Vol.2016-IOT-32 No.25
2016/3/4

配備エージェントを組換え可能なテンプレート型プロビジョニング手法の提案

黒田 貴之[1,a)]　中野谷 学[1]　北野 敦資[1]　登内 敏夫[1]

概要：所望のシステム構成の定義を基にシステムを一括して自動構築するテンプレート型プロビジョニングにおいては，構築対象システム内の各構成要素の配備は要素ごとの配備エージェントにより実行される．従来手法では，利用される配備エージェントは構成要素ごとの固定的な前提を基に決められており，これがシステム構成を定義する上で柔軟性の阻害要因となっていた．そこで本研究では，配備エージェントについても構築対象と同様に柔軟に組換え可能とする手法を提案し，より効率的なシステム自動構築を実現する．本稿では，提案手法におけるシステムの記述方法と配備処理や配備エージェントの実装方法，および構築処理の実行系の設計について述べ，サンプルシステムに基づく検証を通じて本手法の有効性を評価する．

キーワード：テンプレート型プロビジョニング，自動配備，自動構築，構成指向計画

Adding Exchangeability of Deployment Agent to Template-based Provisioning

Takayuki Kuroda[1,a)]　Manabu Nakanoya[1]　Atsushi Kitano[1]　Toshio Tonouchi[1]

Abstract: Template-based Provisioning is a promising approach to provision a system automatically with a definition of a desired system configuration. In the current scheme, a deployment agent for each element of the system is decided statically with a definition of the element, but it causes to reduce flexibility and reusability of the system definition. Therefore, we propose a novel scheme which enhances efficiency of defining a system to provision with adding exchangability of the deployment agents. In this paper, we firstly present the new specification for system definitions, deployment tasks and deployment agents. Secondly, we show design of our execution engine. Finally, we validate usability of our scheme with showing case studies.

Keywords: Template-based provisioning, deployment, provisioning, Configuration-Oriented Planning

1. はじめに

クラウドサービスの普及に伴い，企業や通信キャリア，データセンタ事業者などにおいて，ICT システムの効率的な構築手法が広く求められつつある．従来は構築スクリプトによる構築作業の自動化が一般的であったが，システムの規模や構成の多様性が増加する中で個別の構築スクリプトの作成は困難となりつつあり，近年ではより効率的な構築自動化手法に注目が集まっている [2], [4], [5], [6].

設定管理ツール，或いは Configuration Management Tool(以下，CMT と略記) と呼ばれるツール群は，冪等性と呼ばれる性質を備える点に特徴がある．設定管理における冪等性 [*1] は，所望の要求状態を指定するだけで管理対象物の状態を制御可能であるという性質を意味する．冪等性を備えるツールは管理対象物の現在の状態に応じて実行すべき処理を自動的に判断するため，要求の記述は大幅に簡素化できる．CMT の具体例としては，Puppet，Chef，Ansible，PowerShell DSC などが知られている．また，テ

[1] NEC Corporation
Kawasaki, Kanagawa 211–8666, Japan
[a)] t-kuroda@ax.jp.nec.com

[*1] 本来は，冪等性とはある操作が適用回数に依らず同じ結果に収束する性質を指す数学上の概念である．

ンプレート型プロビジョニング，或いは Template-Based Provisioning(以下，TBP と略記) は，比較的大規模なシステムを対象とした自動配備技術である．これは，予め定義された構築スクリプトのテンプレートに適宜値を挿入することで，多様なシステム向けの構築スクリプトを迅速に作成可能とする技術である．TBP の具体例としては，OSS の IaaS 基盤である OpenStack のプラグインとして開発されている Heat や，国際標準化団体 OASIS による TOSCA[1], [3] などが知られている．

これらの構築自動化手法では，事前に定義される部品の再利用性が低く，実際の利用においては多くの定義作業が必要となる点に課題がある．そこで筆者らは，CMT や TBP の利点を活かしつつ，より簡潔に所望のシステム構成の定義を可能とし構築作業を効率化するための研究を推進してきた [7], [8], [9]．構成指向計画，或いは Configuration-Oriented Planning(以下，COP と略記)[8] は，事前に用意された定義部品の組み合わせとプロパティ値の設定のみにより所望のシステム構成を定義し，且つその構築手順を自動生成する技術である．しかしながら既存の COP においては，構築対象システム内で配備を実行する配備エージェントに関しては部品化の仕組みが整備されておらず，例えば OS 上でのミドルウェアの配備には ShellScript しか利用できなかったり，VM(仮想マシン) の配備に特定の IaaS しか利用できなかったりするなど，システム構成定義の柔軟性を阻害する要因となっていた．

そこで本研究では，配備エージェントについても組換え可能に部品化し構成を柔軟に定義可能とすることにより，効率的なシステム構築自動化手法を実現する．本稿では，部品化された配備エージェントを含むシステムの定義方法に加え，構築処理の実行系についても併せて詳細を示す．

以下，本稿では，2 章において関連技術とその課題について述べ，3 章で提案手法の要旨を説明し，4 章で提案手法を実現するプロトタイプの設計について述べる．続いて 5 章では，サンプルシステムに基づく検証を通じて本手法の有効性を評価し，6 章で結論をまとめる．

