Vol.2019-CDS-25 No.5
2019/5/31

動的適応部品による自己適応ソフトウェアの検討

松塚 貴英[†1]　飯田 一朗[†2]

概要：ソフトウェアは、それが実行される環境との関係において、様々な不確実性がある。その不確実性に対処するために、従来は実行フェーズから開発フェーズに戻ってソフトウェアの修正を行っていたが、近年のデジタル化に伴う IoT などの新しいアプリケーション領域では不適切になってきている。自己適応はそのような不確実性に対し動的に対応するための技術である。本論文では、ソフトウェアの実行環境をオブジェクトの木構造で構造化し、オブジェクトに対し適応を実現する部品をアタッチすることで自己適応を実現する方法を論じる。

キーワード：自己適応, デジタルツイン, CPS

Investigation on Enabling Self-adaptation Software with Dynamic Adaptation Components

Taka Matsutsuka[†1]　Ichiro Iida[†2]

Abstract: Software has uncertainty in relation to its execution environment. To deal with the uncertainty, traditionally developers go back to the development phase to modify the software from the execution phase. This is becoming inappropriate in applications that need to handle frequent environmental changes such as IoT. Self-adaptation is a technology to address such uncertainty with dynamism. This research introduces the tree structure of objects for the execution environment, and attaches self-adaptation parts to the object in order to realise self-adaptation.

Keywords: Self-adaptation, Digital Twin, CPS

1. 背景

近年のソフトウェアは、IoT などのデバイスの発展とクラウドや API エコノミーに代表されるインターネット上でのサービス利用の一般化により、利便性と Time-to-market の両面で著しい進展を見せている。一方、単一のコンピュータに閉じずに種々のデバイスやインターネットに開かれたシステムは、その内部および外部において様々な不確実性を持つ。ソフトウェアが実行される環境に様々な変化が発生するため、配備前に予測することが困難な不確実性を内包している[Weyns2017][Musil2017]。

従来、ソフトウェアは開発と実行が明確に分かれており、コンピュータサイエンスでもそのような立場が現在でも支持を得ている[Baresi2010]。例えば、DevOps という概念は、開発及び運用、すなわち開発フェーズと実行フェーズの間を有機的にブリッジすることにより、双方の担当者が効果的に共同作業できる機会を与えている[Lwakatare2015]。しかし、それでも開発(Dev)と運用(または実行)(Ops)は明確に分かれた 2 つのフェーズであり、これらをいかにスムーズに相互作用させるかという点が大きな技術的、文化的チャレンジとなっている。

前述したように、ソフトウェアは実行フェーズに入った後も絶えず変化にさらされている。この変化は大きくは 3 種類に分けることができる。すなわち、システムと相互作用するエンティティ(人間であるユーザであることも、コンピュータであることもある)、システムの外部世界である動作環境、そしてシステム自身である[Lemos2009]。すなわち、これらの変化にどのように迅速に適応していくかがソフトウェアの大きな課題となっている。

そのこと自体は DevOps などの開発実行の相互作用を否定するものではない。問題は、ソフトウェアは実行時にしか観察できないが、その観察された結果に基づいて実行中に影響を及ぼしたり変更したりすることがほとんど不可能であるということである[Baresi2010]。

このような問題に対するアプローチとして、自己適応が提案されており、様々なアプローチで研究が行われている。自己適応とは、環境の変化に応じてシステムの動作を調整するシステムの機能である。「自己」という接頭辞は、システムが状況や環境の変化に対応できるように、適応方法を

†1 (株)富士通研究所
Fujitsu Laboratories Ltd.
†2 秋田県立大学
Akita Prefectural University

自律的に(人間の干渉がないか、または最小限で)決定することを示す[Macias2013] 。

本論文では、IoT などのシステム周囲の環境の変化が多く観察されるシステムにおいて、適応を行うためのアプローチについて論じる。2 章では、本論文の背景となる、ソフトウェアの動作環境のモデル化について論じる。3 章では、提案手法について説明する。4 章では他の手法との比較を含めた議論を行い、5 章で結論および今後の方向性について論じる。

2. 自己適応のアプローチと従来手法の課題

2.1 自己適応のアプローチ

適応を可能にするためには、ソフトウェアの動作環境および内部状態をモデル化し、状況の変化に合わせてこれらのモデルをアップデートしていく方法が有効である。モデル化には様々な方法があり、適応の手法とも密接な関係がある[Salehie2009]。

