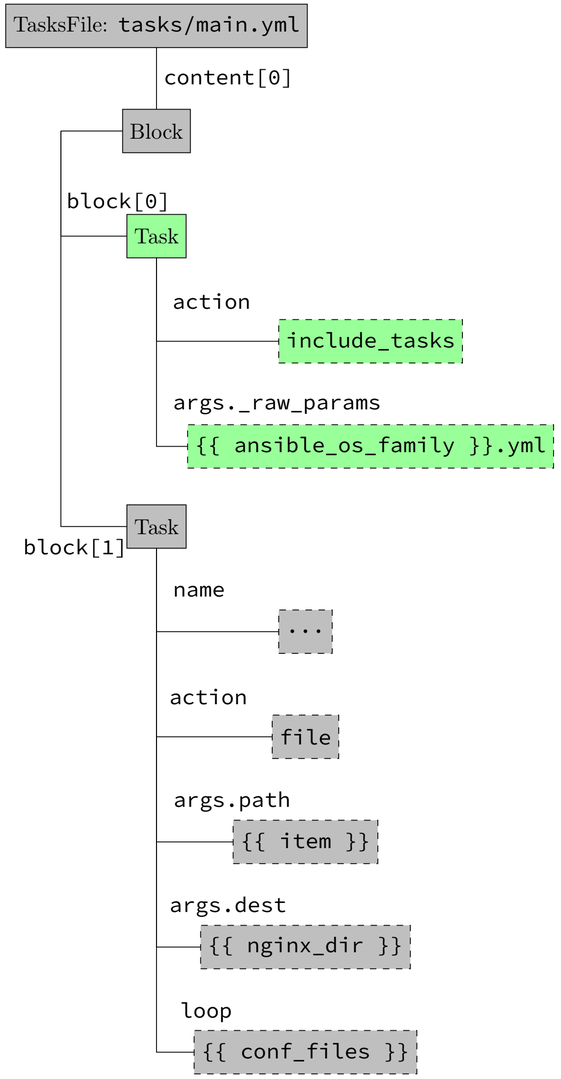
Task Relocation Matching



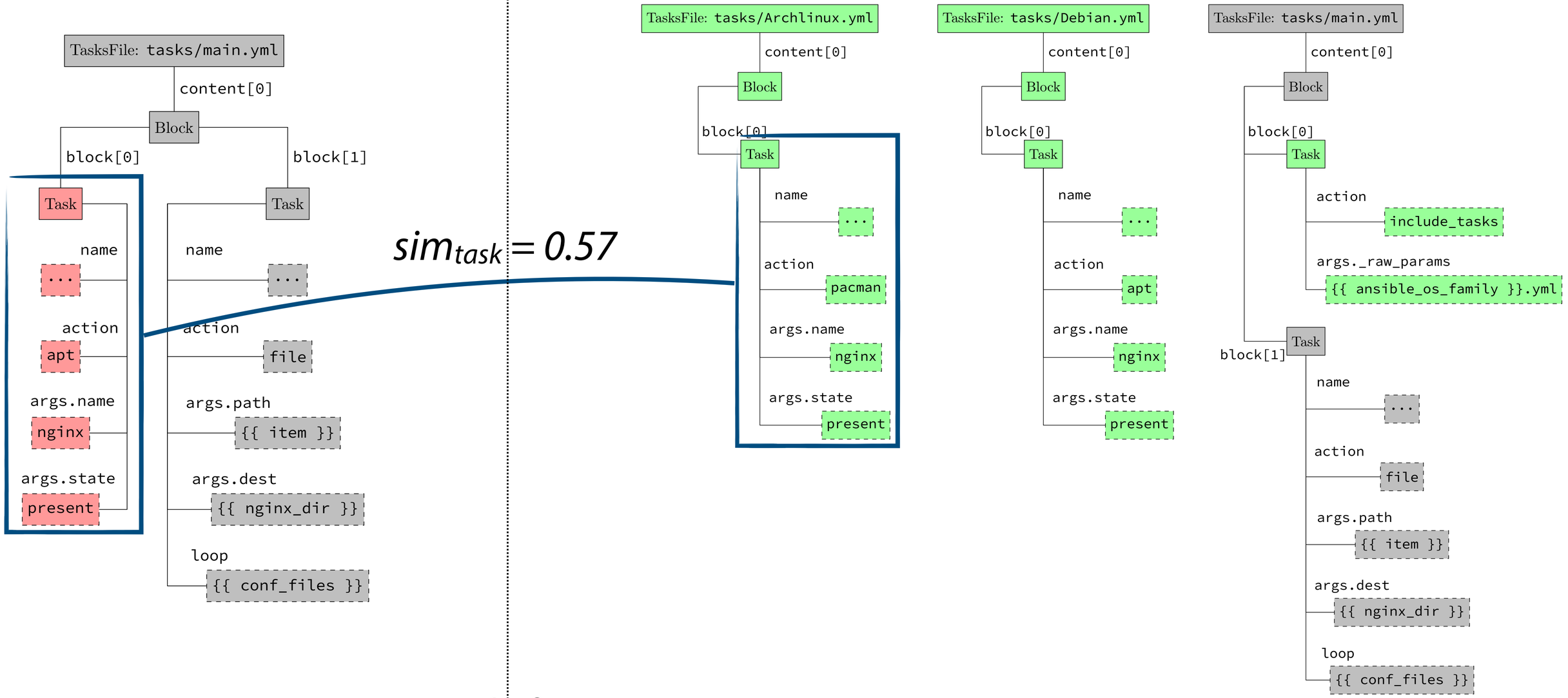
v1 v2



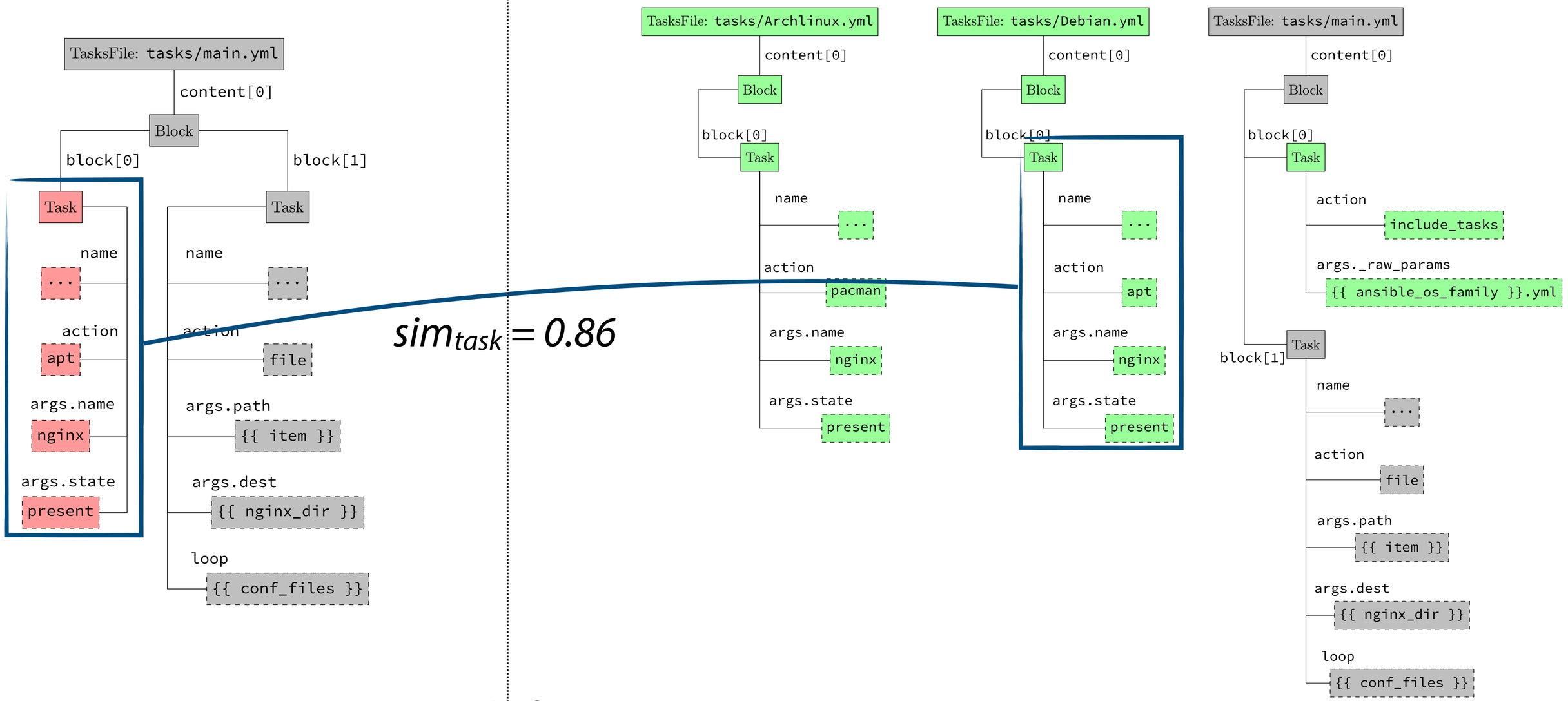




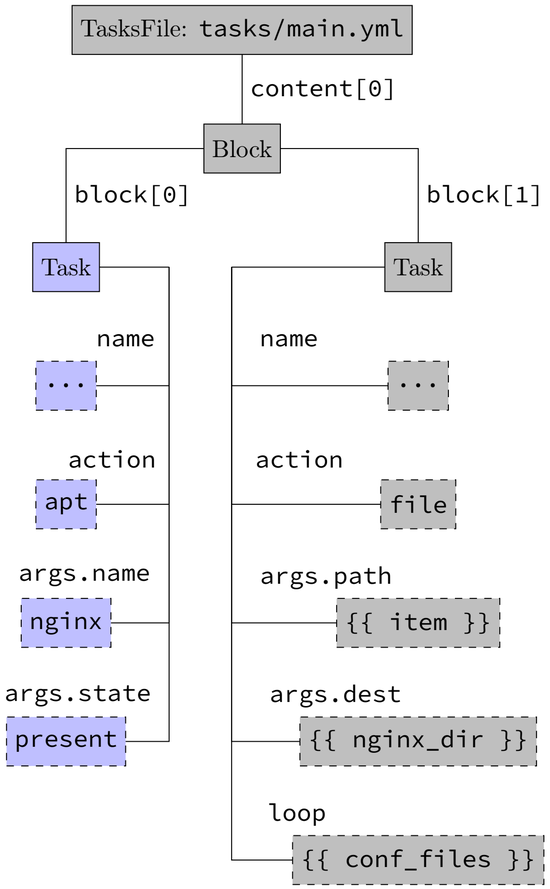
