Managing Conflicts in Cross-DevOps Declarative Reconfigurations

Jolan Philippe
Univ. Orléans, INSA CVL, LIFO EA 4022
Orléans, France
jolan.philippe1@univ-orleans.fr

Hélène Coullon
IMT Atlantique, Nantes Université, Ecole Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004
Nantes, France
helene.coullon@imt-atlantique.fr

Charles Prud'homme
IMT Atlantique, Nantes Université, Ecole Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004
Nantes, France
charles.prudhomme@imt-atlantique.fr

Abstract

When facing scale and geo-distribution challenges, companies may organize the deployment of their distributed software applications and associated infrastructures in cross-functional or cross-geographical DevOps teams, in short cross-DevOps. This leads to decentralize reconfiguration where each team is responsible for the management of a subpart of the overall system. However, in this model, two DevOps teams may concurrently submit two conflicting reconfigurations, which can produce errors or infinite loops. We present Ballet$^+$, an extended version of Ballet (decentralized declarative reconfiguration tool), that is able, when computing the plan of changes (before execution), to detect conflicts, explain the source of the conflicts, and trace back required information to DevOps teams to help solve the problem. To this purpose, Ballet$^+$ relies on constraint programming and distributed algorithms. In particular, Ballet$^+$ leverages the *QuickXplain* algorithm to identify conflicting constraints in local models. *QuickXplain* is coupled with a mechanism to backtrack messages exchanged by the different instances of Ballet$^+$ to report the source of the conflicts to each team. Ballet$^+$ is evaluated on synthetic cases and a real-world multi-site *OpenStack* scenario.

Keywords

cross-DevOps, Decentralized reconfiguration, Infrastructure-as-Code, Conflict, Concurrency, Constraint Programming

ACM Reference Format:
Jolan Philippe, Hélène Coullon, and Charles Prud'homme. 2025. Managing Conflicts in Cross-DevOps Declarative Reconfigurations. In *Proceedings of UCC'25*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/xxxxxx.xxxxxx

UCC'25, Nantes, France

ACM ISBN 978-1-4503-xxxx-x/25/06
https://doi.org/10.1145/xxxxxx.xxxxxx

1 Introduction

Managing the deployment of cloud-based software applications (*e.g.*, cloud-native micro-services applications) and of cloud infrastructures they rely on (*e.g.*, public or private cloud resources such as virtual machines, containers, managed load balancers, and serverless platforms) has been traditionally a complex task. This management has progressively evolved from a slow, semi-manual process, requiring difficult collaborations between developers and system administrators, towards a fully automated and fast process handled by DevOps engineers.

Among mechanisms that favor this automation is the *declarative* approach, where the desired state of *resources* (either infrastructure or applications) is specified, not how to reach it. Then, the declarative engine is in charge of synthesizing a reconfiguration *plan* with the necessary changes, and to *execute* it. This is, for example, a mechanism widely adopted in Infrastructure as Code (IaC) [1], a domain that consists of specifying infrastructure configurations and application deployments as *programs* that become shareable, versioned, and testable, as traditional software. This is also a mechanism that has been studied, under the name of *reconfiguration*, in the component-based software engineering (CBSE) community, an older and broader field, but one that includes contributions close to those of IaC [2]. In this paper, we use interchangeably deployment or reconfiguration to refer either to the initial deployment process or to dynamic changes on an existing deployment.

With the growing complexity and scale of distributed software systems handled by companies, DevOps teams have evolved into collaborating cross-functional or cross-geographical teams, namely *cross-DevOps*. In such an organization, each DevOps team manages a subpart of the resources, but may depends on other teams' resources either by providing or using them. In the purpose of having loosely coupled teams, to avoid the burden (or impossibility) of managing a global central state of a very large deployment, and to favor fault-tolerance, cross-DevOps organizations lead to *decentralized declarative reconfigurations* [3, 4].

A decentralized declarative engine both computes and executes the plan in a decentralized manner without global knowledge. However, with these decentralized engines, two or more DevOps teams can concurrently trigger reconfigurations that are incompatible in practice because of dependencies

between their respective resources. This can lead to errors, or worse, infinite loops within the engine. One solution could be to have a global locking mechanism to avoid concurrent reconfigurations, but this would also delay non-conflicting reconfigurations and reintroduce tight coupling between DevOps teams. In this paper, we instead introduce a new approach to detect conflicts between concurrent reconfigurations during the plan synthesis, find the conflict explanation, and pass on the relevant information to the teams. If no conflicts are detected, then plans can simply be executed without risk of error or deadlock.

This paper introduces BALLET$^+$, an extension of BALLET [4], a fully decentralized reconfiguration framework. While BALLET generates complete reconfiguration plans before execution by combining constraint programming with distributed algorithms, its purely decentralized design prevents it from detecting certain conflicts that may arise from independent DevOps actions. As a result, such conflicts can be detected only during execution, forcing costly rollbacks and risking inconsistency issues also observed in partially on-the-fly approaches like Mjuz [3]. BALLET$^+$ addresses this limitation by enhancing the constraint-programming layer to **identify conflicts prior to execution** and to **produce explicit explanations** that help DevOps teams resolve the conflicting goals in advance.

The contributions of this paper are :

- to leverage, in our declarative cross-DevOps context, mechanisms of unsatisfiability explanation from the constraint programming field;
- to combine local explanations with a distributed algorithm, ensuring that conflict explanations are propagated back to the relevant DevOps teams;
- BALLET$^+$, an extension of BALLET, and its experimental evaluation on both synthetic models representing various distributed topologies of resources, and a realistic case study involving a reconfiguration of a multi-site OpenStack.

The rest of this paper is organized as follows. Section 2 presents our motivating case study that will be used throughout the paper. Section 3 presents background on BALLET's reconfiguration engine and its usage for cross-DevOps teams. In Section 4, we present BALLET$^+$, an approach to detect and manage conflicts, and how it has been implemented. Section 5 evaluates BALLET$^+$ on both synthetic cases and a realistic use-case. Finally, Section 6 studies the related work, and Section 7 concludes this work and opens to some perspectives.

2 Motivating case study

As a motivating example, we use a decentralized version of OPENSTACK on multiple sites. OPENSTACK is the open-source solution to address the IaaS level of the cloud paradigm. When handling large-scale geographically distributed cloud infrastructures (*e.g.*, Edge computing), it is required to handle multiple OPENSTACK sites in a collaborative manner to support the load and to be relatively resilient to network disconnections [5]. One possible way[1] is to decentralize the MariaDB database, central to the Keystone authentication service, using a Galera cluster for replication across sites. Galera Cluster is an overlay for MySQL DBMS engines that enables replication between databases [6]. This use-case was showcased during the 2018 Vancouver OPENSTACK summit[2],[3].