自己適応には、大きく分けて Internal および External の 2 種類の実装方法がある。Internal アプローチでは、適応を行うロジックとシステム機能を提供するロジックが一緒に一つのコンポーネントで実装される。External アプローチは、図 1 に示すようにシステムの機能を提供するアプリケーションロジックを提供する要素(Managed subsystem)に対して、適応の監視と実行のためのサポートを提供する要素(Managing subsystem) で構成されている [Weyns2013]。Internal アプローチでは保守性や拡張性において難がある一方、External アプローチでは適応を行うコンポーネントを再利用できるという利点があるため、本論文でも External アプローチを採用する。

本論文では、IoT などソフトウェアの外部に様々なデバイスが存在するシステムを想定アプリケーションに含んでいる。このようなアプリケーションシステムは、Cyber-Physical System (CPS)とも呼ばれ、環境の変化や、人間も含むシステムの構成要素同士の複雑な相互作用により、高い不確実性を内包している。

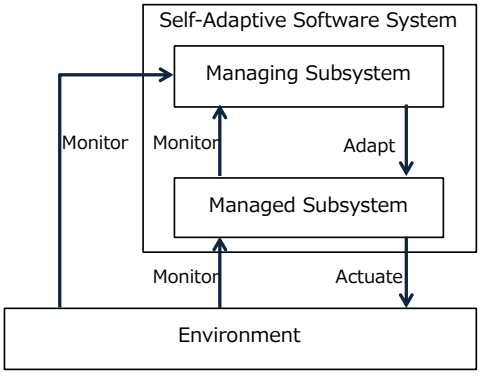

図 1 自己適応システム ([Salehie2009]より)

2.2 Illustrative case

本論文が対象とする事例について述べる。

ここでは、倉庫の中で荷物移動を行う AGV (Automated Guided Vehicle、自動搬送車)の例を考えてみる。AGV は、装備するセンサーで周囲の状況を確認しながら、荷台と車輪が付いていて自走できる、工場や倉庫などで自動で動作する車両であり、荷物運搬に使われている。

ここで例にする倉庫には棚と通路があり、AGV の目的は、通路を走りながら、倉庫のある棚に置いてある荷物を、別の地点まで運ぶことである(図 2)。荷物の大きさや重さは様々であるが、AGV は荷物を落とさないように、また周囲にぶつからないように運ばなければならない。

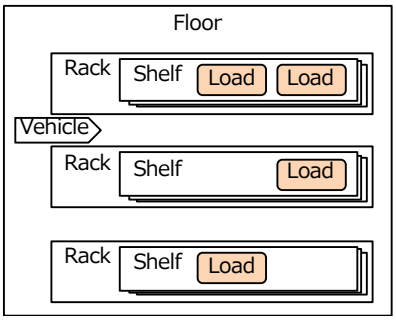

図 2 倉庫

ここで、適応が必要になるのは荷物を落としそうな時、または落としてしまった時である。

AGV は、荷物を落としそうであることを重心の移動などで感知したら、落とさないような適応動作を行う必要がある。その時の適応動作にはいくつかの可能性がある。例えば、荷台が動いて重心移動を補償する、または、車両全体が重心の移動方向に動いて荷物を落とさないようにする、などである。

もし AGV が荷物を落としてしまったら、回復動作を行う。たとえば当該の通路を通行止めにして、人が荷物を片付けるかもしれないし、別のロボットにより片付けが行われるかもしれない。

2.3 課題

前節の事例を、我々のターゲットとなる CPS で実現しようと考えた場合、以下の課題がある。

2.3.1 例外対応のシステムへの組み込み

AGV の動作においては様々な例外事象が発生する可能性がある。そのため、特にルーティングや衝突回避などに関して、様々なメカニズムが提案されている。例えば、AGV の動作範囲を静的あるいは動的なゾーンで区切り、あるゾーンには AGV が 1 台のみ入れるようにする、AGV が何かにぶつかりそうになったら後退し再ルーティングを行う、などである[Le-Anh2006]。