Task Relocation Matching



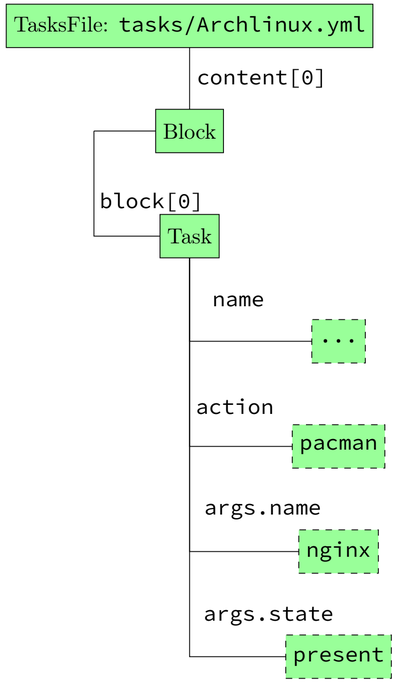
Task Relocation Matching

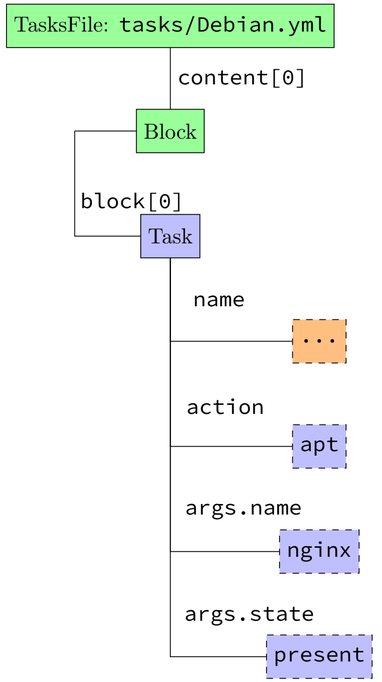


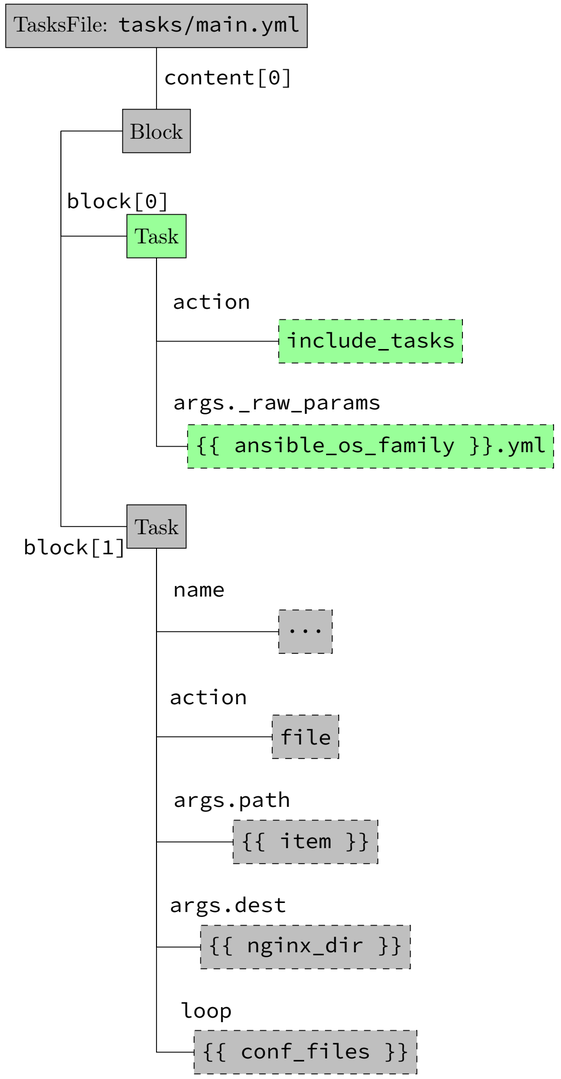