Figure 1 gives an overview of the assembly of services corresponding to this use case with the different sites and nodes, and using the usual CBSE graphical notation for use and provide ports [7, 8]. Multiple DevOps teams (typically one per site, and one for the master database) would be deployed to manage and operate such multi-sites OPENSTACK. In a scenario where one needs to update (in place, not a rolling upgrade) the master database, many changes have to be triggered in other services because there is a chain of dependencies. In fact, with n as the number of sites, a total of $1 + 3 \times n$ reconfiguration programs are required to handle this update in a decentralized way, and a total of $3 + 20 \times n$ changes are required in $1 + 5 \times n$ services. Cross-DevOps reconfiguration solutions would typically handle this complexity automatically in a declarative and decentralized manner [3, 4].

In this paper, we extend this scenario as follows. We assume that three versions of each service are available and that only services with the same version number are compatible. This is a simplification of OpenStack releases, where different versions of each service are packaged together and compatible[4]. We consider the scenario where a DevOps engineer submits a declarative reconfiguration to upgrade the common libraries (*common* component) of the master node to version 2 (v2). Consequently, all dependent services must be recursively upgraded to v2. Second, concurrently (*i.e.*, at the same time or before the end of the first reconfiguration planning phase), another DevOps responsible for *worker site 1* submits a reconfiguration to update *nova* to version 3 (v3). This introduces a conflict: the first goal enforces *nova* to be upgraded to v2 as a consequence of the recursive dependency, while the second goal explicitly requires *nova* to be upgraded to v3. Since *nova* cannot simultaneously exist in both v2 and v3, the reconfiguration is infeasible.

3 Background on Cross-DevOps Reconfiguration

We base our study on a declarative cross-DevOps reconfiguration tool called BALLET [4]. Each node or site (or any way to divide the system) in the considered cross-DevOps system (*i.e.*, managed by a different DevOps team) is managed by one instance of BALLET. Then, BALLET's instances collaborate to calculate and execute the reconfiguration programs needed on each node (or site) to reach the desired state. In this

[1] see https://www.starlingx.io/ for another approach
[2] OPENSTACK summit video (https://youtu.be/AUzaJ8rBvEg)
[3] https://beyondtheclouds.github.io/blog/openstack/cockroachdb/2018/06/04/evaluati
of-openstack-multi-region-keystone-deployments.htmlgalera-in-multi-master-replication-mode-and-keystone
[4] https://releases.openstack.org/

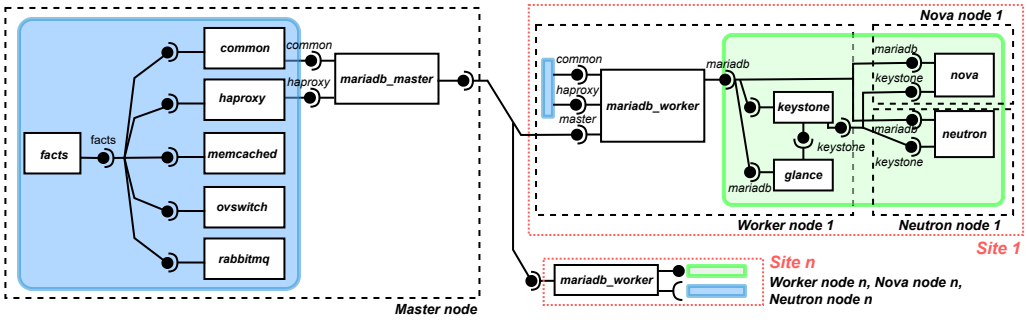

Figure 1: Assembly of a multi-site OpenStack with a Galera cluster of distributed MariaDB databases. One node holds a Galera master and is connected to n sites. Each site is composed of three nodes. The content of the green and blue rectangles are hidden in additional sites for the sake of readability.

section, we give some required background on BALLET. We illustrate by using our motivating case study OPENSTACK.

3.1 Usage of Ballet for developers

BALLET allows developers to define the life cycle of software resources and the interactions a DevOps can have with them. They can specify coarse-grained operations, such as deployment, updates, and destruction, of resources. To do this, they design *control component* (or *components*), which, when instantiated, manage the resource life cycle, similar to CRUD operations in IaC tools. A component is modeled as a set of *places* and *transitions*. Places represent milestones in the component's life cycle, while transitions define the actions (e.g., admin commands) needed to move from one place to another. Each component starts at a *start place* (*e.g.*, fully destroyed state) and reaches a *running place* when fully operational. To expose components to DevOps teams, BALLET introduces *behaviors*. A behavior is an interface to execute a set of transitions. When a behavior is triggered, all associated transitions must complete for it to be considered fulfilled. Some transitions require external information to be executed. Developers specify these dependencies through *use ports*, which group places and transitions needing external inputs. Similarly, components can provide information to others using *provide ports*. BALLET follows a model inspired (but different) from *Petri nets* [7, 9], where tokens move through places and transitions of an active behavior. As tokens progress, they activate and deactivate ports, enforcing synchronization between components.

Figure 2 gives a graphical representation of five component instances, which can be deployed with versions `v1`, `v2`, and `v3`. These five components represent the main components of one site of Figure 1. For example, `mdbmaster` is a component instance of *MariaDB* master. Its lifecycle is composed by seven places (`deployed` with three different versions, `interrupted`, `bootstrapped`, `configured`, and `initiated`), and nine transitions (arrows between group of places) that belong to five different behaviors, each represented by one color of transition and listed in the bottom right corner of the component (`deploy`, `interrupt`, `pause`, `update`, and `uninstall`). This definition of behavior is simplified: in reality, one `deploy` behavior per version has to be specified. For exchanging information with external components, `mdbmaster` has four provide ports and four use ports. The provide port `service` is bound to the group of places `deployed_v`$_i$, and each `deployed_v`$_i$ is bound to a provide port and a use port, creating version dependency between components. In the representation, for clarity reasons, ports associated with a group of places are represented in gray, while those associated with a single place are depicted in plain black. Finally, the use port `common` specifies that, to be bootstrapped, `mdbmaster` has to wait for the provide port `service` of `cmnmaster` to be activated.