これらのメカニズムに共通するのは、どういう種類の例

外に対処するか、あらかじめ決めてシステムに組み込んでおく点である。

本論文で挙げた例では、「荷物を落としそうになる(または落とした)」という例外に対して様々なレベル(荷台で対応可能、車両全体で対応が必要、等)があり、それに対する対処も様々な方法がありうる。そのため、あらかじめ対処すべき例外に対する適応をシステムに組み込んでおく方法では、対処方法が変わるたびに例外対処方法を組み込み直さなくてはならないため、システムのダウンタイムを要するなど手間がかかってしまう。

このことは、適応のメカニズムに External アプローチをとる必要があることを意味する。しかし、その場合は適応を行うソフトウェアコンポーネント、つまり Managing Subsystem と適応を受ける(例外が発生する)ソフトウェアコンポーネント、つまり Managed Subsystem が対応関係になるよう、注意深くソフトウェア設計を行う必要がある。

2.3.2 適応機構の選択

システムはひとたび例外が発生した時に、どこでどのように適応を行うか決定できなければならない。DEECo [Bures13]では、自動車の効率的なナビゲーションを実現するために、周辺の信号や道路と動的に協調しながら適応を行う仕組みを提案している。しかし、ある例外が発生したときにどこで適応動作を行うか、という点について明確なアプローチを持っているわけではなく、事例では自律的なコンポーネントである自動車が適応を行なっている。

本論文の例において、AGV が「荷物を落としそうになる(または落とした)」という例外を考えてみると、その事象に対する対処としては、AGV の荷台が動くこともあるだろうし、車両全体が動くこともあるかもしれない。さらに、実際に落としてしまった場合のために通路を通行止めにする準備が必要かもしれない。ある例外事象に対応するために、適応動作も様々な箇所で発生することになる。

ここで、ある例外に対する適応動作が複数の箇所(上記の場合は荷台、車両および通路)で定義されていたとき、どの適応動作から開始すればよいかという問題が発生する。ちょっとしたバランスの崩れであれば荷台が動くことにより対処できるだろうし、倒れていないのに通路を通行止めにするは無駄である。

別の見方をすると、ある適応動作がうまくいかなかったときに、他の適応方法で対処する仕組みが必要である。つまり、荷台で適応しようとしてうまくいかなかったら、車両が動いて適応を行いたい。それでも適応できず荷物が落ちてしまうようであれば、通路を通行止めにする必要があるかもしれない。

3. 提案手法

前節で議論したような課題に対応するため、本論文では、ソフトウェアが動作する環境に存在する事物をオブジェクトで表し、オブジェクトを構造化することで系全体を表現するアプローチを取る。以下でその詳細について説明する。

3.1 オブジェクトと適応部品

サイバーフィジカルシステム(CPS)では、環境に存在する事物を、それぞれの事物に対応するオブジェクトで表現する。このオブジェクトは Digital Twin (デジタルツイン、または単に Twin)とも呼ばれる[Negri2017]。Twin のアプローチでは、オブジェクトを利用してシミュレーションや分析を行うことで、製品設計を安価に行ったり、製品の予防保守を行ったりする[Madni2019]。本論文でも、環境上の事物をソフトウェアのオブジェクトで表現するアプローチを取るが、製品設計や保守の代わりに、オブジェクトを利用して実行時の自己適応を行う。

AGV の例では、自動搬送車を構成する車両本体、車輪、荷台などがオブジェクトになりえる。また、倉庫内に存在する通路やラックなどもオブジェクトで表現できる。これらのオブジェクトは、実世界上にある実物の状況をセンサー等で把握することにより、現実の状況をソフトウェア上で再現する。

具体的には、オブジェクトとは対象の事物に関する情報を持つソフトウェアコンポーネントである。対象となる事物は、2.2 節の例では現実に存在する物体であるが、それに限らない。例えば、後述する 4.1 節では、ソフトウェアのエンティティをオブジェクトとして扱っている。

本節で導入するオブジェクトの実装としてはオブジェクト指向言語の「オブジェクト」でもいいし、単なる構造体でも構わない。オブジェクトの粒度はアプリケーション依存だが、適応はオブジェクトの単位で行われるため、適応を別個に行う事物同士は別のオブジェクトにする必要がある。