2. 関連技術

2.1 TOSCA

TOSCA[3] とは，国際標準化団体 OASIS が策定を進めているシステムの記述仕様である．本来はクラウド基盤上で稼働するアプリケーションを多様なクラウド環境で動作可能に記述するための仕様であるが，近年では OpenStack のプラグインである Heat や通信キャリアにおけるネットワーク仮想化技術である NFV 仕様との連携が模索されるなど，様々な形態のシステムにおける活用が期待されている．TOSCA における記述内容は，構成に関する記述と配備処理に関する記述に大別される．前者は Node と呼ばれる部品の組み合わせによって記述され，後者は Plan と呼ばれるワークフロー定義によって記述される．

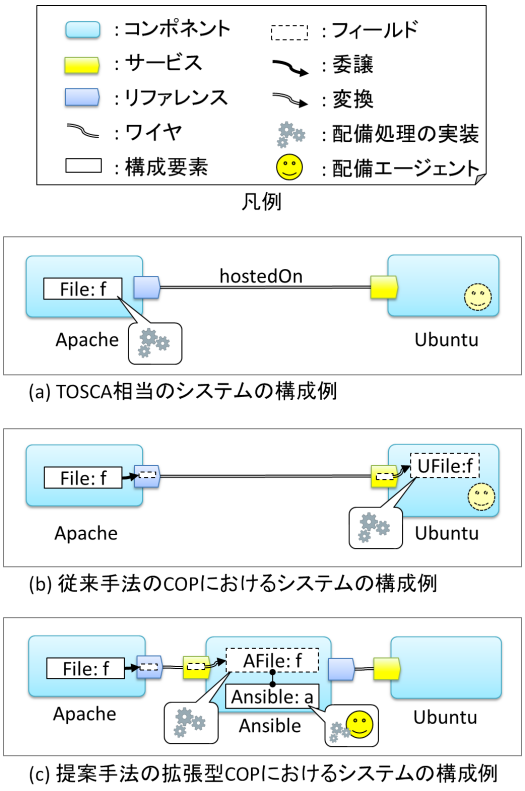

図 1 配備処理実装と配備エージェントの実現方式の比較
Fig. 1 Comparison of specification to define deployment tasks and deployment agents.

ここで，図 1 に配備処理の実装と配備エージェントの実現方式の比較について示す．(a)〜(c) の各図は 3 つの方式におけるシステムの記述例を概念的に示している (但し，凡例に挙げた各種要素の名称は本稿独自の名称である点に注意されたい)．いずれの方式においてもシステムはコンポネントと呼ばれる部品の組み合わせで表現される．各コンポネントには，まず当該コンポネントの構成要素が定義される．加えてサービスとリファレンスの 2 種類のポートが定義可能であり，これにより当該コンポネントが提供可能な機能と必要とする機能をそれぞれ表現できる．ワイヤによりこれらのポート間の連結関係を定義して，コンポネントが要求する機能を他のコンポネントによって充足させることで，配備可能な一連のシステム構成が定義できる．図 1 の例はいずれも Apache HTTP Server が Ubuntu Linux 上にホストされた構成である．ここでは簡単のため構成要素としては主に *Apache* 上の 1 つの *File* 要素のみを示す．

図 1(a) は TOSCA 相当の方式におけるシステムの記述

*2 Puppet, https://puppetlabs.com/
Chef, https://www.chef.io/chef/
Ansible, http://www.ansible.com/
PowerShell DSC, https://msdn.microsoft.com/en-us/powershell/dsc/overview
OpenStack, https://www.openstack.org/
Heat, https://wiki.openstack.org/wiki/Heat
(確認日 2016/01/27.)

例である．本方式において，*File* 要素の配備処理の実装は *Apache* コンポネント上に定義される．具体的には，配備処理は例えば“ create.sh ”といった ShellScript で記述でき，コンポネント上にはスクリプトファイルへのパスが記述される．また，このスクリプトは *hostedOn* とマークされたワイヤで関連付けられた *Ubuntu* 上で実行されることが暗に期待される．ここで仮に *Apache* コンポネントの開発者が *File* 要素の配備処理に Puppet を利用したい場合には，予め Ubuntu 上に Puppet の実行環境がインストールされている必要がある．

2.2 構成指向計画 (COP: Configuration-Oriented Planning)

構成指向計画 (COP) は，筆者らが研究開発を進めている構築自動化技術である [8]．COP の最も重要な特徴は，入力された所望のシステム構成に基づき，その構築処理を自動生成する点にある．システム構成の定義仕様はコンポネントの再利用性や簡潔さを重視した設計となっており，各コンポネントには構成要素毎の配備処理やその実行に必要な制約条件などの要件が抽象的に定義される．これらの要件はコンポネント間の連結状況に応じて重ね合されることで具体化され，手順計画処理の入力情報として利用される．