3.2 Usage of Ballet for DevOps

BALLET provides a *declarative* approach for DevOps teams to manage infrastructure reconfiguration. Instead of specifying step-by-step transitions or behaviors, which would need a full comprehension of the components' life cycle, they define the desired system state for their respective subset of resources, and BALLET instances determine in a decentralized and collaborative manner the necessary actions to achieve it on each subset. The first input for DevOps teams is the specification of a target *assembly*, that is, a list of components with their connections using ports. The second input is a set of *reconfiguration goals*, which describe the desired outcome of the reconfiguration on their subpart of resources. Unlike BALLET$^+$ (Section 4), BALLET does not handle concurrent submissions of goals for more than one instance at a time. BALLET supports three types of goals:

- *State goals*: Define the expected state of the local components (*i.e.*, resources) at the end of the reconfiguration, such as whether they should be *running*, *destroyed*, or unchanged.
- *Port goals*: Specify the status of the local components' ports (active, or inactive) at the end of the reconfiguration.
- *Behavior goals*: Indicate specific behaviors that must be executed during the reconfiguration, such as updating a component.

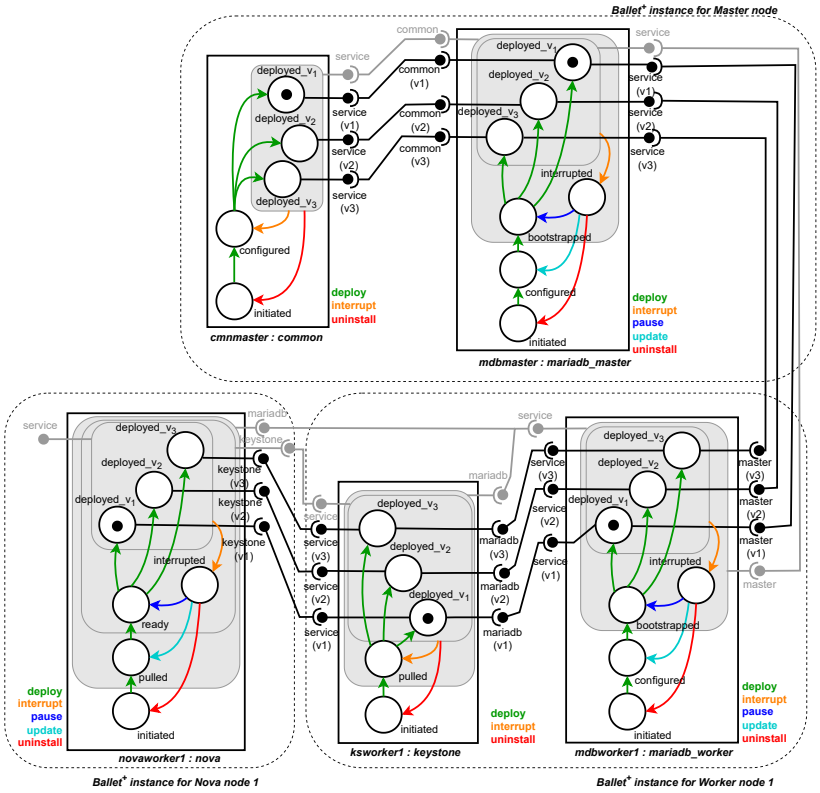

Figure 2: Control components of `Common`, `MariaDB_master`, `MariaDB_worker`, `Keystone` and `Nova` with version dependencies

Listing 1: Example of goals for updating 'common' to v2 on the Master node in Fig. 1 in Yaml

```yaml
components:
  - component: cmnmaster
    status: deployed_v2
ports:
  - forall
    port: service
    status: active
```

Listing 1 gives an example of submitted goals for the Ballet instance responsible for the Master node of Figure 1: `cmnmaster` must reach the place `deployed_v2`; and for all components handled by this Ballet instance (*i.e.*, `cmnmaster` and `mariadb_master`) ports named `service` must be `active` at the end of the reconfiguration. Once the assembly and goals are specified, all Ballet instances collaborate to plan and execute the required reconfiguration steps automatically.

3.3 Ballet's engine

Ballet processes the reconfiguration by combining the component definitions, provided by developers, with the assembly and goals, specified by DevOps teams. The reconfiguration begins by identifying the *roots*, *i.e.*, the components where a behavior must be executed to fulfill a reconfiguration goal. For example, in Listing 1 there is a single root component: `cmnmaster`. Ballet instances first mutualize the list of these components, establishing the starting points for the planning phase. Then, each instance computes a reconfiguration plan using a decentralized planner (*i.e.*, a dedicated planner runs on each instance, with local decision-making).

To determine the reconfiguration plan, each instance of Ballet infers the behaviors that must be executed to satisfy the goals. This inference relies on a constraint programming (CP) model, where each component's life cycle is represented as an automaton encoding the possible sequences of behaviors in the component. The resolution process identifies a sequence of behaviors (a valid *word* in the automaton) that aligns with the specified goals.

Executing the behaviors determined by the constraint solver will modify the status of component ports (turning an active port inactive, or vice versa). When such changes impact other components managed by another instance of Ballet, a message is sent to the relevant instances to enrich their local constraint model. These messages, which propagate through the system using a *gossip-based approach*, ensure that all nodes have the necessary information to adjust their local reconfiguration plans accordingly. Upon receiving new

Listing 2: Example of goals for updating 'nova' to v3 on Nova node 1 in Fig. 1 in Yaml

```yaml
components:
  - component: novaworker1
    status: deployed_v3
ports:
  - forall
    port: service
    status: active
```

constraints, affected instances of BALLET recompute their local resolutions, iterating the process until all dependencies are properly handled.

To be able to decide the end of the planning phase, BALLET uses an *acknowledgment mechanism*. Messages are validated back to the root components, leading to the end of the planning phase. Once all nodes have computed their reconfiguration plans, the system proceeds with a decentralized execution of the planned actions. Technical details on the execution model are omitted in this paper, but can be found in [4, 10].