オブジェクトに対し、適応を実現するため、適応部品を導入する。適応部品は、オブジェクトが表現する事物や、オブジェクトが動作するソフトウェア環境を観察し、何らかの適応が必要な事態になったことを判断し、必要に応じて対象部品に対して働きかけを行うことにより、適応を行うものである。この様子を図 3 に示す。ここでは、AGV の荷台に Carrier オブジェクトが対応しており、そこに転倒防止の適応部品がアタッチされている様子を表している。オブジェクトと適応部品の間はイベントで情報交換を行う。

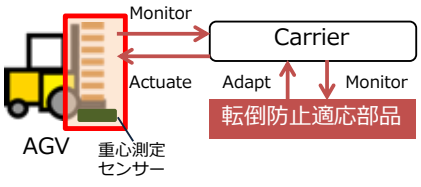

図 3 AGV の例

この例では、荷台には荷物のバランスを見るための重心測定センサーが装備されているものとする。転倒防止適応部品は荷台の重心を監視しており、一定以内の重心の遊動を許容している。荷物のバランスが悪く重心が崩れそうなとき、転倒防止適用部品が対応を試みる。例えば Carrier にアクティブサスペンションが装備されている場合は、このサスペンションの操作を行って転倒防止ができるかを分析し、必要であればサスペンションの動作を行う。

適応部品はオブジェクトに動的にアタッチすることができる。例えば、AGV が荷物を搭載したら、その荷物の種類に対応した転倒防止の適応部品をアタッチする、といった具合である。適応部品はオブジェクトから独立しているため、異なる種類のオブジェクトに同じ種類の適応部品を割り当てることも、一つのオブジェクトに複数種類の適応部品を割り当てることも可能である。たとえば、定格のコンテナを運搬するときとばら積みの荷物を運搬するときでは異なる種類の AGV または荷台を使うかもしれないが、「転倒防止適応部品」は同じ種類のものが利用可能である。

オブジェクトと適応部品の関係は、図 1 で示したそれぞれ Managed Subsystem および Managing Subsystem と同様である。本論文では適応部品の適応ロジックの詳細には立ち入らないが、たとえば MAPE-K [IBM2006]のメカニズムを使うことができる。

3.2 オブジェクトの構造化

本論文で提案する機構の動作環境を表現するオブジェクト群は、系全体で木構造を構成するようにする。

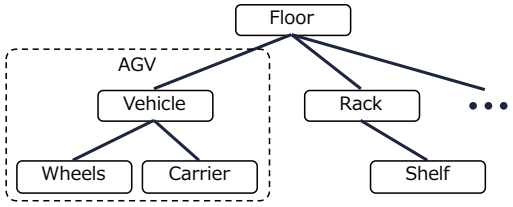

図 4 オブジェクトの木構造

一例として、2.2 節で説明した自動搬送車(AGV)が動作する空間をモデル化したものを図 4 に示す。ここでは、自動搬送車は AGV という点線内で表現されている。その中には、タイヤを表す Wheels および荷台を表す Career というオブジェクトを含んでいることがわかる。また、AGV の周囲の環境として、Floor というフロア全体を表すオブジェクトや、荷物が置かれている Rack や Shelf といったオブジェクトも存在することを示している。一般に、木構造の親子関係は実世界環境の包含関係に対応させている。例えば、より大構造のものは根に近いオブジェクト、より小構造のものは葉に近いオブジェクトとなる。

環境が木構造で表されているため、あるオブジェクトに対する適応ができなかった場合は、木構造におけるより上位のオブジェクトに対して適応を試みることができる。図 3 における転倒防止の動作は図 5①の部分に相当する。このとき、重心移動が大きく、荷台のアクティブサスペンションだけでは転倒が防止できないかもしれない。その場合は Carrier からその親である Vehicle の転倒防止機能部品に制御を渡して適応を試みる(図 5②)。Vehicle には Carrier だけでなく、Wheels という部品も含まれており、Wheels の動作(曲がる、止まるなど)を通じて転倒防止を試みることが可能である。