Task Relocation Matching



v1 v2

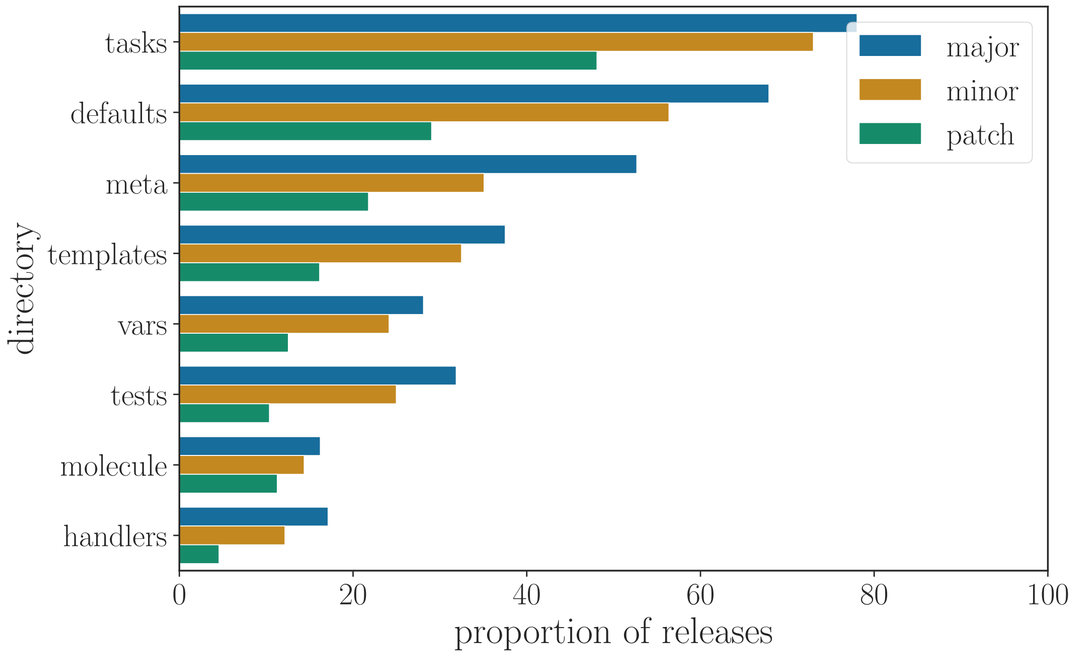




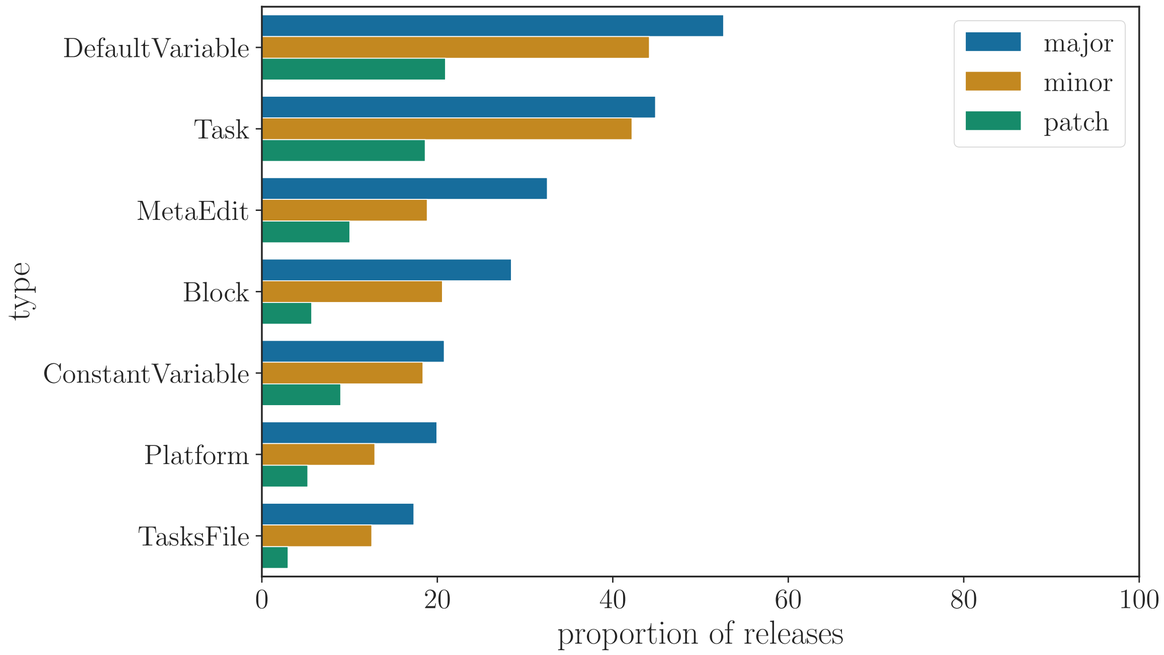


Structural Changes in Role Releases

Syntactical changes