4 Conflict Management

4.1 Overview through our case study

Considering the scenario proposed in Section 2, the goals submitted by DevOps teams are specified in Listings 1 and 2. The component instance `cmnmaster` must reach the `deployed_v2` state while `novaworker1` must be in `deployed_v3`. In addition, the goals specify that for all components, the port `service` must be active at the end of the reconfiguration. In this example, the set of goals cannot be satisfied. Indeed, due to inter-dependencies, having nova in `deployed_v3` while `cmnmaster` is in `deployed_v2` is impossible. On the one hand, if `cmnmaster` is reconfigured into `deployed_v2`, a constraint on `mdbmaster` is first spread to `mdbworker1`, and so on. On the other hand, reconfiguring `novaworker1` to `deployed_v3` spreads the constraint on `ksworker1`, etc., thereby pushing `mdbworker1` and other related components into `deployed_v3`. Internally, these dependencies are modeled as constraints on the ports. Depending on the speed of diffusion from each root, an unsatisfiable resolution can occur on any component. For example, the following constraints may enrich `mdbmaster` for a given resolution and communication times: $status(common(v2), s_n) = active$ and $status(service(v3), s_n) = active$; which are unsatisfiable due to the component's definition.

4.2 Planner constraint model

The local inference of behaviors to execute in BALLET$^+$ is modeled as an optimization problem in constraint programming (see Section 3), which ensures that the output sequence of behaviors (e.g., plan) respects the component's life cycle, fulfills the specified goals, and considers the additional constraints diffused during the planning phase.

Model 1 gives a general overview of the core constraints to be satisfied. In constraint (1), COSTREGULAR ensures the sequence of behaviors B is accepted by the given automaton Π [11], starting from state s_{init} and ending in S_{goal}, while also minimizing the sum of costs associated with each fired transition. As expressed in (2), a resolution allows us to determine the status of a port in a particular state s_i, to assert that the port is active or not, according to the component's definition. Goals are expressed using additional constraints: the state goals are modeled by reducing the possible ending states S_{goal} to the target state s_{goal} (Constraint (3)) as specified by the DevOps; behavior goals are expressed using the COUNT$(b, B, >, 0)$ [12] (Constraint (4)) meaning the behavior b must appear at least once in B; and Constraint (5) models goals on the status of ports by specifying if the status a port p should be (after running the last behavior of the reconfiguration) 1 to be active, or 0 to be inactive.

Model 1 A CP model for finding a sequence of behaviors to execute from an automaton.

Minimize C subject to

$$\text{COSTREGULAR}(B, \Pi, \mathcal{C}, s_{init}, S_{goal}) \quad (1)$$

$$status(p, s_i) = \texttt{active} \Leftrightarrow s_i \in S_p \quad (2)$$

$$S_{goal} = \{s_{goal}\} \quad (3)$$

$$\text{COUNT}(b, B, >, 0) \quad (4)$$

$$status(p, s_{m+1}) = \Gamma_p \quad (5)$$

where Π an automaton, with $\mathcal{C}$ costs
B a sequence of m behaviors
$\Gamma_p \in \{0 : \texttt{inactive}, 1 : \texttt{active}\}$

The model is further enriched based on incoming messages from other BALLET$^+$ instances. Each message gives insight about a port status change, that is, the new port status along with the behavior responsible for this change. Such a message results in additional transitions in the local automaton, reflecting synchronization requirements, and additional constraints on the word matching with the automaton. In this paper for clarity we omit the details regarding this model enrichment.

4.3 Refined constraint model

In BALLET$^+$, we initially attempt to solve the model using a constraint solver. If the solver detects that the model is unsatisfiable, we then run a QuickXplain algorithm [13] on the model. This algorithm relies on a dichotomous search to compute a subset of jointly unsatisfiable constraints. It provides users with sound information in situations where there is no solution. In order to enhance our ability to interpret unsatisfiability, the original Constraint Programming (CP) model needs to be reformulated in this second step. This modification consists of the decomposition of the global constraints to be as close as possible to the DevOps specifications.

It should be noted that Model 1 is based on so-called *global* constraints (COSTREGULAR and COUNT constraints), which encode the initial expression of the problem described by the users. On the one hand, the usage of global constraints aims to improve the efficiency of the model, since they are

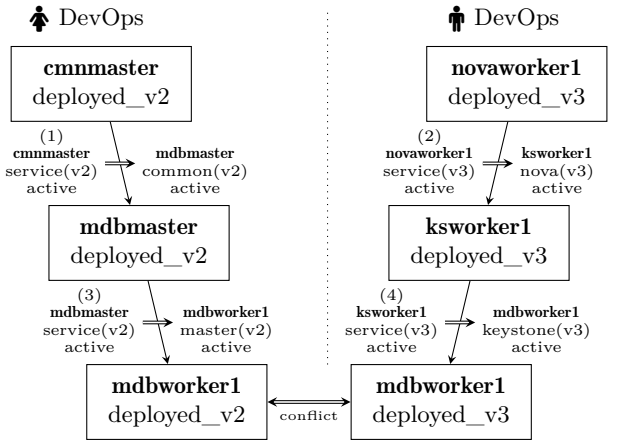

Figure 3: Tree of conflict causality in the reconfiguration example of Listings 1 and 2

equipped with powerful filtering algorithms. On the other hand, the global aspect of these constraints limits the ability for QuickXplain to provide easier results to interpret the source of unsatisfiability for the user.

For example, the COUNT constraint (4), which forces the appearance of behaviors b in the sequence B can be decomposed into a sum constraint (6) and an arithmetic constraint (7) as follows.

$$sum_b = \Sigma_m^{i=1}(\texttt{ if } b_i == b \texttt{ then 1 else 0}) \quad (6)$$

$$sum_b > 0 \quad (7)$$

This evolution of our modeling approach allows for clearer explanations of the underlying causes when a model fails to satisfy given constraints.

In addition, each constraint is tagged by its origin to facilitate unsatisfiability explanation. The possible origins are the following:

- Component definition: Constraints that are inherent to the definition of a component (e.g., correspondence between states and ports status);
- Reconfiguration goal: Constraints that are derived from the goals specified for reconfiguration;
- Inferred from message: Constraints that are inferred from specific messages received during the decentralized planning process.

Next, Section 4.4 explains how tagging constraints allows to trace backtrack the source of the unsatisfiability.

4.4 Unsatisfiability backtracking

Once the QuickXplain (Qx) algorithm identifies the unsatisfiable state, we construct an upward tree to trace the cause of the conflict. An example of a tree with the local vision of each root BALLET$^+$ instance is given in Figure 3. Each vertex of the tree represents the concerned component and its behavior that led to conflict. Each edge is labeled with the logical implication that led to the need for the next vertex. To build this graph, the general idea is to backtrack from the unsatisfiable (unsat) model by identifying which goals led to the constraints that caused the model to become unsatisfiable. The global tree does not exist. It is partly built by each root component (at least two) that led to the conflict.

