SMT-Based Planning Synthesis for Distributed System Reconfigurations

Simon Robillard, Hélène Coullon

SMT-Based Planning Synthesis for Distributed System Reconfigurations

Simon Robillard[1]($\boxtimes$) and Hélène Coullon[2]

[1] LIRMM, CNRS, Université de Montpellier, France
[2] IMT Atlantique, Inria, LS2N, Nantes, France

Abstract. Large distributed systems with an emphasis on adaptability are now considered a necessity in many domains, yet reconfiguration of these systems is still largely carried out in an ad hoc fashion, a process that is both inefficient and error-prone. In this paper, we tackle the planification problem for the reconfiguration of distributed systems in the component-based reconfiguration model Concerto. Specifically, given some tasks to execute and a desired final state of the system, we show how to compute a reconfiguration plan that guarantees satisfaction of inter-component dependencies and is also optimized for parallel execution. Our technique relies on an SMT solver to compute the required dependencies between components and ultimately schedule the reconfiguration. We illustrate the use of this technique on a variety of synthetic examples as well as a real use case in the context of an OpenStack system.

Keywords: reconfiguration, planning, synthesis, component models, distributed systems

1 Introduction

Large distributed software systems are now ubiquitous, with component-based systems (e.g., service-oriented architectures or microservices) offering a convenient way to structure large applications. Indeed, isolating functionalities in components and building systems through composition greatly enhances adaptability and scalability of applications, two important requirements for many organizations. This approach is also promoted by the massive adoption of highly-distributed computing infrastructures such as cloud and edge computing.

However, the advantages of distributed architectures come at the price of increased complexity and technical challenges related to observability, coordination, maintenance, etc. Notably, the system reconfigurations that are required to achieve adaptability commonly lead to faults. For example, a study of 597 unplanned outages that affected popular cloud services between 2009 and 2015 found that 16% of them were caused by a software or hardware upgrade [16]. The study concludes that "the complexity of cloud hardware and software ecosystem has outpaced existing testing, debugging, and verification tools". Indeed, testing and debugging methods are largely inadequate in the context of distributed systems, while the adoption of more suitable formal methods remains marginal in

industry. The latter can be attributed to the difficulty of using formal methods and tools. Yet formal methods can lighten the burden of program developers and system administrators instead of adding to it, with synthesis techniques used to generate correct-by-construction programs. In that spirit, we propose to employ a Satisfiability Modulo Theories (SMT) solver to automate the planning of reconfigurations (deployment, migrations, software updates, etc.) of component-based systems, i.e., to generate programs that coordinate the non-functional operations required to perform such reconfigurations. There have been some attempts to synthesize reconfiguration programs for component-based systems (some of them relying on an SMT solver), but they either target ad hoc, non-executable models [20], or are limited to specific cases such as deployment [22], where the problem of executing parallel tasks is reduced to finding a precedence order. In contrast, our work targets the full scope of the component-based reconfiguration model Concerto [9], which provides a formally-defined execution model with expressive constraints on parallelism, as well as a concrete execution engine, making it suitable for formal analysis and experimental work.

In Concerto, reconfigurations are driven by asynchronous *behavior* requests to components. The execution of a behavior may depend on the state of other components: such dependencies are denoted by *ports* that form the interface of components, indicating their provisions and requirements towards each other. Section 2 gives an overview of Concerto, for a more complete presentation, the reader can refer to [9]. Our goal with this work is to automatically generate reconfiguration scripts for systems of Concerto components, i.e., determine required behaviors and coordinate their execution. We take as starting point a reconfiguration goal composed of behaviors to execute over some components and a specification of the final state of the system, particularly the statuses of ports. That goal may be provided by a system administrator, or could have been generated in the context of an autonomic control loop [19]. Importantly, it is a partial specification that typically only mentions parts of the system. For example, an administrator may specify only that a certain utility component should execute a behavior to update its software, whereas the completion of this task actually requires other components to suspend and later resume their activity.

Since a reconfiguration goal can require changes in any component of a system, the search space for reconfiguration scripts grows rapidly with the number of components. To synthesize reconfigurations for large systems, we propose a novel technique that takes advantage of the nature of component-based models. It first solves the problem for each component individually, by considering the internals of the component to find relevant behaviors, under the simplifying assumption that external requirements are all satisfied. Later the method coordinates behaviors over the whole system, relying on a first-order encoding of the scheduling problem and making use of the model-finding capabilities of an SMT solver. If this step fails due to unsatisfied dependencies, individual component reconfiguration goals are refined and the process iterated. Section 3 describes this method, and Section 4 measures its performance and scalability on a variety of synthetic examples, and illustrates its applicability on a real use case.

2   Reconfiguration With Concerto

Components and Assemblies. A distributed system in Concerto is represented as an *assembly*, i.e., a collection of *components* that correspond to control entities for the elements of the system. Components are not intended to represent the functional aspects of those elements, but instead to pilot the actions (installation, maintenance, suspension of service, etc.) required to operate them during their lifespan. In other words, a Concerto component is a wrapper around a new or legacy piece of software (e.g., service, module), typically written by its developer, that acts as replacement for scripts to install and maintain it.

The structural interface of a component is provided by its *provide ports* and *use ports*. Provide ports denote services or data provided by that component when those ports are *active*, while use ports denote requirements that the component has when those ports are active. Ports can be connected in an assembly to allow the satisfaction of component requirements. Connected ports impose synchronization rules between their components: a use port cannot be activated unless connected to an active provide port (the user component may have to wait for that requirement to be fulfilled in order to continue its internal activity) and a provide port cannot be deactivated while connected to an active use port.

Internally, components are characterized by *places* representing milestones in the life cycle, and *transitions* between places, mapped to concrete reconfiguration actions (e.g., starting a virtual machine, downloading an image, etc.). The internal state of a component is given by its places: at any point during execution, one or more places are active. While a place π is active, transitions originating from it can be (simultaneously) fired, after which π ceases to be active. Conversely, a place π' becomes active after the completion of all the transitions that reach it. The completion of a transition takes a non-deterministic duration after firing, modeling the execution of the associated action. Active places also determine the statuses of ports: each port is bound to a set of places, and is active whenever one of them is active. Thus the status of ports changes according to the life cycle of the component. In graphic representations, ports are linked to the place (or set of places, denoted by rounded boxes) to which they are bound.

The last characteristic attribute of a component is its set of *behaviors*. A behavior is a subset of the transitions in a component, such that the associated subgraph is acyclic. At any point in an execution, a component may execute one behavior. Only then can the transitions in that behavior be fired. The behaviors of a component serve as its operational interface: a component may have one behavior including the actions to start it, another including the actions to update it, etc. A component can be requested to execute a behavior, which will determine its evolution and the actions that it performs. Graphically, different behaviors are represented by depicting transitions in different colors.