図 1(b) は，COP におけるシステムの記述例である．前述の図 1(a) における概念に加え，フィールド，委譲，変換などの概念が導入されている．この例において，*Apache* 上の *File* 要素は配備処理の実装を持たない抽象的な構成要素として定義されており，その実現はリファレンス・ポートを介してワイヤで連結された *Ubuntu* へ委譲されている．一方の *Ubuntu* は，*Ubuntu* に特化した *File* 要素である *UFile* 要素を格納するフィールドを備えており，ここに *Apache* から委譲された *File* 要素を *UFile* 要素に変換した上で格納している．変換とは *File* 要素に設定されたプロパティ (属性値) を *UFile* 要素のプロパティに割り当て直して *UFile* 要素を生成する操作を示す．ここでポートは *File* 要素を媒介するインタフェースの役割を果たしている．

本方式では，*File* 要素の配備処理の実装は *Ubuntu* コンポネント上に定義される．より詳細には，*Ubuntu* に特化した構成要素型である *UFile* の定義内に Ubuntu 上での実行を前提としたファイルの配備処理が実装される．図 1(a) の方式と比較すると *Apache* コンポネントと *Ubuntu* コンポネント間の依存関係が解消されており，*Apache* は同様のサービス・ポートを備える他の OS のコンポネントとも連携可能である．

しかしながら本方式においても，配備エージェントについては Ubuntu 上に予め備わっていることを前提としており，柔軟に組換えられないために構成選択上の制約を生み出している．例えば，MySQL における DB インスタンスの配備処理の実装には Ansible で既に用意されたライブラリが活用できるが，このような *MySQL* コンポネントを利用するには Ansible の利用環境が整備された *Ubuntu* コンポネントを選択する必要がある．また，異なる配備エージェントを利用するコンポネントを同時に利用したい場合には，要求される全ての配備エージェントを備えた *Ubuntu* コンポネントが必要となるが，これに対応する多様なコンポネントを用意することは困難である．

3. 配備エージェントの組換えを可能とする拡張型 COP の提案

2 章で述べた課題を解決し，より柔軟かつ高効率にシステム構成を定義可能とするため，本研究では以下の 3 点からなる COP の拡張を提案する．

- 配備エージェントの構成要素化
- 配備エージェントのコンポネント化
- 配備エージェント実装の外部モジュール化

以下，本章ではこれらの提案について順に述べる．

3.1 配備エージェントの構成要素化

本提案手法では，配備エージェントは配備エージェント要素と呼ばれる構成要素の一種として構成定義内に明示的に表現される．配備エージェント要素は，ある特定の配備エージェントを制御する構成要素である．また配備エージェント要素は特定の配備エージェントのインスタンスを示すポインタの役割を果たしており，これを他の構成要素やフィールドと関連付けることにより，当該構成要素や当該フィールドに格納された構成要素の配備処理を実行する配備エージェントを明確に指定できる．構成要素と配備エージェント要素との関連付けの具体例は次節で述べる．また配備エージェント要素の記述方法は 4.1.2 節で述べる．

3.2 配備エージェントのコンポネント化

COP において，組換え可能な部品はコンポネントとして定義される．本手法では配備エージェントについてもコンポネントとして明示的に定義し配備対象の一部として扱う．図 1(c) は提案手法の COP におけるシステムの構成例である．*Apache* コンポネントと *Ubuntu* コンポネントの間に，*Ansible* コンポネントが追加されている．*Ansible* コンポネントは *AFile* 要素型のフィールドを持つ．*AFile* 要素型は *Ansible* に特化した *File* 要素である．*Ansible* コンポネントは *Apache* コンポネントから委譲された *File* 要素を *AFile* 型に変換した上で当該フィールドに格納している．*File* 要素の配備処理は *AFile* 要素型の定義内に Ansible の利用を前提として実装される．*Ansible* コンポネント内には更に *Ansible* 要素が定義されている．これは *Ansible* インスタンスを示す配備エージェント要素である．*AFile* フィールドは *Ansible* 要素と関連付けられているため，*Apache* コンポネント内の *File* 要素は *AFile* フィール

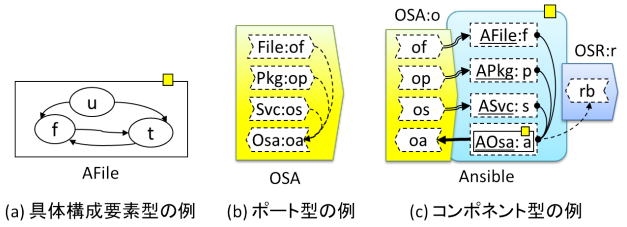

図 2 モデル定義の例
Fig. 2 An example of models.