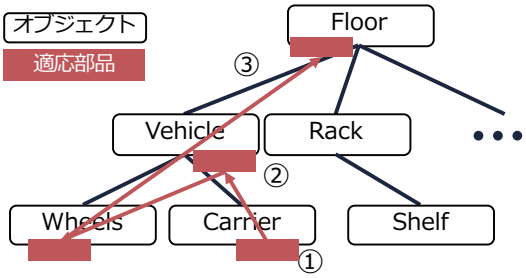

図 5 適応部品のエスカレーション

さらに、Vehicle でも転倒防止ができなかった場合は、さらに上位のオブジェクトに対応を移譲することができる(図 5③)。これは本事例の場合だと、Floor オブジェクトが該当する。すると、Floor オブジェクトは Rack オブジェクトを使った適応を行うかもしれない(可能であれば Rack を動かす等)。または転倒そのものは防げないが、対応する通路を塞いで二次災害を防いだり、対応のために人間に通知したりするかもしれない。また、動作不能になった Vehicle の部分木をシステムから切り離し、転倒している Vehicle が利用されるのを防ぐこともできる。

以上の動作をまとめると、次の通りである。

適応は木構造の葉に近い部分から開始する。これは、局所的に解決可能なものはなるべく局所的に解決する、ということを意味する。局所的に解決ができなかったものは徐々に根に向かって、すなわち当該オブジェクトが含まれる大構造のオブジェクトに対処を移譲していく。適応の方法については、次のようなパターンが考えられる。

- オブジェクトが局所的に対応する。
- 親オブジェクトに処理を移譲し、その親が直接対処するか、親が持つ別の子要素を使う。
- さらに親に移譲する。最終的には木構造の唯一の根に到達し、そこで適応ができなければシステムのエラーとなり、例えばシステムの利用者に通知される。

この仕組みにより、ある事象変化を木構造の末端(leaf)で捉えて、なるべく局所的に対応可能なものは局所的に済ませることができる。また局所的に対応ができなかったものについては、木構造を根に向かって上がっていくことによ

り、段階的に大局的な影響として捉えることができる。

3.3 初期化と Factory

適応部品はソフトウェアの動作中にオブジェクトにアタッチ、デタッチされる。そのため、どこで適応部品のライフサイクルを管理するかが問題になる。これを集中管理する方法もあるが、本論文では適応をなるべく局所化するアプローチをとっているため、ライフサイクル管理も同じく局所化することで、全体の一貫性を取る。

ここでは、そのための機構としてファクトリ(Factory)適応部品を導入する。この部品は、文字通りオブジェクト及び適応部品に対するファクトリ(生成)機能を提供するものである。Factory は全てのオブジェクトに対し定義される適応部品で、オブジェクトの生成とともにシステムによりアタッチされる。Factory は環境の変化に応じて他の適応部品をオブジェクトにアタッチしたり外したりする。

4. 議論

4.1 Web 画面における適応

本論文における適応部品の仕組みは、Ajax 実現のための機能付加部品に着想を得ている[松塚 2008]。本機構は、Web アプリケーションに適用することも可能である。

Web における環境は、Web ページであり、DOM (Document Object Model)で表されている。[松塚 2008]では、DOM 要素に対して機能付加部品をアタッチすることで様々な機能を追加することができる。たとえば、画面に表示される要素(TextBox 等)に対し、NumberOnlyLimiter という機能付加部品をアタッチすることで、入力を数値のみに制限することができる。

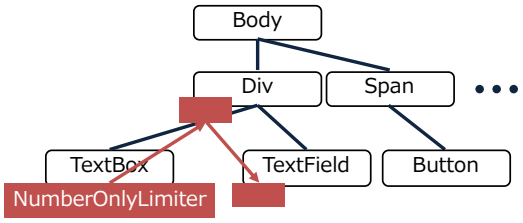

図 6 Web 画面における適応

この機能付加部品を拡張して、適応部品としての機能を持たせることができる。機能付加部品 NumberOnlyLimiter が付加された TextBox は、数値のみの入力が許された要素となるが、ここにもし数値以外の文字が入力されたときは無視される、つまり何も入力されない。

これを機能付加部品を適応部品にしたときの様子を図 6 に示す。ここでは、TextBox に数値以外の文字が入力された時は単に無視する代わりに、上位(図 6 の場合は Div)の要素に処理を移譲する。Div は自己の子要素にテキスト入力が可能なもの(この図の場合は TextField)を探し、そちらへの入力として処理を移譲することができる。

別の例として、キーボード入力を受け付けない要素(たとえば Button)にフォーカスが当たっている時にキー入力があった場合、キー入力が可能な要素に自動的にフォーカスを変更して処理を続ける、といったことも可能となる。