The example trees depicted in Figure 3 are constructed as illustrated in Figure 4. On one side, message (1) is sent from `cmnmaster` to `mdbmaster`. It implies a message (3) from `mdbmaster` to `mdbworker1`. On the other side, (2) is sent from `novaworker1` to `ksworker1`. It implies a message (4) from `ksworker1` to `mdbworker1`. When the conflict occurs on `mdbworker1`, messages (3) and (4) are identified as the cause. A refusing message (5) is then sent, containing these causes. On `mdbmaster` enriches the case with (1), since (1) is responsible for (3) (resp. `ksworker1` enriches the case with (2), since (2) is responsible for (4)), and sends (6) to `cmnmaster` (resp. (7) to `novaworker1`). The trees have been fully constructed at this point, and then relevant messages can be returned to DevOps teams.

Figure 3 depicts the two (reuniting) trees built by the two root components `cmnmaster` and `novaworker1` representing the following. The submitted goals (Listing 1) on the Master node enforce `cmnmaster` to end its reconfiguration in `deployed_v2` state. The message (1), indicating that the provide port `service(v2)` will be active at the end of the reconfiguration, has been sent. As a consequence, because of the defined goals, `mdbmaster` must have its use port `common(v2)` active at the end of the reconfiguration. To do so, `mdbmaster` must end in `deployed_v2` state. Again, the message (3) indicates `service(v2)` will be active on `mdbmaster`. It implies `mdbworker1` to activate the port `master(v2)` at the end of the reconfiguration, which means `mdbworker1` will end the reconfiguration in `deployed_v2` state On the other side, the goals on Nova (Listing 2) enforce `novaworker1` to reach the `deployed_v3`. Its incidence on its provide port `service(v3)` is sent by message (2). `ksowrker1` infers to activate its use port `nova(v3)`, which enforces `ksworker1` to end the reconfiguration in `deployed_v3`. Again, the message (4) indicates `service(v3)` will be on `ksworker1`. As a consequence, `mdbworker1` has to activate its use port `keystone(v3)`, which enforces `mdbworker1` to end the reconfiguration in `deployed_v3`. This conflicts with the other constraint on `mdbworker1`'s final state.

This tree construction process is done by diffusing messages between the nodes, incrementally adding a layer until reaching the goals submitted by the DevOps teams. Once a set of constraints is decided to be responsible of the conflict, the origin is determined. Then, if the constraint was inferred from a message, this message is refused with its causality. Recursively, the origin of the refused message is determined, and the causality is incremented until all messages propagated from the DevOps goals that caused the conflict have been covered. Once the tree is complete, it can be returned as output to the DevOps team. DevOps teams can then coordinate to agree on a new reconfiguration plan.

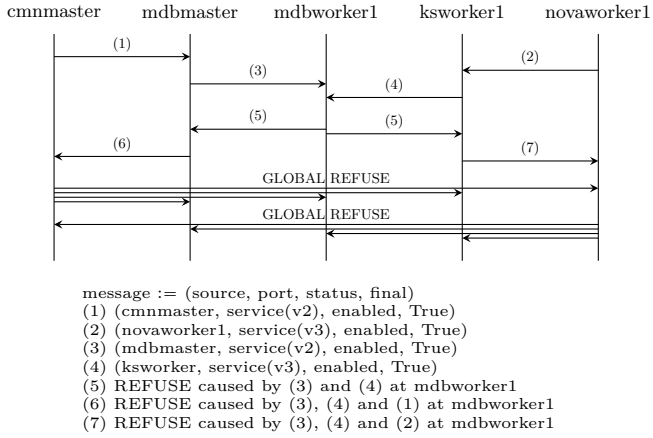

Figure 4: Time diagram of message exchanges for unsatisfiability management

5 Experimental Evaluation

5.1 Implementation and general setup

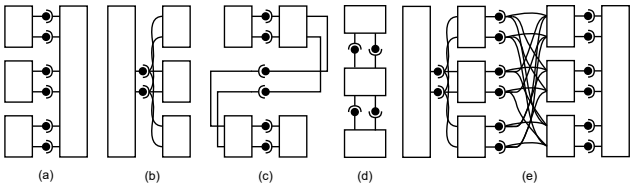

Figure 5: The five assembly topologies used in synthetic examples: (a) central-user (*c-user*), (b) central-provider (*c-provider*), (c) linear (*linear*), (d) circular (*circular*), and (e) stratified (*stratified*).

Our implementation of BALLET$^+$ is accessible at the following link https://anonymous.4open.science/r/Ballet-0181/.

BALLET$^+$ extends the core of BALLET with 2500 LoC using PYTHON 3.11.2. Communications between nodes rely on `grpcio` (version 1.58.0). To solve our constraint programming problems, we adopted two complementary approaches: (i) We use the high-level, solver-independent constraint modeling language MiniZinc [14] (version 2.7.6), interfaced through the `minizinc` PYTHON package (version 0.9.0). Among the solvers available in MiniZinc, the open-source solver CHUFFED [15] (version 0.12.1) has demonstrated the best performance in our case and is used to solve our initial CP models. Chuffed is used as a black box solver, i.e., without modifying its default search strategy. (ii) In case of unsatisfiability with Chuffed, we use the JAVA-based Choco solver [16] to handle constraint resolution. In particular, we employ Choco's QuickXplain algorithm to identify the minimal set of constraints responsible for making the model unsatisfiable. The adopted strategy for QuickXplain explores variables in their input order, assigning them the lowest possible value first. To use Choco to our specific needs, we extended its functionality with 650 LoC. It includes a label management of constraints, needed for identifying them to trace back their origin, and the support of loading BALLET$^+$ models.

Our experiments are performed on a real infrastructure offered by the experimental platform Grid'5000, thus facing real distribution and communications between nodes. All evaluations were conducted on the `Gros` cluster[5]. `Gros` has 124 hosts equipped with one 18-core Intel Xeon Gold 5220 with 96 GiB of memory, 480 GB SSD + 960 GB SSD, and a transfer rate of 2 x 25 Gbps (SR-IOV) on the network. EnosLib [17] scripts have been used as a front interface for BALLET$^+$ nodes. The scripts and results are available at https://anonymous.4open.science/r/UCC25-experiments-88F1/.

5.2 Results over topological synthetic examples

Setup. To test our model explainability and protocol across diverse scenarios, we categorized assemblies into five distinct topologies, as described below. Every component in these assemblies is equipped with behaviors for deployment, updating, suspension, and deletion.