ドを介して *Ansible* 要素に割り当てられることとなる．結局，*File* 要素は，*AFile* 要素により配備処理の実装を与えられ，その実行は *Ansible* インスタンスによって行われる．

以上の仕組みにより，配備エージェントの存在は *Ubuntu* コンポネントや *Apache* コンポネントから切り離され，疎結合となるため，他の配備エージェントを追加したり或いは交換したりするなど，柔軟な構成定義が可能となる．コンポネントの記述方法は 4.1.4 節で述べる．

3.3 配備エージェント実装の外部モジュール化

配備エージェントが実行する処理は最終的には何らかのプログラミング言語によって記述される．ここで使用される言語は開発者の意向により自由に選択可能であることが望ましい．また，コンポネントや構成要素の記述内容は宣言的な定義であるため，ここに処理を追記するような仕様は定義全体の保守性を低下させる要因となる．そこで本手法では配備エージェントの実装を外部モジュール化し，構成要素内には API を介して配備エージェントのモジュールを呼び出す定義のみを記述するスタイルを採る．これにより，配備エージェントの実装における共通的な処理は外部モジュールに一元化され，構成要素では簡潔な宣言のみで配備処理の実装が記述可能となる．このような配備エージェントの実装方法は 4.1.1 節で述べる．

4. 拡張型 COP の設計と実装

本章では，まずシステム構築に必要となる各種モデルの具体的な定義仕様とこれらの定義に基づく構築処理の導出方法について述べ，続いて実行系を含めた全体アーキテクチャについて述べる．なお配備エージェントに関する要素以外の要素の設計は既存の COP とほぼ同様であるが，本稿の提案事項と密接に関わるため併せて説明する．

4.1 モデル定義仕様

モデル定義は，配備エージェントの実装以外は独自の DSL により定義される．しかしながら本稿では DSL 文法までの詳細は省略し，定義内容の説明のみに留める．

4.1.1 配備エージェントクラス

配備エージェントの実装は何らかのオブジェクト指向言語におけるクラスとして表現できる．このクラスには配備エージェントが扱う各種の構成要素について，これを管理するための作成，更新，削除などのメソッドが定義される．これらのメソッドを API として呼び出すことにより，構成要素では所望の配備処理を簡潔に記述できる．各メソッドは共通的な引数構成に則って実装され，構成要素から各種オプションの情報を受け付ける．さらに配備エージェントのインスタンスを管理するためにコンストラクタやデストラクタが定義される他，インスタンスごとに保持すべき情報がインスタンス変数として定義される．

例えば *OpenStackAgent* であれば，*createVM* や *deleteVM* などのメソッドが定義され，それぞれ *imageId* や *vmId* などのオプションを受け付ける．またインスタンス変数として特定の OpenStack サービスへの接続情報などを保持し，コンストラクタでは接続確認が行われる．

4.1.2 構成要素型

構成要素型は役割に応じて 3 種類に分類される．以下に，各種の構成要素の役割と定義内容について述べる．

- 具体構成要素型: 構成要素を具体的に利用可能に定義する型．構成要素が取り得る状態と可能な状態遷移の情報を持ち，各状態遷移にはこれを実行するための処理が定義される．前提とする配備エージェントの型を指定可能であり，処理は当該配備エージェントが持つメソッドとオプションにより定義される．プロパティを定義可能であり，ここに値を渡すことで処理内容を外部から調整できる．図 2(a) は *AFile* 要素の例である．楕円は取り得る状態を示しており，f は “存在しない”，t は “所望の状態で存在する”，u は “存在するが内容が所望の状態でない” ことを意味している．例えば $f \rightarrow t$ の遷移はファイル作成に相当する状態変化であるため，ここには *AnsibleAgent* の *createFile* メソッドが指定される．プロパティには，例えば *fileName* などが定義され，その値をメソッドのコール時のオプションに渡すことでファイル作成処理が定義できる．
- 配備エージェント要素型: 配備エージェントクラスとの紐付け情報が規定された型．それ以外の内容は具体構成要素型と同様である．
- 抽象構成要素型: プロパティのみが定義された型．具体的な処理の定義を避けることで多様な状況で再利用可能な要件が定義される．最終的に具体構成要素に変換されることで実行可能となる．

尚，具体構成要素には任意の取り得る状態や状態遷移を定義可能であるが，本稿で例示する具体構成要素はいずれも図 2(a) と同様の状態や状態遷移を持つものとする．

4.1.3 ポート型

ポート型はコンポネント間の接続可能性を機能の観点で規定するインタフェースである．フィールドにより媒介可能な構成要素が定義される．ポート型におけるフィールドには，機能の利用側から提供側へと構成要素を媒介する

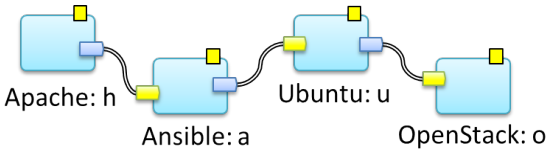

図 3 システム構成定義の例
Fig. 3 An example of system configuration.