なお、Web 画面の場合、基本的なイベントの種類(onclick, onkeypress など)が DOM の規定で決まっている。そのため、適応部品と DOM 要素との対応関係は、当該イベントをサポートするか否かで判断することが可能である。

4.2 木構造による環境表現

本論文では、環境を木構造で表現し、木構造の要素ごとに適応部品をアタッチする、というアプローチをとっている。ここで、木構造の以下の性質を利用している。

- 根を持つ
  木構造が根を持つということは、ある適応事象がオブジェクトで対応できなかった場合に、どの方向に向かってエスカレートすべきか一意に決まることを意味する。環境構造が根を持たないと、あるオブジェクトで適応が失敗した場合、どこに向かって処理を移譲するかが決められない、または別途定義が必要となる。
- 閉路を持たない
  構造が根を持ちかつ閉路を持たないということは、適応が上位にエスカレートしていった場合に、いつか終わりがあることを意味する。環境構造が閉路を持ってしまうと、適応のエスカレーションが無限ループに陥る可能性がある。
- 連結である
  本論文で利用している木構造は連結である。これは、根が一つであることを意味する。連結でない複数の木を利用することも可能だが、環境において連結でない木の要素同士がどのような関係か、ある木の根で適応が不可能だった時に他の木に影響があるかどうかなどを定義しておく必要がある。

最後の点においては、システムの規模や性質によっては、全体を唯一の木構造で表すのが適切ではないかもしれない。そのような場合は、複数の木構造にて環境を表現することもあり得るが、本論文では範囲外とした。

4.3 自己適応ソフトウェアの性質

自己適応ソフトウェアにおいては、そうでないシステムに対し、システムが様々な性質を持つ。本節では、[Kephart2003][Salehie2009]他の論文で共通的に定義している以下の 4 種類の性質において、提案手法がどのように貢献しているかを論じる。

- Self-Configuring

変化する環境に対応できる能力である。提案手法では、適応部品が変化を認識し、オブジェクトに対して働きかけを行うことにより本能力を実装する。

- Self-Optimising
  実行状態を改善し最適化する能力である。提案手法では、Self-Configuring と同様の方法で変化を認識してオブジェクトに対して働きかけを行うことにより最適化を実装する。最適化は一般に局所的な変更のみでは終了しないため、親オブジェクトにエスカレートして複数のオブジェクトの変更が必要になる可能性がある。
- Self-Healing
  自ら問題を発見し、復旧する能力である。提案手法では、オブジェクトにアタッチされた適応部品が局所的な問題を解決するほか、木構造を活用し、あるオブジェクトで問題発見・復旧ができなかった場合に上位のオブジェクトに動作を移譲することにより、復旧を実現する。
- Self-Protecting
  内外のアタックから自己を守り、システムのセキュリティおよび一貫性を確保する能力である。提案手法では、通常の変化対応の範囲で保護を実現することは可能だが、保護に特化した能力については述べていない。今後の課題である。

さらに、これらの能力を実現するベースになるものとして、Self-awareness と Context-awareness がある[Salehie2009]ため、これらについても以下で考察する。

- Self-awareness
  ソフトウェアが自身を認識する能力である。提案手法においては、適応部品がオブジェクトを介して、または直接環境から情報を収集することにより、自己認識を行う。オブジェクトは木構造をとるため、木構造上より上位の適応部品は、より下位の適応部品から情報を受け取ることにより広範な認識を行うことができる。
- Context-awareness
  実行環境を認識する能力である。4.1 節の例では、実行されるのはブラウザのホスト環境であり、実行環境に対して全般的な情報を得ることが可能だった。しかし提案手法はこれを IoT 環境に拡張しており、実行環境については十分に分析できていない。今後の課題である。

4.4 関連研究

本節では関連研究について述べる。

4.4.1 Architectural Blueprint

自己適応研究でよく知られているモデルに、IBM による Architectural Blueprint がある[IBM2006]。この研究では、アーキテクチャの最下層にサーバやストレージ、ネットワークといった物理･論理リソースがあり､それを Manageability Endpoints (または Touchpoint)と呼ばれる層でインタフェース化し、センスおよびアクチュエートを可能とする。その上で Touchpoint Autonomic Managers および Orchestrating Autonomic Managers という MAPE-K (Monitor, Analysis, Plan, Execute, Knowledge)が含まれるコンポーネントで適応を行う。