Figure 1 gives a graphic representation of an assembly. Component `dep1` includes three places (`uninstalled`, `installed` and `running`) and three transitions (arrows between places) that belong to three behaviors (`deploy`, `update`, and `uninstall`). Place `running` is active (denoted by a token) and bound to pro-

vide port `service`, whereas places `installed` and `running` are bound to provide port `config`. Both ports are connected to use ports belonging to `server`.
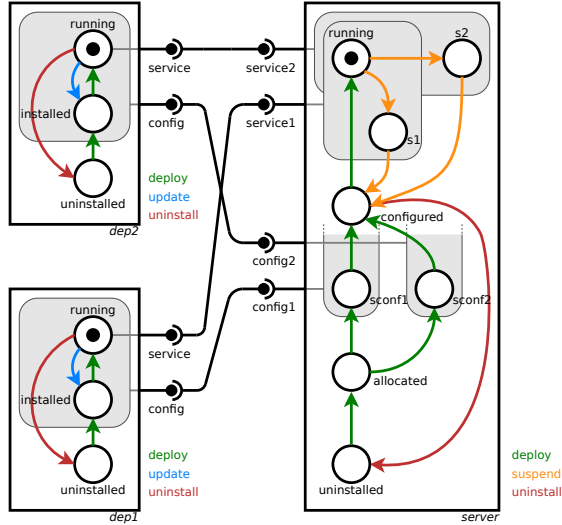

Fig. 1: A Concerto assembly with three components. For readability, the bindings of ports `config1` and `config2` are only partially depicted: they also contain places `configured`, `running`, `s1` and `s2`.

Reconfiguration Scripts. Concerto is equipped with a simple language to execute reconfigurations. Whereas a Concerto component is written by a developer, the reconfiguration language is intended to be used by system administrators or DevOps engineers. Components are piloted through asynchronous requests via the command `pushB(id, b)` that asks the component identified by `id` to execute behavior `b`. The command takes its name from the fact that requests received by a component are queued and asynchronously executed by that component in the order in which they were received. While a component executes a behavior request, transitions in that behavior are fired until the component reaches a state where none of them can be fired. The behavior request is then considered complete, and the component executes the next one, until no more requests remain. The Concerto language also provides synchronization commands: `wait(id)` blocks the execution of the reconfiguration program until the component identified by `id` has executed all behaviors requests submitted to it, and `waitAll()` blocks the execution until all components have executed all pending behavior requests. These three commands allow parallel asynchronous execution in Concerto, leading to more efficient reconfigurations. Based on the description of the components provided by their developers, Concerto can execute reconfiguration scripts , allowing for empirical performance comparisons [10].

The goal of this work is to generate a reconfiguration script using the three aforementioned commands to execute behaviors over components and bring them to a desired state. In addition to those three commands, the Concerto language also provides four usual commands to modify the topology of an assembly: create and delete components, connect and disconnect them. These operations are out of the scope of reconfiguration planning as we define it. Indeed, the decision to modify the topology of the assembly is usually taken by the same entity that determines reconfiguration goals (system administrator or autonomic analysis tool) [15, 17] rather than left to the planning phase. Furthermore, if topological changes in the assembly are deemed necessary, they can almost always be implemented through a reconfiguration script with the following steps: (i) creations of components, (ii) creations of connections, (iii) changes in component states, (iv) deletions of connections and (v) deletions of components [5, 7]. The main difficulty is to determine the operations of the third step that take the components to a safe state, in particular ensuring that none of the connections that will be deleted include an active use port. Computing a reconfiguration program to lead components to a desired state (or to have them perform some required operations) is the focus of this paper.

As an example, consider the assembly in Figure 1, where all the components are running. We wish to run software updates on dep1 and dep2, but this will deactivate their provide port service. To carry out the updates, component server must first deactivate its corresponding use ports, which is accomplished by executing its behavior suspend. Figure 2a depicts a reconfiguration script that performs this, then returns the components to a running state. No explicit synchronization is needed between the suspension of server and the updates: the execution model of Concerto ensures that the updates cannot be executed as long as the provide ports are in use. An explicit synchronization is however needed before re-deploying the server, to prevent it from reactivating its use ports before the updates start. As a side note, the ports config (that represent configuration information that is not affected by the update, such as connection information) remain active throughout the reconfiguration: the fine-grained management of dependencies in Concerto avoids a full restart of the system. This assembly also illustrates the capacity of Concerto components to execute actions in parallel: for example, after server has reached place allocated, it can fire multiple transitions, corresponding to independent reconfiguration actions.

Concerto provides structured semantic tools to design efficient reconfiguration plans with highly parallel, asynchronous execution. However, taking full advantage of these features adds complexity to the internal structure of components and to associated reconfiguration scripts. Automated synthesis of reconfiguration scripts is therefore particularly useful in this context.

3   Reconfiguration Script Synthesis

This section describes the synthesis process used to generate reconfiguration scripts. This process takes as input a description of the current state of the

system, namely the topology of the assembly (components and their connections) and the active places. We assume that the system is in a state where no component has pending behavior requests or ongoing transitions. Besides that information, the synthesis process also depends on a reconfiguration goal that is composed of (i) constraints Γ_{ports} on the final state of ports and (ii) a set of behaviors Γ_{bhv} to execute on designated components.

The constraints Γ_{ports} are given by a partial function that maps specific instances of component ports to a boolean indicating whether that port is required to be active or inactive. A reconfiguration satisfies that goal if it ends in a state such that for any component c and port p, if $\Gamma_{ports}(c, p)$ is defined, the port p of component c is active if and only if $\Gamma_{ports}(c, p) = \top$. Where the value of Γ_{ports} is undefined, any status of the port satisfies the constraint. This means that a reconfiguration goal does not have to specify a unique final state for components, but instead allows for multiple target states. It may appear tedious to specify constraints for all components of an assembly when a reconfiguration is specifically aimed at a subset of it, but in practice the current state of the assembly can be used to guide the choice of Γ_{ports} for those other components. A reasonable strategy might specify that provide ports active before the reconfiguration should remain active, and leave other ports unspecified.

The other element of the reconfiguration goal is the set Γ_{bhv}, where each element is a pair composed of a component and a behavior. The reconfiguration satisfies it if it executes at least all these behaviors on the corresponding components. The set Γ_{bhv} alone may not correspond to a feasible reconfiguration. For example, a system administrator wishing to update the components of Figure 1 might give a behavior goal $\Gamma_{bhv} = \{(\texttt{dep1}, \texttt{update}), (\texttt{dep2}, \texttt{update})\}$ and a port goal Γ_{ports} that maps every port instance to $\top$. The behaviors listed in that reconfiguration goal are not enough to carry it out, as it lacks a behavior to deactivate the use ports of the server prior to the update, and behaviors to reactivate all ports after the update. The synthesis process must therefore deduce necessary behaviors to carry out the reconfiguration goal, then schedule their executions in a suitable order. It proceeds as follows:

1. for each component independently, we find a sequence of behaviors that satisfies the goal, assuming that ports requirements are fulfilled (Subsection 3.1);
2. we find a global schedule for these sequences of behaviors (Subsection 3.2);
3. if the scheduling problem is found unsatisfiable, we analyze the incomplete schedule to deduce unsatisfied port requirements, compute additional reconfiguration sub-goals and iterate the process (Subsection 3.3);
4. once a feasible solution has been found, we attempt to optimize it by relaxing synchronization conditions (Subsection 3.4).