フィールドとその逆向きに媒介するフィールドとがあり，前者をアクセプト・フィールド，後者をプロバイド・フィールドと呼ぶ．各フィールドには，ここに格納される構成要素が従うべき制約条件が定義できる．

図 2(b) は *OSA* ポートの例である．ここで *OSA* は OS Agent の略であり，Ansible や Puppet など OS 上での各種操作を代行する配備エージェントとしての機能を意味している．*File*，*Pkg*，*Svc* は，それぞれファイルの操作，パッケージの操作，サービスの操作に関する抽象構成要素型であり，*Osa* は OS Agent の操作に関する抽象構成要素型である．なおフィールドを示す山形の向きは，それがアクセプト・フィールドかプロバイド・フィールドのいずれであるかを示している．フィールド間の矢印は制約条件を示しており，ここでは “*Osa* 要素が存在しなければ各構成要素は状態遷移できない” といった制約が記述されている．

4.1.4 コンポネント型

コンポネントはシステムを構成する部品を定義する要素である．コンポネント型には，構成要素，フィールド，リファレンス・ポート，サービス・ポートおよびプロパティが定義される．ここで，各ポートはポート型のインスタンスとして定義される．構成要素やフィールドには，当該構成要素または当該フィールドに格納される構成要素との間の関連性について定義できる．具体的には，従うべき制約条件や配備エージェントの割り当てなどの関連性が定義できる．また当該コンポネントや各ポートが保持するフィールド間における構成要素の委譲や変換が定義できる．

図 2(c) は *Ansible* コンポネントの例である．*OSA* 型のサービス・ポートを持ち，当該ポートから受け渡される各種構成要素を，*Ansible* コンポネント特有の具体構成要素型に変換してそれぞれのフィールドに格納している．これらのフィールドは *AAgt* を配備エージェント要素として関連付けられている．また，*AAgt* 要素はサービス・ポートを介して，機能の利用側から参照可能に提供されている．*OSR* は OS Resource の機能を表すポートであり，このリファレンス・ポート内の *ri* フィールドには，当該 Agent が稼働する OS の起動に関する構成要素が受け渡されることが期待できる．*AAgt* 要素のフィールドから *ri* フィールドへの制約条件は，“OS が起動していなければ Agent は操作できない” といった制約を示している．

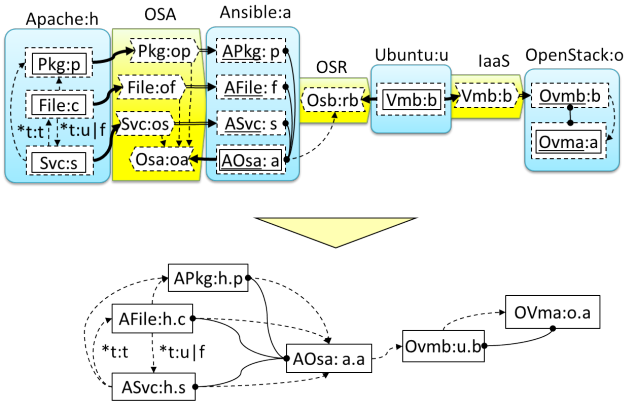

図 4 構成要素インスタンスの導出過程の例
Fig. 4 An example of compilation process.

4.1.5 システム構成定義

システム構成定義は任意のコンポネントのインスタンスの組み合わせで定義される．各コンポネントのインスタンスは，コンポネント型と代入するプロパティ値により定義できる．コンポネントの組み合わせは，対象となるコンポネントのサービス・ポートとリファレンス・ポートの組を表すワイヤで定義できる．尚，組み合わせ可能なポートは型が一致するものに限定される．

図 3 はシステム構成定義の例である．この例では，*Apache*，*Ansible*，*Ubuntu*，*OpenStack* の 4 つのコンポネントから構成されたシステムが定義されている．

4.2 構築処理の導出と実行

本稿で提案する拡張型 COP における構築処理の導出方法は，概ね既存の COP における導出方法と同様である．本稿では配備エージェントに関する部分を中心に詳細を述べ，最後に実行系の動作について述べる．

4.2.1 構成要素インスタンスの導出

まず入力されたシステム構成定義内のコンポネント型やポート型の定義およびワイヤの定義に基づいて，フィールド間の委譲と変換の関係を構築する．続いて，このフィールド間の関係に基いて全ての抽象構成要素を具体構成要素に変換すると共に，委譲や変換の過程でフィールドを介して定義された関連性を回収し構成要素自体に転記する．

図 4 に，システム構成定義から構成要素インスタンスを導出する過程の例を示す．例えば *Apache* 内の *File* 要素は *OSA* の *File* フィールドを介して最終的に *AFile* 要素へと変換される．また，この過程で *Apache* 内の *Pkg* 要素や *Svc* 要素，*Ansible* 内の *AOsa* 要素などと関連付けられる．これを全ての構成要素定義について実施することで，図 4 の下部に示した構成要素インスタンスが導出される．インスタンス内のラベルはそれぞれの具体構成要素型とインスタンス ID を示している．ここで，実線は配備エージェントの割り当てを示しており，点線の矢印は制約条件を示し

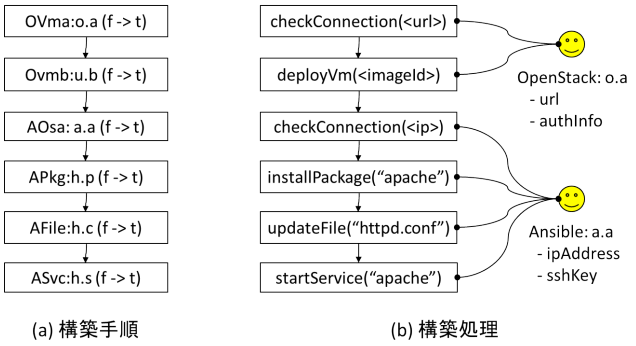

図 5 構成処理の生成過程の例
Fig. 5 An example of process generation process.