より具体的には、このアーキテクチャは Touchpoint Autonomic Managers とそれ以下の層でリソースに対する適応を実現し、その上の Orchestrating Autonomic Managers が Touchpoint Autonomic Managers を束ねたより高位の適応を実現するという 2 階層になっている。IBM の提案するアーキテクチャはコンピューティング環境の管理が目的だったこともあり、リソースそのものの構成管理は CMDB (Configuration Management Database)などの手法が確立している。そのため、本提案のようにリソースの管理と適応機構を組み合わせる必要がないと考えられる。

一方、本論文で提案するアーキテクチャにおいては、リソースの出現位置が最下層に固定されておらず、Managing Subsystem (Autonomic Manager に相当する)とセットで木構造のすべての階層に出現する。また、階層の数にも制限がない。この構造をとることにより、IoT など細粒度のものからシステム全体に至るまで、対象とするリソースの粒度に合わせた適応を可能にしている。対応関係を図 7 に示す。また、リソース自体の構造と適応の階層が対応しているため、環境にデバイスが動的に加えられたり削除されたりした場合に、それに対する適応コンポーネントをデバイスのオブジェクトと一緒に追加したり削除したりすることが可能である。そのため、環境の構成が複雑で動的に変化する IoT などのシステムにより適していると考える。

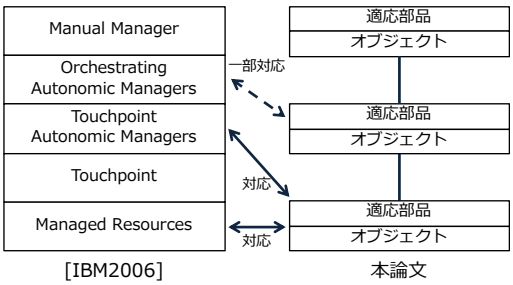

図 7 Architectural Blueprint と提案手法の対比

4.4.2 DEECo

前述した DEECo [Bures13]では、自律コンポーネント同士の関係が、データ交換を制御する「アンサンブル」によって組み替えられながら実行を行うコンポーネントシステムを提案している。論文では、電気自動車のナビゲーションを例に説明している

アンサンブルは動的に構成される一種のコンポーネントのグループで、その構成は「同じ場所を目指している車のアンサンブル」や「同じ信号を待っている車のアンサンブル」など状況に応じて刻々と変化する。一つのアンサンブルの中には一つの Coordinator と 0 以上の(通常は複数の)Member が含まれ、情報交換を行う際の役割分担を行なっている。あるコンポーネントは複数のアンサンブルのメンバーになることが可能である。この様子を図 8(a)に示す。ここでは、点線で囲まれた 2 つのアンサンブルを表しており、各アンサンブルにおいて C_C が Coordinator、C_M が Member である。いくつかのコンポーネントは 2 つのアンサンブルに属しており、同じコンポーネントでも属するアンサンブルによって役割が変化する。

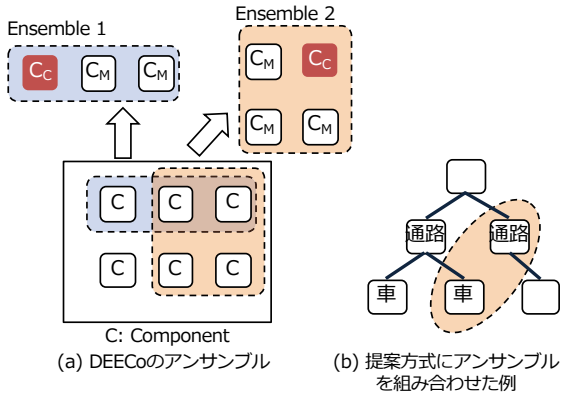

図 8 DEECo と提案手法の対比

本論文での提案手法では、適応メカニズムを構造化しており局所的な対処から始め徐々に大局的な適応を行うという DEECo にはない特徴を持っている。一方、オブジェクトが同時に複数のグループに所属することを可能にする DEECo に比べると構造の柔軟性は低い。木構造であっても、データ交換を親子関係上離れた場所にあるオブジェクトと行いたい、という要件は容易に想像がつくため、DEECo が提案する動的グループは一考の余地があると考える。たとえば図 8(b)のように、ある通路にいる車が、走行先の通路の状況を確認する、といったことである。