3.1   Determining Sequences of Component Behaviors

A procedure $localSeq(c, act^c, \Gamma_{bhv}, \Gamma_{ports})$ finds a sequence of behaviors that satisfies a reconfiguration goal Γ for a single component c starting in a state with active places act^c. This is achieved by enumerating all sequences of behaviors

with at most one occurrence of any behavior, and selecting one that satisfies the goal constraints. In practice, this enumeration is short because the number of behaviors of a component is usually small. More importantly, for a given component state (denoted by its active places), many behaviors do not have transitions originating from the active places. Since executing these behaviors would not have any effect, they can be ignored during the enumeration. Consequently, the number of useful sequences of behaviors to analyze is often much lower than the number of permutations. If no satisfying sequence is found by *localSeq*, then the problem has no solution, and the whole synthesis process fails. However, if multiple solutions are returned, the best possible sequence is picked, according to some (possibly user-defined) selection criterion. Some interesting optimization criteria are: the length of a sequence, its execution time (if time estimations are available for individual transitions, this may be computed with great accuracy [10]), the number of transitions it executes sequentially, or the number of ports it (de)activates. In our experiments, we used this last criterion, as it picks the component reconfiguration that is least likely to induce changes in other components, leading to simpler and potentially faster reconfiguration plans.

In order to coordinate sequences or behaviors across the assembly, we keep track of ports requirements and activity during each behavior of a sequence. In particular, for each behavior in a sequence, we record use ports of the component that are activated at least once by the behavior (they must be connected to an active provide port during the execution of the behavior), and provide ports that are deactivated at least once (they must *not* be connected to an active use port). In addition, we also record the status of each port at the end of the behavior. This information is computed with a simple traversal of the behavior graph, starting from the places that are active at the beginning of the behavior.

In the example of the update for the assembly in Figure 1, *localSeq* determines that components `dep1` and `dep2` should each execute the sequence [`update`, `deploy`]: the first behavior is included in Γ_{bhv} and the second is required to take the components to a state that satisfies Γ_{ports}.

3.2   Assembly-Level Reconfiguration Scheduling

Once sequences of behaviors to execute over each component have been determined, we turn our attention to the whole assembly and attempt to compute a sequence of reconfiguration commands (specifically, behavior requests and synchronization requests) that execute these behaviors. The challenge is to coordinate these behaviors in a way that satisfies all port requirements. To facilitate coordination and to restrict the search space, we specifically try to generate a reconfiguration composed of *steps*, such that each component executes at most one behavior per step, and each step is followed by a global synchronization request. This assumption on parallelism is reminiscent of the BSP model [4]. Figure 2b gives an example of such a reconfiguration, to be compared with Figure 2a, which achieves the same result with fewer synchronization points.

```
                                      pushB(server, suspend)
                                      waitAll()
    pushB(server, suspend)            pushB(dep1, update)
    pushB(dep1, update)               pushB(dep2, update)
    pushB(dep2, update)               waitAll()
    pushB(dep1, deploy)               pushB(dep1, deploy)
    pushB(dep2, deploy)               pushB(dep2, deploy)
    wait(dep1)                        waitAll()
    wait(dep2)                        pushB(server, deploy)
    pushB(server, deploy)             waitAll()
    wait(server)
```

(a) Target reconfiguration program.   (b) A reconfiguration with four syn-
                                      chronized steps.

Fig. 2: A reconfiguration plan to perform updates on components `dep1` and `dep2` of the assembly in Figure 1, then restore the system to a working state.

SMT Constraints To find a reconfiguration plan, ordering constraints and port requirements are encoded as a problem in a many-sorted first-order logic (i.e., the logic is equipped with *sorts* that partition the domain, similarly to a simple type system), and an SMT solver is used to obtain a solution. That encoding of the scheduling problem centers around a sort Behavior, with a finite number of elements that represent the behaviors to schedule. The main task of the SMT solver is to find an interpretation for a function schedule that maps behaviors to a reconfiguration step during which to execute them. Conceptually, schedule could range over natural numbers, with behavior b executed at the ith step if $i = $ schedule(b). However, such a model would require constraints with universal quantifiers over natural numbers, which pose a challenge for SMT solvers. It is also unnecessary, since there are only a finite number of behaviors to schedule: the number of steps required is at most the number of behaviors, when only one component executes a behavior at each step. If behaviors are executed in parallel over different components, fewer steps are required. Consequently, to improve the performance of the solver, the different steps of the reconfiguration are represented by another finite-domain sort Step, with elements $\text{step}_1, \ldots \text{step}_n, \text{step}_{\text{final}}$. The element $\text{step}_{\text{final}}$ represents the ultimate state of the system rather than a reconfiguration step. Accordingly, the scheduling function has the signature schedule : Behavior $\rightarrow$ Step, and the problem contains the constraint $\text{schedule}(b) \neq \text{step}_{\text{final}}$ for each behavior b.

A successor function succ : Step $\rightarrow$ Step is needed to describe the effect of a reconfiguration step on the subsequent state of the system. Constraints $\text{succ}(\text{step}_i) = \text{step}_{i+1}$ (for $0 \leqslant i < n$), $\text{succ}(\text{step}_n) = \text{step}_{\text{final}}$ and $\text{succ}(\text{step}_{\text{final}}) = \text{step}_{\text{final}}$ define the interpretation of succ. Likewise, to easily express sequentiality constraints, a function int : Step $\rightarrow$ Int maps each step to its step number, as defined by constraints $\text{int}(\text{step}_i) = i$. With this function, sequentiality is easily expressed: for any two consecutive behaviors b_1 and b_2 in the sequence of behaviors to schedule for a given component, the constraint $\text{int}(\text{schedule}(b_1)) < \text{int}(\text{schedule}(b_2))$ is added. This function reintroduces an infinite domain, which we sought to eliminate with the sort Step. However, since the problem contains

no quantifiers over integers, the solver only has to check that the aforementioned formula is satisfied by a speculated interpretation of schedule. This limited form of integer reasoning has a negligible impact on the search.