ている．制約条件の内，ラベルの付いたものは制約条件の詳細を示している．例えば “*t:u|f” の記載は，“元の構成要素を任意の状態から t 状態へ遷移させるには，先の構成要素が u 状態か f 状態でなければならない” ことを意味する．ラベルの無い制約条件は全て “**:t” と同義である．

4.2.2 構築手順の導出

前節で導出した構成要素インスタンスに基づき，全ての構成要素を f 状態から t 状態へ制約条件を満たしつつ遷移させるような状態遷移の順序を探索することにより，システムの構築手順を導出する．手順導出手法の詳細については，筆者らの文献 [7] を参照されたい．当該文献の手法を用いれば，必要な状態遷移の手順が有向非巡回グラフ (DAG: Directed Acyclic Graph) として高速に導出できる．

図 5(a) は導出された構築手順の例である．各ステップは操作対象の構成要素と実行される状態遷移からなる．尚，この例では生成された手順は全順序列であるが，例えば複数の VM の起動など並列実行可能なステップがある場合にはしかるべき半順序列が生成される．

4.2.3 構築処理の生成

構築処理は前節の手法で得られた構築手順を基に，各ステップに対して処理の実装を注入し配備エージェントを割り当てることで生成できる．

図 5(b) は生成された構築処理の例である．各ステップの処理の実装は，各構成要素インスタンスの具体構成要素型において該当する状態遷移に指定された配備エージェントへのメソッド・コールである．図中の各ステップのラベルにはコールするメソッド名やオプションを記載している．ここでは考え方を端的に示すため，オプションは代表的な物だけを一部概念的に記載しているが，当然ながら実際には多数のオプションについて具体的な値が与えられる．こうしたオプションの値は構成要素やコンポネントのプロパティを通じて取得される．

配備エージェントは配備エージェント要素のインスタンスごとに生成され，それぞれ配備エージェント要素のプロパティから得られた接続情報などを保持する．配備エージェント要素およびこれに割り当てられた構成要素に関する処理が，当該配備エージェント要素に相当する配備エージェントによって実行されることとなる．

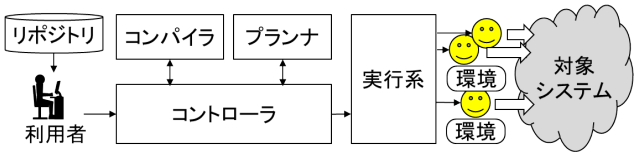

図 6 拡張型 COP の全体アーキテクチャ
Fig. 6 Architecture of extended COP.

4.2.4 構築処理の実行

構築処理の実行は，指定された配備エージェントを用いて構築処理内の各ステップを順次実行するのみである．構築処理が半順序列である場合には，順序関係のないステップは並列に実行し，また複数のステップに対して順序性を持つステップがある場合には先行するステップがすべて終わったタイミングで実行する．本手法の構築処理には配備エージェントに関する操作や制約がすべて考慮されているため，配備エージェントについても必要なタイミングで利用環境が整備されながら，構築処理の全体が完遂される．

4.3 全体アーキテクチャ

図 6 に，拡張型 COP の全体アーキテクチャを示す．図中のリポジトリにはコンポネント等の各種定義が保持されている．コントローラは利用者からシステム構成定義の入力を受け、構築完了に至るまでの行程を制御する．コンパイラはシステム構成定義をコンパイルし，構成要素インスタンスを生成する．プランナは構成要素インスタンスに基づき，しかるべき構築手順を導出する．また実行系は構築処理の情報を受け，配備エージェントを用いて自動構築を実施する．配備エージェントは必要に応じて個別の実行環境に配置される．例えば，Ansible クライアントは Linux 系 OS でのみ動作可能である一方で，PowerShell DSC は Windows 系 OS でのみ動作可能であるため，それぞれ Linux 環境と Windows 環境に配置される．

動作シーケンスは次の通りである．まず利用者はリポジトリ内のコンポネント定義を参照しつつ所望のシステム構成を定義し，これをコントローラへ入力する．コントローラはコンパイラによってシステム構成定義をコンパイルし，構成要素インスタンスを生成する．続いてコントローラは得られた構成要素インスタンスをプランナに入力し，構築手順を導出する．さらにコントローラは得られた構築手順に構成要素インスタンス内に定義された処理実装の情報を盛り込むことで最終的な構築処理を生成し，これを実行系に入力することでシステムを自動構築する．

4.4 実装

本稿で述べた手法のプロトタイプは CoffeeScript で実装されており，Node.js 上で実行・テストされている．図 6

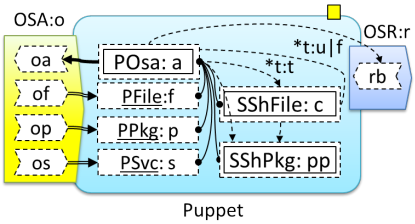

図 7 Puppet コンポネントの設計例
Fig. 7 An example design of puppet component.