5. おわりに

本論文では、実世界に対応するオブジェクトを木構造で構造化し、それぞれのオブジェクトに適応部品を関連づけることで自己適応を実現するソフトウェアを提案した。オブジェクト群を木構造にすることにより、あるオブジェクトに関連づけられた適応部品で自己適応が不可能だった場合に上位のオブジェクトに関連づけられた適応部品で適応動作を行うことが可能となる。言い換えると、局所的な適応が不可能な場合もより大構造での適応をすることが可能となる。

IoT を用いたアプリケーションなど、多数のデバイスやコンポーネントから構成されるソフトウェアでは、トップダウンで環境の全てを把握し管理することが大きなコストになる。そのため、局所的に対応可能なことはできるだけ局所的に済ませることが必要であると考える。一方で、局所的に対応が済まない事象への対応も行う必要があり、本提案はそのような環境に適しているのではないかと考えている。

本提案は、実行環境に非依存な形での設計となっているが、今後、実装を行い検証を行う予定である。その際、4.3 節で議論した Context-awareness という性質で表されるように、実行環境が自己適応に与える影響は大きいため、実行環境をどのように抽象化して系に持ち込むかは今後の課題である。

参考文献

[Weyns2017] D. Weyns, “Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges,” in Handbook of Software Engineering, K. Kyo Chul Kang and S. Cha, Eds. Springer, 2017.

[Musil2017] Musil A., Musil J., Weyns D., Bures T., Muccini H., Sharaf M. (2017) Patterns for Self-Adaptation in Cyber-Physical Systems. In: Biffl S., Lüder A., Gerhard D. (eds) Multi-Disciplinary Engineering for Cyber-Physical Production Systems. Springer, Cham

[Baresi2010] Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER 2010, pp. 17–22. ACM, New York (2010)

[Lwakatare2015] L. E. Lwakatare, P. Kuvaja, and M. Oivo, “Dimensions of DevOps,” in 16th International Conference on Agile Software Development (XP). Springer International Publishing, 2015, pp. 212–217.

[Lemos2009] R. de Lemos, H. Giese, H. Müller, and M. Shaw, "Software Engineering for Self-Adaptive Systems: A Research Roadmap.," In Software Engineering for Self-Adaptive Systems, Lecture Notes In Computer Science, Vol. 5525., 2009.

[Macias2013] Frank D. Macías-Escrivá, Rodolfo Haber, Raul del Toro, Vicente Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications. Expert Systems with Applications, 2013. 40(18): p. 7267-7279.

[Salehie2009] M. Salehie and L. Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. ACM Transactions on Autonomous and Adaptive Systems (TAAS), Volume 4 Issue 2, May 2009, Article No. 14.

[Weyns2013] Weyns D. et al. On Patterns for Decentralized Control in Self-Adaptive Systems. In: de Lemos R., Giese H., Müller H.A., Shaw M. (eds) Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science, vol 7475. Springer, Berlin, Heidelberg

[Le-Anh2006] Le-Anh, T., & De Koster, R. (2006). A review of design and control of automated guided vehicle systems. In L. Peccati, R. Slowinski, & J. Teghem (Vol. Eds.), European Journal of Operational Research: Vol. 171, (pp. 1–23).

[Negri2017] Negri E, Fumagalli L, Macchi M (2017) A review of the roles of digital twin in CPS-based production systems. Procedia Manuf 11:939–948.

[Madni2019] Madni AM, Madni CC, Lucero SD. Leveraging Digital Twin Technology in Model-Based Systems Engineering. Systems. 2019; 7(1):7.
[IBM2006] IBM: An architectural blueprint for autonomic computing. Tech. rep. IBM (January 2006)
[松塚 2008] 松塚貴英: 業務アプリケーションに適用する Ajax フレームワーク, 情報処理学会論文誌,Vol.49 No.7,2008.
[Kephart2003] J. O. Kephart, D. M. Chess, The Vision of Autonomic Computing, IEEE Computer, 36 (1), pp. 41-50, Jan. 2003
[Bures13] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECO: an ensemble-based component system, CBSE '13 Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering, Pages 81-90, 2013.