The main difficulty in scheduling a reconfiguration lies in ensuring that ports requirements are satisfied for each behavior of a component. A predicate $\mathsf{act_p : Step \to Bool}$ is introduced for each (use or provide) port p to indicate the activity status of the port at the beginning of reconfiguration steps. The status of each port p after each behavior b is uniquely defined, as determined during the computation of the sequences of behaviors of the component to which the port belongs. Correspondingly, a constraint $[\neg]\mathsf{act_p(succ(schedule(b)))}$ is added to reflect that status. The square brackets denote the absence or presence of the negation, depending on whether the port is inactive or active at the end of the behavior. Conversely, the status of a port cannot change if its component is not executing a behavior. For a component with behaviors $\mathsf{b_1}, \ldots, \mathsf{b_n}$, the constraint $\mathsf{schedule(b_1) \neq step_i \land \cdots \land schedule(b_n) \neq step_i} \implies (\mathsf{act_p(step_i)} \iff \mathsf{act_p(succ(step_i))})$ is added for every step i such that $0 \leqslant i < n$. Ports requirements can then be modeled. Let u be a use port that needs to be provided (i.e., connected to an active provide port) during behavior b, and p the provide port to which it is connected, the constraint $\mathsf{act_p(schedule(b))}$ ensures that p is active (and u provided) when b begins. Conversely, for a provide port p deactivated by a behavior b and connected to a use port u, $\neg\mathsf{act_u(schedule(b))}$ ensures that u is inactive when b begins. Furthermore, for any behavior b that activates a use port u and any behavior $\mathsf{b'}$ that deactivates the connected provide port p, the constraint $\mathsf{schedule(b) \neq schedule(b')}$ ensures that the behaviors are executed at different steps, hence separated by a synchronization barrier.

The problem[3] is passed to an SMT solver. If satisfiable, the interpretation found for schedule is used to build a reconfiguration script such as in Figure 2b.

Note that the scheduling problem could be encoded as a SAT problem. However, SMT solvers can reason about the theory EUF (equality and uninterpreted functions) using a dedicated congruence algorithm. We also use (non-recursive) data types, for which some SMT solvers have a dedicated reasoning algorithm [3], to represent the domains of Behavior and Step. These capabilities allow us to encode the problem straightforwardly and obtain solutions efficiently. Also note that the size of the scheduling problem is only a function of the number of behaviors to schedule and the number of component ports, but does not depend on the internal complexity of components, so that optimized components with several parallel transitions will not adversely affect the synthesis method.

3.3   Determining Missing Behaviors

Until now, we have considered the scheduling problem under the assumption of a fixed sequence of behaviors to schedule for each component. In general, a set of behaviors may have no feasible schedule. For example, it is not possible to

---

[3]Illustrating instances for the running example, in the SMT-LIB file format, can be found at https://doi.org/10.5281/zenodo.5820571.

fully execute the behavior `update` on components `dep1` and `dep2` of the assembly in Figure 1 without first deactivating the use ports `service1` and `service2` of component `server`, i.e., executing its behavior `suspend`. To plan reconfigurations for an incomplete set of behaviors, we use our SMT encoding of the scheduling problem to detect the point in the reconfiguration at which additional changes must be performed, then we create new component reconfiguration sub-problems and use the solutions to augment the sequences of behaviors to schedule.

Let S be a mapping that associates to each component a sequence of behaviors (i.e., the sequence to be executed by that component, as determined in Subsection 3.1), a *maximal executable schedule* S' of S is a mapping that associates to each component c a prefix of $S(c)$, such that (i) the scheduling problem corresponding to the sequences in S' has a solution (ii) no reconfiguration problem built by extending a prefix in S' with one behavior has a solution. Intuitively, a maximal executable schedule is a point up to which the reconfiguration S can be carried out, before unsatisfied port requirements prevent further execution.

Procedure 1 iteratively computes a maximal executable schedule S' and uses the resulting information to refine the sequences of behaviors to execute for each component, until a solution is found that executes them all. By analyzing the statuses of ports in the assembly at the end of the execution of S' (which depend only on the last behavior in each sequence), and comparing them to the requirements of the first unscheduled behaviors in S, we deduce a set of provide ports to activate and use ports to deactivate to allow further scheduling of S, and compute intermediary ports constraints Γ'_{ports}. For each component c that does not have unscheduled behaviors in S, we determine a sequence s_1 of behaviors that satisfies this intermediate goal (assuming that the component starts with active places $act_{S'}^c$ corresponding to its state after executing the last behavior in $S'(c)$) and a sequence s_2 that takes the component from its state after executing s_1 (active places $act_{s_1}^c$) to one that satisfies the port constraints Γ_{ports} of the original goal. Sequences of behaviors to execute are thus extended ([] denotes the empty sequence, and $s_1 \cdot s_2$ the concatenation of two sequences). To ensure a monotonic search, sequences are extended only for components c without unscheduled behaviors in S, i.e., not the components that brought about the intermediary goal Γ'_{ports}. If no such extension can be found ($\neg progress$), the scheduling of S is blocked by a circular dependency between components and the synthesis process fails. If the procedure terminates, it returns a reconfiguration script corresponding to a solution of the scheduling problem of S.

Consider the example of running updates in the assembly of Figure 1. Initially (see Subsection 3.1), the mapping S of sequences of behaviors computed with *localSeq* is defined by $S(\texttt{dep1}) = S(\texttt{dep2}) = [\texttt{update}, \texttt{deploy}]$, and $S(\texttt{server}) = []$, because Γ_{bhv} does not include any behavior for that component, and the component is already in a state that satisfies Γ_{ports}. This combination of sequences of behaviors has no feasible schedule. In particular, the mapping S' that associates to every component the empty sequence is found to be a maximal executable schedule of S. The first unscheduled behaviors in S are two instances of `update`, they require use ports `service1` and `service2` of component `server` to be deac-

Procedure *globalSolution*$(A, \Gamma_{bhv}, \Gamma_{ports})$ **is**
    for $c \in A$ **do** $S(c) \leftarrow localSeq(c, act^c_A, \Gamma_{bhv}, \Gamma_{ports})$;
    while $findMaxExecSchedule(S) \neq S$ **do**
        $S' \leftarrow findMaxExecSchedule(S)$;
        $\Gamma'_{ports} \leftarrow$ port conditions required to execute, for every component c,
          the first behavior in $S(c)$ that is not in $S'(c)$;
        $progress \leftarrow false$;
        for $c \in A$ **such that** $S'(c) = S(c)$ **do**
            $s_1 \leftarrow localSeq(c, act^c_{S'}, \Gamma_{bhv} \setminus S'(c), \Gamma'_{ports})$;
            $s_2 \leftarrow localSeq(c, act^c_{s_1}, \emptyset, \Gamma_{ports})$;
            if $s_1 \neq [\,]$ **or** $s_2 \neq [\,]$ **then**
                $S(c) \leftarrow S'(c) \cdot s_1 \cdot s_2$;
                $progress \leftarrow true$;
            end
        end
        if $\neg\, progress$ **then** fail ;
    end
    return *reconfigurationScriptOfSolution*(S) ;
end

Procedure 1: Synthesizes a reconfiguration script.

tivated. Consequently, two new reconfiguration sub-goals are created for `server`. The first requires it to reach a state where the two ports are deactivated, a call to *localSeq* returns the solution $s_1 = [\texttt{suspend}]$. From the resulting component state, the second reconfiguration sub-goal requires `server` to go to a state that satisfies Γ_{ports}, in this case *localSeq* returns the sequence $s_2 = [\texttt{deploy}]$. S is updated so that $S(\texttt{server}) = [\texttt{suspend}, \texttt{deploy}]$. At this point, S is found to be a maximal executable schedule of itself, and the corresponding solution is returned, i.e., the reconfiguration plan in Figure 2b. Note that Procedure 1 is not guaranteed to terminate, nor is it a complete search algorithm. In particular, it relies on two heuristics: the selection function used when *localSeq* finds multiple candidate sequences, and the choice of maximal executable schedule for a given mapping S.