(a) In central-user assemblies (*c-user*), a central user component connects to multiple provider components, each with a pair composed of a provide port linked to a use port on the central component.
(b) In central-provider assemblies (*c-provider*), a single central provider component connects to multiple user components, each with a pair made of a provide port and the corresponding use ports of the user.
(c) Linear assemblies form a chain of components (*linear*), where each component provides a port that is connected to the next component's use port in sequence. The first component, with only provide ports, is referred to as the provider, while others are referred to as transformers.
(d) In circular assemblies (*circular*), each pair of components is mutually dependent, each providing a port and using the port of the other component, thus forming a circular chain of dependencies. The two extreme components are referred to as the provider and the user, while others are referred as transformers.
(e) The stratified architectures (*stratified*) combine (a), (b), and (c) topologies into a complex one, where components are organized into levels. Each level contains up to three components. Each component in a level has a pair of provide ports connected to the use ports of every component in the level above, allowing a provide port to connect to up to three use ports. The two extreme components are referred to as the provider and the end-user, while others, that are intermediate users, are referred to as mid-users.

Figure 5 illustrates these five topologies, with internal component connections omitted for clarity. For each topology,

[5]https://www.grid5000.fr/w/Hardware

topology	**#components**	**initial states**	**SAT-case** goals	**UNSAT-case** goals
c-user	1 user; 15 providers	user: running provider$_i$: running	user: (state) running provider$_i$: (state) running + (behavior) update	user: (state) running provider$_i$: (state) uninstalled
c-provider	1 provider; 15 users	user$_i$: running provider: running	user$_i$: (state) running provider: (state) running + (behavior) update	user$_i$: (state) running provider: (state) uninstalled
linear	1 provider; 15 transformers	provider: running transformer$_i$: running	provider: (state) running + (behavior) update transformer$_i$: (state) running	provider: uninstalled transformer$_i$: (state) running
circular	1 provider; 15 transformers; 1 user	provider: running transformer$_i$: running user: running	provider: (state) running + (behavior) update transformer$_i$: (state) running user: (state) running	provider: uninstalled transformer$_i$: (state) running user: (state) running
stratified	1 provider 15 mid-users 1 user	provider: running mid-user$_i$: running user: running	provider: (state) running + (behavior) update mid-user$_i$: (state) running user: (state) running	provider: uninstalled mid-user$_i$: (state) running end-user: (state) running

Table 1: Testing scenarios for assembly topologies with goals for a satisfiable case (SAT-case), and an unsatisfiable case (UNSAT-case)

topology	**SAT-case** time (σ)	**UNSAT-case** time (σ)
c-user	23.362s (3.79)	19.8341s (0.91)
c-provider	11.1907s (1.15)	9.7482s (0.91)
linear	34.9178s (2.40)	29.0932s (1.64)
circular	14.0585s (0.52)	1.9996s (0.61)
stratified	19.9338s (1.19)	24.0926s (1.32)

Table 2: Average times (for 10 runs, with standard deviation) to entirely solve the SAT and UNSAT cases for each scenario on topologies.

topology	**SAT** solving time (σ)	**UNSAT** solving time (σ)
c-user	4.1970s (1.40)	1.3841s (0.23)
c-provider	2.0057s (0.30)	0.7192s (0.17)
linear	0.7192s (0.12)	0.6753s (0.11)
circular	1.0587s (0.63)	0.4983s (0.10)
stratified	1.8807s (0.25)	0.7918s (0.14)

Table 3: Average time of the maximum local solving time for one component (for 10 runs, with standard deviation) for each scenario on topologies.

we ran two reconfigurations: a satisfiable one (named SAT-case) and an unsatisfiable one (named UNSAT-case). The reconfiguration goals are described in Table 1. To ensure Ballet$^+$ scalability, each component in our experiments was managed by its dedicated Ballet$^+$ instance, running on a separate physical node.

Result. For all our cases, Ballet$^+$ is capable of detecting unsatisfiability in the reconfiguration process. When an unsatisfiable situation arises, Ballet$^+$ performs a local explanation to identify the set of conflicting constraints. This explanation is correctly propagated back along the chain of constraint diffusion. The returned explanation is representing correctly the causality tree. The correctness of this explanation structure has been manually verified. Table 2 illustrates the average execution times, with their standard deviation, for 10 runs for each topology and reconfiguration goal.

More detailed results provided in Table 3 indicate that detecting an unsatisfiable model and returning its set of conflicting constraints does not introduce significant overhead. To evaluate this, we compared the maximum local solving time dedicated to model resolution in both satisfiable (SAT) and unsatisfiable (UNSAT) scenarios. In SAT cases, this corresponds to the resolution time needed to find a valid sequence of behaviors, while in UNSAT cases, it includes both the detection of unsatisfiability and the explanation phase using QuickXplain. Results show that the solving times remain within the same order of magnitude. For instance, for the *stratified* topology, the maximum resolution time (in average) for a satisfiable scenario was 1.88s, while it was 0.79s for the unsatisfiable scenario.

Discussion. The standard deviation in solving times can be significant in satisfiable cases because the model is iteratively solved each time new messages are received. To prevent excessive resolutions, a random delay (between 0 and 2 seconds) is introduced between each iteration, allowing the system to aggregate multiple constraints before attempting a new resolution. As a result, the total number of resolution attempts is directly influenced by the frequency at which messages are received. In any case, the overall planning time remains driven by communication overhead and the synchronization required between nodes, rather than the computational complexity of constraint resolution itself.

We experimented with topologies of sizes comparable to our OpenStack motivating examples (around 15 components) to evaluate the behavior of Ballet$^+$ across different topologies.

5.3 Results over multi-site OpenStack with Galera cluster

Setup. To test our solution on a realistic scenario, we submitted goals on the assembly represented in Figure 1, with one Master node and one Worker site. The experiment was set up with a dedicated physical node on `Gros` cluster for each node represented in Figure 1: one for the Master node, one for the Worker node, one for the Nova node, and one for the Neutron node. In our experiments, each node is managed by one Ballet$^+$ instance. Starting from all

	SAT-case time (σ)	**UNSAT-case** time (σ)
Planning time	8.4922s (1.29)	10.4161s (1.45)
Solving time	3.85s (0.11)	3.2124s (0.86)

Table 4: Average time (for 10 runs, with standard deviation) of the full planning process, and average time of the maximum local (for Ballet$^+$ instance) solving times for satisfiable and unsatisfiable scenarios over the multi-site OpenStack case.

components in a running state, we compare two scenarios. First, we only submit the goal from Listing 1, leading to a non-conflicting reconfiguration. Then, as a second scenario, we submit goals from Listing 1 and Listing 2, leading to a conflictual reconfiguration.