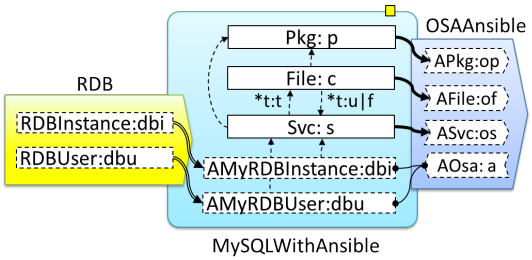

図 8 特定の配備エージェントを活用したコンポネント定義の例
Fig. 8 An example of component definition with utilizing particular deployment agent.

に記載の各モジュール間の通信には http を用い，その内容は JSON 形式で表現されている．各種モデルやシステム構成定義および構築処理の定義などは JSON を基にした独自の DSL で記述されている．また，配備エージェントの実装も CoffeeScript によって試作されている．

5. 評価と考察

本章では，まず 4 章で示したサンプルシステムの構築例を基に本手法の実用性について議論し，続いて各種配備エージェントの利用に必要となる作業量の観点から本手法の有効性を評価する．また提案手法の更なる活用例を示し，最後に今後の課題を含めた考察を述べる．

5.1 サンプルシステムの構築事例に基づく評価

4 章で述べたように，本稿では図 3 のサンプルシステムの構築例により，コンポネントとして定義された配備エージェントを用いた一連のシステム自動構築の実現可能性を示した．また，システム構築に必要となる定義が十分に簡潔であることを示し，利用者がこれを記述してシステムを構築するという本手法全体の実用性について確認した．

図 3 の例に示したように，本手法においては配備エージェントをコンポネントとしてシステム構成定義上に明示的に表現する．そのため，利用する配備エージェントはコンポネントの選択により組み替えできる．ここで図 8 に *Puppet* コンポネントの設計例を示す．但し本稿執筆時点において *Puppet* コンポネントは実装レベルでは未検証であり，ここでは設計のみを示している点に注意されたい．*Puppet* は *OSR* 型のサービス・ポートを備えており，図 3 のシステムにおいて *Ansible* コンポネントの代わりに利用できる．この場合，例えば *Apache* 内の *File* 要素は *PFile* 要素に変換され，*Puppet* クラスの対応するメソッドをコールする．*Puppet* クラスは実行環境上の PuppetMaster を通じて PuppetClient に対する命令を実行する．

本手法では，配備エージェント自身の準備作業も構築処理内で実施できるため，事前の準備作業が効率化できる．*Ansible* 要素については，プロパティを介して IP アドレスや利用する SSH 秘密鍵を指定することで簡単に動作可能にセットアップできる．また *Puppet* に関してはそれ自体のインストールが必要となるが，これはコンポネント内部の *SShPkg* 構成要素などにより実装可能と考えられる．*SShPkg* 構成要素は SSH を介してパッケージの操作を実施することを想定している．

*3 CoffeeScript,http://coffeescript.org/ (確認日 2016/01/27)
Node.js,https://nodejs.org/ (確認日 2016/01/27)

5.2 定義作業の効率改善効果に関する評価

本提案手法では，配備エージェントはコンポネントの追加により柔軟に利用可能であり，要求に応じた配備エージェントを備えた OS のコンポネントを予め用意しておく作業は不要である．ある OS 上で動作可能な配備エージェントが n 種類存在するとき，要求される可能性のあるあらゆる配備エージェントの組み合わせの数は 2 項係数の和より 2^n となる．そのため，仮に m 種類の OS のそれぞれに対してあらゆる要求を網羅した OS のコンポネントを用意する場合には，定義すべきコンポネント数は $2^n m$ と試算できる．厳密には 4.3 節でも述べたように OS 種別に応じて動作可能な配備エージェントは異なるが，ここでは便宜上全ての配備エージェントが全ての OS 種別上で動作可能と考える．一方，本手法の COP では，各配備エージェントや OS 向けにそれぞれ n 個と m 個のコンポネントを用意するだけで，要求に合わせた任意の組み合わせが可能であるため，必要なコンポネントの数は $n + m$ と試算できる．利用が想定される OS の種類や配備エージェントの種類はバージョンの違いも考慮するとある程度の数が想定されるため，これらの試算の絶対値の差は非常に大きくなる．

実運用上は，従来手法において全ての可能性を網羅することは考え難く，何らかの回避策を採ることになるが，しばしば場当たり的な対応が運用上の不具合を生じる原因となる．一方，提案手法においては回避策を採らずとも，少数の定義により，従来手法で全ての可能性を網羅的に準備した場合と同等の高い柔軟性が得られているといえる．

5.3 他の活用例

5.3.1 例 1: Ansible を用いた MySQL コンポネント

2.2 節でも述べたように，Ansible は MySQL 上でのユーザ作成や DB インスタンス作成のための機能を有する．こ

うした特定の機能の利用を前提に配備処理を実装可能であれば，コンポネントはより効率的に開発できる．

図 8 に，Ansible の活用を前提とした MySQL コンポネントの定義例を示す．このコンポネントは，リファレンス・ポートとして *OSAAnsible* 型のポートを備えており，ここから Ansible の配備エージェント要素が提供されることが想定されている．2 つの具体構成要素型 *AMyRDBInstance* と *AMyRDBUser* を実行する配備エージェントには当該 Ansible の配備エージェント要素が指定されており，これらの具体構成要素内の配備処理の定義は Ansible の配備エージェントクラスのメソッドにより効率的に定義できる．