Computing a Maximal Executable Schedule Procedure 1 relies on a function *findMaxExecSchedule* to compute a maximal executable schedule of a mapping S, illustrated in Procedure 2, that maintains a mapping containing prefixes of elements in S (initially mapping every component to the empty sequence) and incrementally extends those prefixes, checking every time the satisfiability of the corresponding scheduling problem. This procedure calls the SMT solver to check the satisfiability of the scheduling problems. In the actual implementation, some simple checks are also used to quickly detect some trivially unsatisfiable or satisfiable instances of the scheduling problem, although these are left out of Procedure 2 for clarity. The procedure continues until all behaviors have been included or no additional behavior can be scheduled. A maximal executable

schedule always exists (the mapping that associates every component to the empty sequence always has a satisfiable scheduling problem, and may be maximal), and *findMaxExecSchedule* always finds one. However maximal executable schedules are not unique, and a bad choice may result in an ineffective reconfiguration plan. In the example above, during the second iteration, the mapping S of sequences to schedule is defined by $S(\texttt{server}) = [\texttt{suspend}, \texttt{deploy}]$ and $S(\texttt{dep1}) = S(\texttt{dep2}) = [\texttt{update}, \texttt{deploy}]$. S itself is a maximal executable schedule of S, but so is the mapping S' defined by $S'(\texttt{server}) = [\texttt{suspend}, \texttt{deploy}]$ and $S'(\texttt{dep1}) = S'(\texttt{dep2}) = [\,]$. S' corresponds to the case where the server is restarted too early. Picking this maximal executable schedule will ultimately lead to a reconfiguration that stops the server at least twice. To avoid this, a good heuristic for *findMaxExecSchedule* is to extend in priority the prefixes for which the added behavior is least likely to affect other components, i.e., those that deactivate the fewest provide ports and activate the fewest use ports.

Procedure *findMaxExecSchedule(S)* **is**
    suffixes ← S ;
    for c **such that** *suffixes(c)* is defined **do** *prefixes(c)* ← [ ];
    progress ← *true* ;
    while *progress* **do**
        progress ← *false* ;
        for c **such that** *suffixes(c)* ≠ [ ] **do**
            b ← *head(suffixes(c))* ;
            if the scheduling problem for *prefixes* extended with b is satisfiable
            then
                progress ← *true* ;
                prefixes(c) ← *prefixes(c)* · $[b]$;
                suffixes(c) ← *tail(suffixes(c))* ;
            end
        end
    end
    return *prefixes* ;
end

Procedure 2: Computes a maximum executable schedule for sequences of behaviors S.

3.4   Relaxation of Synchronization Barriers

The assumption that reconfigurations should proceed in globally synchronized steps, although useful to find a solution, severely limits the potential for inter-component parallelism, a key feature of Concerto. A final optimization stage takes the reconfiguration plan with synchronized steps and relaxes synchronization where possible. First, every command `waitAll()` is replaced with a sequence

of commands `wait(c)` for every component c that executes a behavior in the preceding step. This preserves the semantics of the reconfiguration and makes the targets of synchronization explicit. Then, for a given step i and a given command `wait(c)` after this step, we apply the following rule: if for all behaviors executed by c since the last command `wait(c)` up to step i, no provide (resp., use) port is deactivated (resp., activated) and connected to a use (resp., provide) port that is activated (resp., deactivated) at step $i+1$, then `wait(c)` can be delayed until after step $i + 1$. This rule is applied for every step in order, delaying barriers as late as possible and removing duplicates. This transformation reduces the number of barriers yet ensures that behaviors with conflicting effects on ports remain separated by an explicit synchronization. Port requirements for behaviors do not have to be taken into account, as the Concerto execution model ensures implicit synchronization for those. As an example, this optimization applied to the reconfiguration plan in Figure 2b yields the one in Figure 2a.

4   Experiments

The implementation described here, the examples, and the experimental results are available at https://doi.org/10.5281/zenodo.5820571.

4.1   Implementation

We implemented the synthesis process in a Python tool that attempts to produce a reconfiguration script for a given assembly and reconfiguration goal. The process is entirely automated. Given a description of an assembly and a reconfiguration goal, it generates relevant scheduling problems and interacts with an SMT solver to generate reconfiguration programs. Intermediate scheduling problems can be output in the SMT-LIB file format, the standard used by most SMT solvers [2], and can be solved using any solver that complies with version 2.6 of the SMT-LIB standard. The preferred mode of operation for our tool does not output files, but interacts with the SMT solver Z3 [23] through the Z3 Python API. This interface makes it easy to analyze interpretations returned by the solver for satisfiable problems, and thus to reconstruct schedules. This is the mode of operation used to conduct the experiments described below.

4.2   Results Over Synthetic Examples

To test our technique on a variety of cases, we devised assemblies with four types of topology. In *central-user* assemblies, a set of provider components, each with a pair of provide ports, is connected to different use ports of one central user component. In *central-provider* assemblies, one central provider component has a pair of provide ports that is connected to (a pair of use ports of) multiple other components. In *linear* assemblies, components form a chain such that each component has a pair of provide ports connected to the pair of use ports of the next component. In *stratified* architectures, components are organized in

levels containing up to three components, such that each component in a level has a pair of provide ports connected to use ports on every component in the level above (i.e., a provide port can be connected to up to three use ports). Every component in these assemblies is equipped with behaviors to deploy it, update or suspend it, and uninstall it. Figure 3 depicts those four topologies, with internal nets of components omitted for clarity. As an example of the internal structure of components, Figure 1 shows the central-user assembly with three components. For other types and sizes of assembly, components follow similar internal structures, adapted to offer adequately many ports.

For each architecture, we generated assemblies with 10, 30 and 100 components (scaling the number of providers for *central-user*, the number of users for *central-provider*, the length of the chain for *linear* or the number of levels for *stratified*), and ran three scenarios. The *deployment* scenario starts with all components uninstalled, Γ_{ports} requires the activation of a provide port on the last component(s) in the dependency order, while Γ_{bhv} is empty. The *update* scenario starts with all components running, Γ_{ports} requires a similar final state, and Γ_{bhv} includes update behaviors for components that are first in the dependency order and no behavior for the others. The *uninstall* scenario starts with all components running, Γ_{ports} requires the deactivation of all ports, while Γ_{bhv} is empty. Each scenario affects every component of the assembly.