Result. For the OpenStack case, experiments have demonstrated that Ballet$^+$ successfully detects and explains conflicts in the reconfiguration process. The average total execution time was 8.49s for a satisfiable (SAT) case and 10.42s for an unsatisfiable (UNSAT) case. When focusing on model resolution, the maximum local solving time recorded was 3.85s in the satisfiable case and 3.21s in the unsatisfiable case, showing there is no overhead introduced by conflict detection. Table 4 summarizes these results and includes the standard deviation of records.

Discussion. To model our test case, we represented each different version of the deployed state as distinct states, each associated with different ports. A key-value configuration mechanism would be more adapted as in IaC tools (*e.g.*, Terraform, Pulumi). Indeed, the component definitions in Ballet (and Ballet$^+$) are primarily designed to manage life cycles rather than parameterized configurations. As a result, while our controller definitions for planning are suitable for three versions, they would not be directly applicable for executing concrete updates through time. Additionally, our evaluation does not assess scalability on a very large set of resources (*i.e.*, components). However, as in topological cases, the planning phase ends when an unsatisfiable local model is detected. This early termination mechanism suggests that increasing infrastructure size would not impact planning time in such cases. Nonetheless, further experiments are necessary to validate this assumption and analyze the overall behavior of our approach on a larger scale.

6 Related Work

To the best of our knowledge, Ballet$^+$ is the only existing contribution offering a decentralized declarative cross-DevOps solution that detects and explains conflicts in concurrent reconfigurations.

DevOps IaC (Infrastructure as Code) solutions offer reconfiguration capacities in a declarative manner. Examples include Terraform [18], Puppet [19], and Pulumi [20], which support declarative provisioning and incorporate a planning phase. Orchestration tools like Kubernetes [21] also provide planning capabilities, such as automatic scalability and failure recovery. Nonetheless, these tools use a centralized approach for both planning and execution. Dynamic reconfiguration is also addressed in component-based software engineering (CBSE) [2]. Aeolus [22] and Concerto [8, 23] offer high flexibility in defining lifecycles and provide planners that automatically generate reconfiguration programs from specified goals. However, these solutions are also centralized.

Muse [3] enables the automated and decentralized deployment, update, and deletion of resources on top of Pulumi. However, it does not manage unsatisfiable cases. Efforts to decentralize container-based orchestrators like Kubernetes primarily focus on decentralizing the scheduler rather than the planning and coordinating reconfiguration requirements from DevOps teams [24–26]. Therefore, these contributions fall outside our scope.

Some tools automatically determine requirements without DevOps intervention, using methods like constraint programming and SMT solvers [27, 28]. Our work complements these approaches by focusing on decentralized computation and execution of reconfiguration plans from a given set of requirements.

Moreover, in some processes, the definition of goals is not done manually. For instance, using a MAPE-K approach [29], the analysis phase is responsible for providing goals that make reconfiguration satisfiable. Our extended protocol, which manages unsatisfiability, further differentiates our approach by allowing us to handle and explain why certain reconfiguration goals cannot be met, thereby facilitating better coordination and planning among DevOps teams.

In CP community, the study of finding the Minimal Unsatisfiable Subsets (MUS) [30] of constraint within a model, and provide a comprehensive interpretation [31] has been studied for other applications than IaC. Bogaerts et al. [32] propose a framework for step-wise explanations in constraint satisfaction problems, focusing on making inference steps understandable for humans. Espasa et al. [33] developed Demystify, which uses MUS to generate step-by-step explanations for solving puzzles. Sreedharan et al. [34] introduce methods for providing post-hoc symbolic explanations for sequential decision-making problems with inscrutable representations. Additionally, Sreedharan et al. [35] discuss generating personalized contrastive explanations for AI agent behavior using state abstractions.

7 Conclusion and Perspectives

In this paper, we have presented Ballet$^+$, a novel approach for managing conflicting goals in cross-DevOps declarative reconfigurations. Ballet$^+$ is the first attempt to introduce conflict management in decentralized reconfigurations. Ballet$^+$ leverages constraint programming techniques, specifically Chuffed exploration combined with the QuickXplain algorithm of Choco, to successfully detect, explain, and trace back the source of conflicting concurrent reconfigurations. Our results validate the feasibility and efficiency of integrating conflict management into decentralized declarative reconfigurations with Ballet$^+$ instances for various types of resource topologies.

Future work will focus on enhancing our approach by providing mechanisms to assist DevOps teams in resolving conflicts through consensus-based suggestions. This requires a more detailed understanding of service configurations. To this purpose, future work could either target tools with parameterized configurations, such as Pulumi, to properly model versioned components in decentralized reconfiguration (e.g., Mjuz [3]), or extend BALLET$^+$ and its current constraint model with parameters.

The future research direction will also focus on decision-making strategies for scenarios where only a subset of goals can be fulfilled. In such scenarios, only the infeasible goals would be reported to the DevOps teams. In that sense, MaxSat [36] could help identify a maximal set of feasible goals by excluding the minimal set of conflicts, providing support to DevOps teams. Overall, our work lays the foundation for robust conflict management in decentralized reconfigurations, opening the door to more advanced reasoning and decision-support mechanisms for cross-DevOps environments.

References

[1] R. Wang, *Infrastructure as Code, Patterns and Practices: With Examples in Python and Terraform*, ser. ITpro collection. Manning, 2022.

[2] H. Coullon, L. Henrio, F. Loulergue, and S. Robillard, "Component-based distributed software reconfiguration: A verification-oriented survey," *ACM Comput. Surv.*, 2023.

[3] D. Sokolowski, P. Weisenburger, and G. Salvaneschi, "Automating serverless deployments for devops organizations," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[4] J. Philippe, A. Omond, H. Coullon, C. Prud'Homme, and I. Raïs, "Fast Choreography of Cross-DevOps Reconfiguration with Ballet: A Multi-Site OpenStack Case Study," in *SANER 2024 - IEEE International Conference on Software Analysis, Evolution and Reengineering*. Rovaniemi, Finland: IEEE, Mar. 2024, pp. 1–11.