5.3.2 例 2: マルチクラウド型システムへの応用

本稿の提案手法は複数のクラウド環境からなるマルチクラウド型システムに応用できる．本手法においては，各クラウドサービスをコンポネントとして定義することで，複数のクラウドサービスにまたがる大規模なシステムを単一のシステム構成定義上に表現し，一括して自動構築可能である．クラウドサービスをまたいだ順序依存がある場合でも，実行系により適切に制御できる．

5.4 考察

図 8 の *MySQL* コンポネント内に定義された 2 つの具体構成要素では，Ansible の配備エージェントクラスのメソッドを利用している．しかしながら，当該配備エージェントクラスは，必ずしも対応する *createMySQLDbInstance* などのメソッドを備えていなくてもよい．その代わりに，利用する機能をオプションとして指定可能な汎用的なメソッドを定義しておき，具体構成要素の実装として必要となる MySQL 関連の Ansible 機能を指定すればよい．このような方法によれば，配備エージェントクラスが実装を追加せずとも，ツールが持つ機能を柔軟に利用してコンポネントの定義を追加できる．

最後に，実行系の機能に関する今後の課題について述べる．本稿で述べた実行系は，与えられた構築処理を順次実行していく機能しか持たないため，実用に向けては (1) 例外処理への対応，(2) 配備中に発生する値の利用，などの課題を残している．例外処理については，最も単純には全ての動作を停止し，適宜構築中の VM を廃棄した上で構築処理をやり直す対応が考えられる．しかしながら，配備対象が常に廃棄可能な仮想リソースであるとは限らず，また軽微な例外であれば適切に回復して構築を再開することで早期に構築が完了できることが望ましい．配備中に発生する値については，例えば VM に対して動的に与えられる IP アドレスなどの情報を利用して関連するミドルウェアの設定を行うケースが考えられる．現状では，全ての設定値を事前に与える設計となっており，こうした柔軟性を備えた機構の開発が求められる．

6. おわりに

本稿では，配備エージェントが組換え可能なテンプレート型プロビジョニング手法として拡張型の COP を提案し，その設計や利用に必要となる各種定義の記述仕様などの詳細について述べた．また，事前に用意が必要となるコンポネント数の削減効果から本手法の有効性を評価した．さらに，サンプルシステムの構築例やマルチクラウド型システムへの応用などについて述べ，提案手法の実用性について議論した．今後は，例外処理への対応や構築中に生成される値の利用など，より高度な配備機能の研究開発を推進していく．

参考文献

[1] Binz, T., Breiter, G., Leyman, F. and Spatzier, T.: Portable Cloud Services Using TOSCA, *Internet Computing, IEEE*, Vol. 16, No. 3, pp. 80–85 (online), DOI: 10.1109/MIC.2012.43 (2012).

[2] Breitenbucher, U., Binz, T., Kepes, K., Kopp, O., Leymann, F. and Wettinger, J.: Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA, *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pp. 87–96 (online), DOI: 10.1109/IC2E.2014.56 (2014).

[3] Derek, P. and Thomas, S.: Topology and Orchestration Specification for Cloud Applications Version 1.0 (2013).

[4] Eilam, T., Elder, M., Konstantinou, A. and Snible, E.: Pattern-based composite application deployment, *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, pp. 217–224 (online), DOI: 10.1109/INM.2011.5990694 (2011).

[5] Fischer, J., Majumdar, R. and Esmaeilsabzali, S.: Engage: A Deployment Management System, *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, New York, NY, USA, ACM, pp. 263–274 (online), DOI: 10.1145/2254064.2254096 (2012).

[6] Herry, H. and Anderson, P.: Planning with Global Constraints for Computing Infrastructure Reconfiguration, *CP4PS-12 - The AAAI-12 Workshop on Problem Solving using Classical Planners*, (online), available from ⟨http://homepages.inf.ed.ac.uk/dcspaul/publications/CP4PS-12.pdf⟩ (2012).

[7] Kuroda, T. and Gokhale, A.: Model-Based IT Change Management for Large System Definitions with State-Related Dependencies, *Enterprise Distributed Object Computing Conference (EDOC), 2014 IEEE 18th International*, pp. 170–179 (online), DOI: 10.1109/EDOC.2014.31 (2014).

[8] Kuroda, T., Nakanoya, M., Kitano, A. and Gokhale, A.: The Configuration-Oriented Planning for Fully Declarative IT System Provisioning Automation, *To Appear in the Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS), 2016 IEEE* (2016).

[9] Kuroda, T. and Gokhale, A.: Model-based Automation for Hardware Provisioning in IT Infrastructure, *the Proceedings of Systems Conference (SysCon), 2014 IEEE International* (2014).