(a) Central-user (b) Central-provider   (c) Linear   (d) Stratified
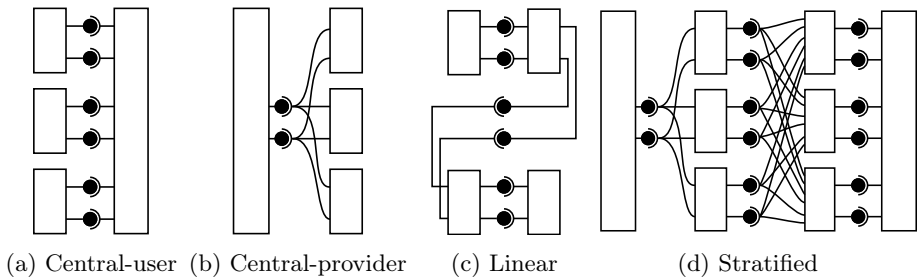
Fig. 3: The four assembly topologies in synthetic examples.

Table 1 describes the solving process and resulting solution for these 36 examples. Experiments were executed on a computer with an 8-core 1.6GHz processor and 16 GiB of RAM. Solutions were successfully generated for all but 4 examples (the process was aborted after one hour). For 21 of them, the process took less than a minute. Results indicate that the solving time, and ultimately the success of the method, depend on the topology of the assembly: the assemblies for which some reconfigurations could not be computed within one hour are those with long chains of dependencies (linear and stratified assemblies with 100 components). This can be explained in two ways: firstly, the propagation of port requirements and the deduction of missing behaviors requires a number of iterations of the main loop of Procedure 1 proportional to the length of the longest chain of dependency. Secondly, architectures with long chains of dependencies

are less conducive to parallel execution of behaviors, and therefore the instances of scheduling problems solved have a high number of steps, leading to a large search space and long solving times. For example, the deployment of 100 component in the linear architecture ultimately requires 100 steps. For each of the 17 instances of the scheduling problem solved to compute that reconfiguration, the SMT solver took on average 147 seconds to return a solution. In contrast, to deploy 100 components in the central-user architecture, the reconfiguration script requires only 2 steps, as a result the SMT solver was able to return a solution after only 0.21 seconds on average each time it was called. For difficult problems, the solving time is dominated by calls to the SMT solver as shown in the solving time column of Table 1 (in parentheses, the cumulated time taken by the SMT solver). Overall, these examples show that our method is able to plan reconfigurations affecting large number of components. Furthermore, architectures with a very large number of components, such as microservice architectures, typically have a shallow depth rather than long chains of dependencies, and scale horizontally [21, 24, 25], similarly to our central-user and central-provider architectures. Our method scales well in those conditions.

Writing a correctly coordinated reconfiguration plan with tens of asynchronous behaviors is a non-trivial task. It is particularly difficult when explicit synchronizations commands are needed in the reconfiguration script. The execution model of Concerto ensures that this is seldom necessary, but some synchronization barriers are required, e.g., in the update scenarios to prevent early restarts that would block the updates. Our synthesis technique determines synchronization points required for completion of the reconfigurations, but it also avoids synchronization points that would slow the execution unnecessarily. It performs these tasks quickly, with a time gain that is especially significant when compared to the service interruption that an incorrect reconfiguration would cause.

4.3   OpenStack Use Case

We also tested our method on a real OpenStack system. OpenStack is the de facto standard open-source solution to address the IaaS level of the cloud paradigm, it can be seen as the open-source operating system of the cloud.

In previous work [8], Madeus, a subset of Concerto restricted to deployment, was used to deploy an OpenStack system. Following the deployment strategy of the reference production deployment tool Kolla, 11 components were specified, resulting in a real OpenStack deployment up to 70% faster than Kolla. Here we use the same components, extended with behaviors for reconfiguration. We analyzed the official installing, updating and uninstalling procedures of OpenStack to design the associated internal nets. Figure 4 depicts those components and their connections, with details of the internal structure depicted for the four main components, and omitted for clarity on seven others. The reconfiguration starts with all components running. The reconfiguration scenario requires an update of the database component ($\Gamma_{bhv} = \{(\texttt{mariadb}, \texttt{update}), (\texttt{mariadb}, \texttt{deploy})\}$) and Γ_{ports} specifies that all ports must eventually return to their initial (active) state. Our method generates a reconfiguration plan in 1.95 seconds, correctly deducing

	assembly		solving		plan	
	arch.	size	smt	time (s)	steps	bhvs
deployment	c-user	10	2 (2)	0.25 (0.02)	2	10
		30	5 (5)	1.99 (0.18)	2	30
		100	17 (17)	23.94 (3.59)	2	100
	c-provider	10	2 (2)	0.33 (0.08)	2	10
		30	5 (5)	3.95 (1.80)	2	30
		100	17 (17)	93.07 (68.67)	2	100
	linear	10	2 (2)	0.40 (0.08)	10	10
		30	5 (5)	9.19 (4.27)	30	30
		100	17 (17)	2689.86 (2512.22)	100	100
	stratified	10	3 (3)	0.64 (0.09)	5	10
		30	6 (6)	12.04 (5.28)	12	30
		100	18 (18)	1274.52 (1121.40)	35	100
update	c-user	10	6 (5)	1.69 (0.78)	4	20
		30	12 (11)	25.07 (18.59)	4	60
		100	36 (35)	1737.29 (1654.67)	4	200
	c-provider	10	13 (4)	1.79 (0.41)	4	20
		30	40 (11)	22.98 (9.97)	4	60
		100	133 (34)	685.54 (541.69)	4	200
	linear	10	50 (5)	12.72 (3.26)	20	20
		30	446 (11)	1388.50 (825.06)	60	60
		100	–	–	–	–
	stratified	10	20 (6)	14.55 (5.82)	13	26
		30	147 (16)	2306.85 (1885.17)	34	86
		100	–	–	–	–
interruption	c-user	10	3 (3)	0.54 (0.14)	3	11
		30	6 (6)	6.01 (2.76)	3	31
		100	18 (18)	162.51 (125.51)	3	101
	c-provider	10	4 (4)	0.91 (0.30)	3	19
		30	10 (10)	12.36 (7.25)	3	59
		100	34 (34)	571.73 (514.48)	3	199
	linear	10	10 (10)	1.30 (0.19)	10	10
		30	30 (30)	39.31 (13.69)	30	30
		100	–	–	–	–
	stratified	10	4 (4)	2.50 (1.10)	9	19
		30	11 (11)	132.59 (108.53)	23	59
		100	–	–	–	–