Structural changes

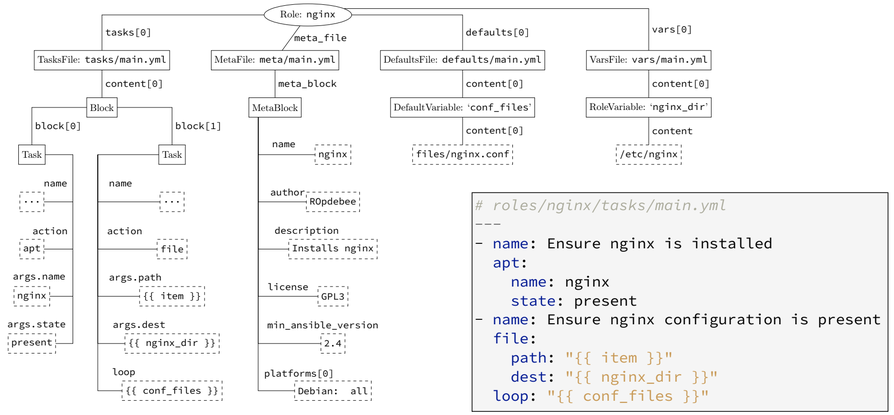


~30% of releases contain no structural change

R. Opdebeeck, A. Zeraouli, C. Velázquez-Rodríguez, C. De Roover. "Does Infrastructure as Code Adhere to Semantic Versioning? An Analysis of Ansible Role Evolution", In Proc. 20th International Working Conference on Source Code Analysis and Manipulation, 2020.