[5] R.-A. Cherrueau, A. Lebre, D. Pertin, F. Wuhib, and J. M. Soares, "Edge Computing Resource Management System: a Critical Building Block! Initiating the debate via OpenStack," in *HotEdge 2018 - USENIX Workshop on Hot Topics in Edge Computing*, 2018. [Online]. Available: https://hal.science/view/index/docid/1812747

[6] "Galera cluster," https://galeracluster.com/, 2023, : 2023-24-10.

[7] M. Chardet, H. Coullon, and S. Robillard, "Toward Safe and Efficient Reconfiguration with Concerto," *Science of Computer Programming*, 2021.

[8] M. Chardet, H. Coullon, and C. Pérez, "Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto," in *CCGrid 2020 : 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2020.

[9] H. Coullon, C. Jard, and D. Lime, "Integrated Model-checking for the Design of Safe and Efficient Distributed Software Commissioning," in *Integrated Formal Methods*, ser. Integrated Formal Methods, Bergen, Norway, Dec. 2019, pp. 120–137.

[10] F. Arfi, H. Coullon, F. Loulergue, J. Philippe, and S. Robillard, "An Overview of the Decentralized Reconfiguration Language Concerto-D through its Maude Formalization," in *Electronic Proceedings in Theoretical Computer Science*, ser. Electronic Proceedings in Theoretical Computer Science, Groningen, Netherlands, Jun. 2024, pp. 1–18, satellite workshop of DisCoTec 2024. [Online]. Available: https://inria.hal.science/hal-04572043

[11] G. Pesant, "A regular language membership constraint for finite sequences of variables," in *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, ser. Lecture Notes in Computer Science, M. Wallace, Ed., vol. 3258, 2004.

[12] "count," 2014, accessed: 2024-12-07. [Online]. Available: https://sofdem.github.io/gccat/gccat/Ccount.html

[13] U. Junker, "Quickxplain: Preferred explanations and relaxations for over-constrained problems," ser. AAAI'04. AAAI Press, 2004, p. 167–172.

[14] "Minizinc 2.7.6," 2023, accessed: 2023-07-12. [Online]. Available: https://www.minizinc.org/resources.html

[15] G. Chu, P. J. Stuckey, A. Schutt, T. Ehlers, G. Gange, and K. Francis, "Chuffed, a lazy clause generation solver," 2018. [Online]. Available: https://github.com/chuffed/chuffed

[16] C. Prud'homme and J.-G. Fages, "Choco-solver," *Journal of Open Source Software*, vol. 7, no. 78, p. 4708, 2022.

[17] B. Jonglez, M. Simonin, J. Philippe, and S. M. Kaddour, "Multi-provider capabilities innbsp;enoslib: Driving distributed system experiments onnbsp;thenbsp;edge-to-cloud continuum," in *Distributed Applications and Interoperable Systems: 25th IFIP WG 6.1 International Conference, DAIS 2025, Held as Part of DisCoTec 2025, Lille, France, June 16–20, 2025, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2025, p. 25–42.

[18] Y. Brikman, "Terraform: Up and running," 2022.

[19] "What is puppet," http://puppetlabs.com/puppet/what-is-puppet, 2014, : 2023-24-10.

[20] "Pulumi," https://www.pulumi.com/, 2023, accessed: 2024-12-07.

[21] B. Burns, J. Beda, K. Hightower, and L. Evenson, "Kubernetes: up and running," 2022.

[22] R. Di Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro, "Aeolus: A component model for the cloud," *Information and Computation*, 2014.

[23] S. Robillard and H. Coullon, "SMT-Based Planning Synthesis for Distributed System Reconfigurations," in *FASE 2022 : 25th International Conference on Fundamental Approaches to Software Engineering*, 2022.

[24] M. A. e. a. Tamiru, "mck8s: An orchestration platform for geo-distributed multi-cluster environments," in *2021 International Conference on Computer Communications and Networks (ICCCN)*, 2021.

[25] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018.

[26] L. L. Jiménez and O. Schelén, "Docma: A decentralized orchestrator for containerized microservice applications," in *2019 IEEE Cloud Summit*, 2019.

[27] J. A. Hewson, P. Anderson, and A. D. Gordon, "A declarative approach to automated configuration," in *Proceedings of the 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques*, ser. lisa'12. USA: USENIX Association, 2012, p. 51–66.

[28] E. Ábrahám, F. Corzilius, E. B. Johnsen, G. Kremer, and J. Mauro, "Zephyrus2: On the fly deployment optimization using smt and cp technologies," in *Dependable Software Engineering: Theories, Tools, and Applications*, 2016.

[29] A. Alidra, H. Bruneliere, H. Coullon, T. Ledoux, C. Prud'Homme, J. Lejeune, P. Sens, J. Sopena, and J. Rivalan, "SeMaFoR - Self-Management of Fog Resources with Collaborative Decentralized Controllers," in *SEAMS 2023 - IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems*. Melbourne, Australia: IEEE, May 2023, pp. 25–31.

[30] K. Leo and G. Tack, "Debugging unsatisfiable constraint models," 05 2017, pp. 77–93.

[31] S. Dev Gupta, B. Genc, and B. O'Sullivan, "Explanation in constraint satisfaction: A survey," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Z.-H. Zhou, Ed., 8 2021, pp. 4400–4407, survey Track.

[32] B. Bogaerts, E. Gamba, and T. Guns, "A framework for step-wise explaining how to solve constraint satisfaction problems," *Artificial Intelligence*, vol. 300, p. 103550, 2021.

[33] J. Espasa, I. P. Gent, R. Hoffman, C. Jefferson, A. M. Lynch, A. Salamon, and M. J. McIlree, "Using small muses to explain how to solve pen and paper puzzles," 2023. [Online]. Available: https://arxiv.org/abs/2104.15040

[34] S. Sreedharan, U. Soni, M. Verma, S. Srivastava, and S. Kambhampati, "Bridging the gap: Providing post-hoc symbolic explanations for sequential decision-making problems with inscrutable representations," 2022. [Online]. Available: https://arxiv.org/abs/2002.01080

[35] S. Sreedharan, S. Srivastava, and S. Kambhampati, "Using state abstractions to compute personalized contrastive explanations for ai agent behavior," *Artificial Intelligence*, vol. 301, p. 103570, 2021.

[36] C. M. Li and F. Manya, "Maxsat, hard and soft constraints," in *Handbook of satisfiability*. IOS Press, 2021, pp. 903–927.