Table 1: Results of the synthesis process on synthetic examples. For each problem, the table indicates the architecture of the assembly and (arch.) its number of components (size), the number of problems solved by the SMT solver (smt) followed in parentheses by the number of those that were found satisfiable, the total solving time in seconds followed in parentheses by the cumulated time taken by the SMT solver (time), the number of steps in the solution before relaxation of the synchronization barriers (steps), and the number of behaviors executed in that solution (bhvs).
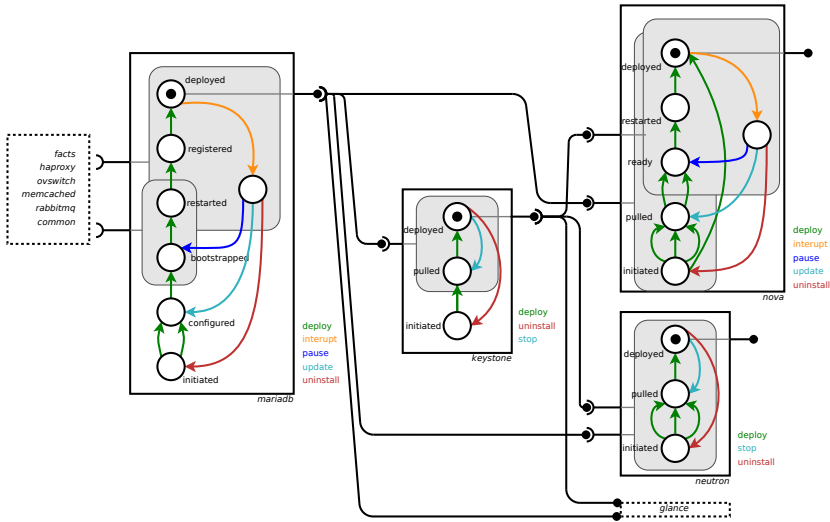
Fig. 4: A Concerto assembly for an OpenStack system.

missing behaviors for `mariadb` as well as components affected by the interruption of its service, i.e., `keystone`, `nova`, `neutron`, and `glance`. The generated plan coordinates 12 behaviors on these 5 components. After optimization, it includes only 2 synchronization points needed to ensure the complete re-deployment of `mariadb` and `keystone`, whose services are required by other components.

While the scale of this use case may seem limited, its architecture is not trivial. This real-world scenario leads to a complicated synchronization problem. The 12 behaviors in the reconfiguration program require 8 global synchronization steps before optimization. The optimization phase reduces this to 2 individual synchronization points, thus enhancing the level of parallelism and asynchrony of the reconfiguration program, while preserving its correctness. A DevOps engineer or system administrator would be challenged to write such a program without errors or unnecessary synchronization points, whereas our solution only requires them to specify a reconfiguration goal.

5   Related work

For models with fixed component life cycles, planning and scheduling techniques have been used to plan reconfigurations [1, 13]. Pre-established protocols can also be used: while such solutions are in general less flexible, they have desirable features such as decentralized coordination [14] or recovery policies [6]. Comparatively fewer works study the problem of reconfiguration planning in models with programmable component life cycles, such as Concerto. Kikuchi et al. [20] synthesize reconfiguration plans with a model finder. Unlike us, they assume that all available reconfiguration operations are given in the input of the scheduling problem, which may limit scalability. Operations and reconfiguration goal are

encoded in the Alloy specification language, and synthesis is performed by the Alloy Analyzer. This work relies on a simple ad hoc component-based model, with reconfiguration operations that must be sequentially ordered. The model does not have specific execution semantics, instead the list of operations has to be given by the user, with their effects described as constraints on the states before and after the operations. Therefore the correctness of the correspondence between the synthesized procedure and its executable counterpart depends on the user. Metis [22] closes that gap between planning and execution, as it schedules deployment plans for distributed systems in the Aeolus model [12], which has formal execution semantics. The authors first describe the problem as a generic planning problem and use standard planners to solve it, then present a specialized solving algorithm. Metis is limited to deployment rather than general reconfiguration, making the computation of dependencies more straightforward. Aelous shares many similarities with Concerto, but lacks intra-component parallelism and asynchronous commands in its reconfiguration language. These features improve the efficiency of reconfigurations but also make them more difficult to plan. Note that these features can also be represented through planning and scheduling problems [18], typically solved by approximation.

The problem of determining reconfiguration goals (i.e., the analysis phase) is complementary to the planning problem. Engage [15] uses a SAT solver to build a complete target configuration (a set of components to deploy) from a partial specification, based on a hierarchical specification of a distributed software stack. It also performs limited planning, namely sequentially ordering deployments. Engage does not account for the state of the system, and is thus limited to initial deployments or reconfigurations from the ground up. Zephyrus [11] and ConfSolve [17] are two tools to infer, from the state of the system/environment, a target configuration that could be used as an entry of our planning tool.

6   Conclusion

We have described a synthesis method for reconfiguration plans of component-based systems, that relies on (i) finding local solutions at the component level, (ii) finding a schedule that coordinates those solutions at the assembly level, with the help of an SMT solver, (iii) determining unsatisfied dependencies to refine the reconfiguration goal until it becomes satisfiable, and (iv) optimizing the synthesized reconfiguration plan to improve its level of parallel and asynchronous execution. Dividing the problem in this manner, as opposed to attempting to solve it at once with an SMT solver, is a key to solving large instances, although it leads to incompleteness (the third step relies on an incomplete search guided by some heuristic choices). This design decision does not appear to affect the success of the method or the quality of synthesized plans, and allows the technique to scale to applications with large number of components, as demonstrated in our experiments on synthetic examples and a real use case. To improve scalability on complex architectures, this technique could be adapted to a hierarchical composition model, which would lend itself to a recursive resolution algorithm.

References

1. Arshad, N., Heimbigner, D., Wolf, A.L.: Deployment and dynamic reconfiguration planning for distributed software systems. In: Proceedings. 15th IEEE International Conference on Tools with Artificial Intelligence. pp. 39–46. IEEE (2003)
2. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB). www.SMT-LIB.org (2016)
3. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. Journal on Satisfiability, Boolean Modeling and Computation **3**, 21–46 (2007)
4. Bisseling, R.H.: Parallel Scientific Computation: A Structured Approach Using BSP. Oxford Unviersity Press (2020)
5. Boyer, F., Gruber, O., Pous, D.: Robust reconfigurations of component assemblies. In: 35th International Conference on Software Engineering (ICSE). pp. 13–22 (2013)
6. Boyer, F., Gruber, O., Pous, D.: Robust reconfigurations of component assemblies. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 13–22. IEEE (2013)
7. Cansado, A., Canal, C., Salaün, G., Cubo, J.: A formal framework for structural reconfiguration of components under behavioural adaptation. Electronic Notes in Theoretical Computer Science **263**, 95–110 (2010)
8. Chardet, M., Coullon, H., Pérez, C., Pertin, D., Servantie, C., Robillard, S.: Enhancing separation of concerns, parallelism, and formalism in distributed software deployment with Madeus (2020), https://hal.inria.fr/hal-02737859, preprint
9. Chardet, M., Coullon, H., Robillard, S.: Toward safe and efficient reconfiguration with Concerto. Science of Computer Programming **203** (2021)
10. Chardet, M., Hélène, C., Perez, C.: Predictable efficiency for reconfiguration of service-oriented systems with Concerto. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). pp. 340–349 (2020)
11. Di Cosmo, R., Lienhardt, M., Treinen, R., Zacchiroli, S., Zwolakowski, J., Eiche, A., Agahi, A.: Automated synthesis and deployment of cloud applications. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. p. 211–222 (2014)
12. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: A component model for the cloud. Information and Computation pp. 100–121 (2014)
13. El Maghraoui, K., Meghranjani, A., Eilam, T., Kalantar, M., Konstantinou, A.V.: Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing. pp. 404–423. Springer (2006)
14. Etchevers, X., Coupaye, T., Boyer, F., De Palma, N.: Self-configuration of distributed applications in the cloud. In: 2011 IEEE 4th International Conference on Cloud Computing. pp. 668–675. IEEE (2011)
15. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: A deployment management system. In: ACM SIGPLAN PLDI. pp. 263–274 (2012)
16. Gunawi, H.S., Hao, M., Suminto, R.O., Laksono, A., Satria, A.D., Adityatama, J., Eliazar, K.J.: Why does the cloud stop computing? lessons from hundreds of service outages. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. pp. 1–16 (2016)