Conclusion

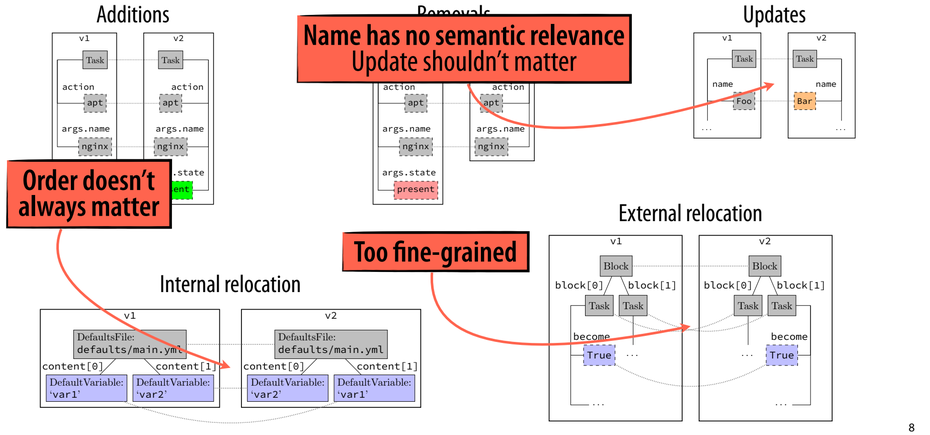
Structural Representation of Ansible Roles




```yaml
# roles/nginx/tasks/main.yml
---
- name: Ensure nginx is installed
  apt:
    name: nginx
    state: present
- name: Ensure nginx configuration is present
  file:
    path: "{{ item }}"
    dest: "{{ nginx_dir }}"
  loop: "{{ conf_files }}"
```

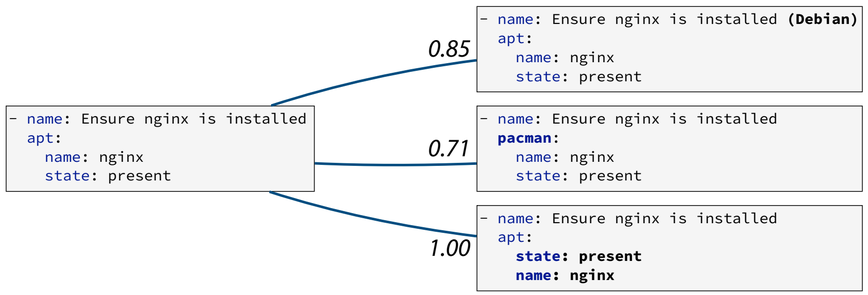
5

Change Distilling of Ansible Roles



8

Task Similarity

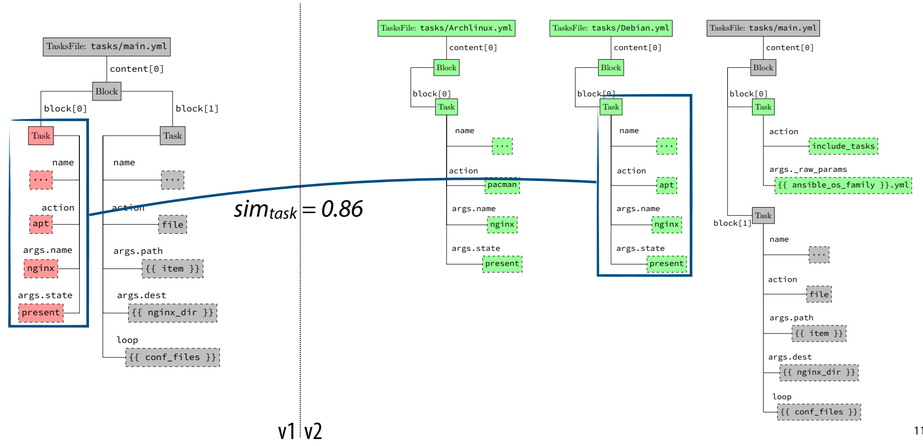


$$sim_{task}(T_1, T_2) = \frac{\sum_{kw \in T_1 \cap T_2} w(kw)}{\sum_{kw \in T_1 \cup T_2} w(kw)} \qquad w(kw) = \begin{cases} 1, & \text{if } kw \in \{\texttt{action}, \texttt{args}, \texttt{when}, \texttt{loop}, \texttt{loop\_control}\} \\ 0.5, & \text{otherwise} \end{cases}$$

"Weighted Jaccard"

9

Task Relocation Matching



11