17. Hewson, J.A., Anderson, P., Gordon, A.: A declarative approach to automated configuration. In: lisa'12: Proceedings of the 26th international conference on Large Installation System Administration: strategies, tools, and techniques. pp. 51–66 (2012)
18. Keller, A., Hellerstein, J.L., Wolf, J.L., Wu, K.L., Krishnan, V.: The CHAMPS system: Change management with planning and scheduling. In: 2004 IEEE/IFIP Network Operations and Management Symposium (IEEE Cat. No. 04CH37507). vol. 1, pp. 395–408. IEEE (2004)
19. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer (2003)
20. Kikuchi, S., Tsuchiya, S., Hiraishi, K.: Synthesis of configuration change procedure using model finder. IEICE TRANSACTIONS on Information and Systems **96**(8), 1696–1706 (2013)
21. Kratzke, N., Quint, P.C.: Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. Journal of Systems and Software **126**, 1–16 (2017). https://doi.org/https://doi.org/10.1016/j.jss.2017.01.001
22. Lascu, T.A., Mauro, J., Zavattaro, G.: A planning tool supporting the deployment of cloud applications. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence. pp. 213–220 (2013)
23. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 4963, pp. 337–340. Springer (2008)
24. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M.: Microservice Architecture. O'Reilly Media, Inc. (2016)
25. Taibi, D., Lenarduzzi, V., Pahl, C.: Architectural patterns for microservices: A systematic mapping study. In: CLOSER (2018)

Appendix A   Algorithm to find a sequence of behaviors for a single component

---

Procedure $enum(seq, act, bhvs, \Gamma'_{bhv}, \Gamma'_{ports})$ **is**
    if $\Gamma'_{bhv} = \emptyset$ **and** act satisfies Γ'_{ports} **then return** $\{seq\}$;
    $R \leftarrow \emptyset$;
    for $b \in bhvs$ **such that** b has outgoing transitions from act **do**
        $act' \leftarrow$ places active after execution of b starting from act ;
        $R \leftarrow R \cup enum(seq \cdot [b], act', bhvs \setminus \{b\}, \Gamma'_{bhv} \setminus \{b\}, \Gamma'_{ports})$;
    end
    return R ;
end
Procedure $localSeq(c, act^c, \Gamma_{bhv}, \Gamma_{ports})$ **is**
    $L \leftarrow enum([\,], act^c, bhvs^c, \Gamma_{bhv}{}^c, \Gamma_{ports}{}^c)$;
    if $L = \emptyset$ **then**
        fail ;
    else
        return $bestSolution(L)$;
    end
end

Procedure 3: Finds a sequence of behaviors that satisfies the reconfiguration goal for a component c.

Procedure 3 shows how a sequence of behavior that satisfies a goal for a single component c is found: *enum* recursively enumerates sequences of behaviors, it returns the set of sequences that include requested behaviors and ensure that the component finishes in a state that satisfies the port constraints. The argument *seq* denotes the sequence being analyzed, *act* denotes the active places after the execution of that sequence, *bhvs* is the set of candidate behaviors whose permutations remain to be tested, Γ'_{bhv} denotes the behaviors that remain to be included in the sequence to satisfy the goal, and Γ'_{ports} the port conditions to satisfy. *bestSolution* return the best sequence in a set, according to some (possibly user-defined) optimization criterion. Solutions returned by *localSeq* are valid under the assumption that all ports requirements are eventually satisfied by the other components connected to those ports, hence Procedure 1 may call *localSeq* multiple times with refined reconfiguration goals for a component c.

Appendix B   Internal structure of components used in synthetic benchmarks

Figure 5 describes the three types of components used in the synthetic benchmarks of Section 4.2. Provider components are used in all four architectures, user

components are used in the *central-provider* and *linear* architectures, and parallel user components are used in *central-user* and *stratified* architectures. Parallel user components can be scaled to offer p pairs of use ports, to accommodate the architecture. The internal net is adapted accordingly with p places `suspended` and `sconf`. Recall that the internal structure of components does not affect the complexity of scheduling problems, and that the time required for synthesis is dominated by the solving of these problems.

For some instances of user and parallel user components in the assemblies, the provide ports are not connected to other components (and not shown in Figure 3). All other component ports are connected in the assemblies.

(a) Provider component   (b) User component
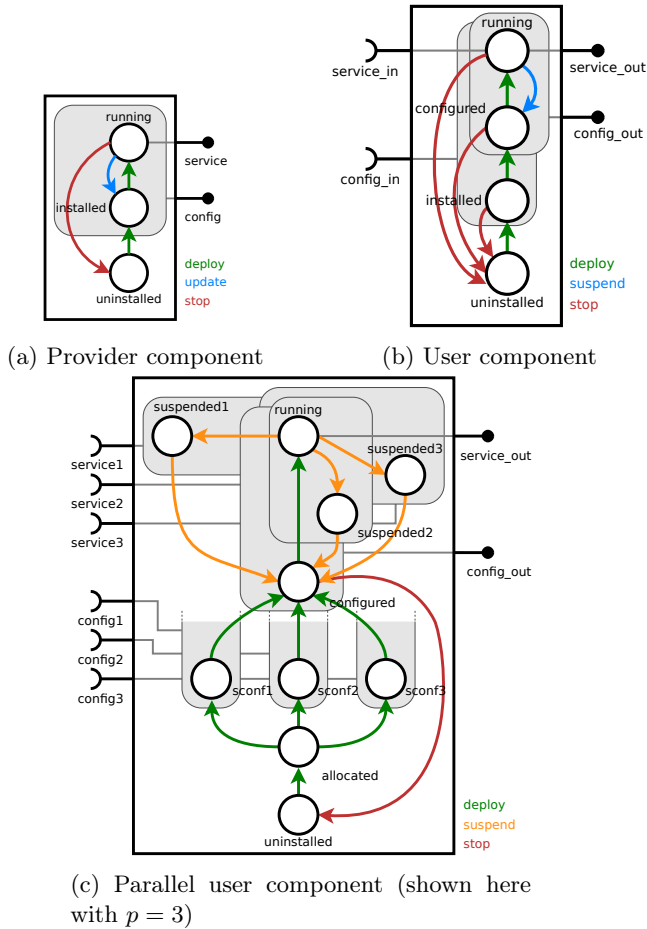
(c) Parallel user component (shown here with $p = 3$)

Fig. 5: The components used in synthetic benchmarks.