情報処理学会論文誌 Vol. 50 No. 10 2500–2513 (Oct. 2009)

ゴール指向要求分析を用いた self-adaptive システムの構築

中川博之[†1] 大須賀昭彦[†1] 本位田真一[†2,†3]

近年，環境の変化に応じて柔軟に振舞いを変えることのできる self-adaptive システムに対する期待が高まっている．しかし，その統一的な開発方法はいまだ確立されておらず，システムに対する要求に応じた柔軟なシステムアーキテクチャの構築が研究課題となっている．そこで本研究では，要求分析モデルを利用した self-adaptive システムのアーキテクチャモデル構築法を提案し，あわせて，本モデルを利用したマルチスレッド・プログラミングによるプログラム実装方針を示す．本提案手法により，要求に柔軟に対応できる self-adaptive システムの構築が可能となる．

Constructing Self-adaptive Systems Using Goal-oriented Requirements Analysis

Hiroyuki Nakagawa,[†1] Akihiko Ohsuga[†1] and Shinichi Honiden[†2,†3]

Self-adaptive systems have recently attracted attention as flexible software because they can change their own behaviors to react to changes in their environments. This paper describes our approach to developing self-adaptive systems utilizing a requirements model to build the system architecture. This paper also discusses the implementation style using the acquired architecture model, and our evaluation of the effectiveness of our development process through a case study.

†1 電気通信大学
The University of Electro-Communications
†2 国立情報学研究所
National Institute of Infomatics
†3 東京大学
The University of Tokyo

1. はじめに

近年，ソフトウェアの活躍する場面が広がり，あらゆる環境下で適切に動作するソフトウェアが求められるようになっている．そのような背景から，環境に適応し，みずからの動作を変化させる self-adaptive システム[1] の実現に対する期待が高まっている．self-adaptive システムとは，自身の振舞いを評価し，目的を管理することで，予期しない環境の変化に対しても振舞いを切り替えたり，場合によっては代替となる目的へと変更したりすることで環境への適応を実現するシステムである．

しかし self-adaptive システムの実現のためには，ソフトウェア工学の観点からもまだまだ解決すべき課題は多い[2]．特に文献 3) においても指摘されているように，ソフトウェアに対する要求とアーキテクチャとは相関関係にあるといえるが，self-adaptive システムは要求から導出されるシステムの目的に合致した振舞いの動的な切替えを期待されていることから，従来のソフトウェア開発よりも要求分析結果を反映したアーキテクチャモデルを構築する必要がある．現在，self-adaptive システムのアーキテクチャモデルとして参照されることの多い Kramer らの 3 層アーキテクチャモデル[4] においても，振舞いを実現する最下層のコンポーネント制御層と，目的に応じた行動プランを決定する最上層のゴール管理層との連携に関しては，文献 5) などの報告があるものの，いまだ十分に議論されているとはいえない．

そこで本研究では，要求記述を利用した self-adaptive システムのアーキテクチャモデル構築手法を提案する．要求記述としては，階層構造を持つ記述法であるゴール指向要求分析モデルを利用し，本論文では特に，要求間の衝突を明示的に記述することのできる KAOS[6),7)] をゴール指向要求分析法として用いる．また，本論文では，構築されたアーキテクチャモデルに対するマルチスレッド・プログラミングによる実装方針も示す．本提案により，目的に応じた振舞いの変化が可能な self-adaptive システムのアーキテクチャモデルを要求記述から形式的に構築できるようになるとともに，構築されたアーキテクチャモデルを実装する 1 つの実現手段が提供される．

本論文の構成は以下のとおりである．まず 2 章で，self-adaptive システムが持つべき特性を定義し，その特性を実現するためにシステムアーキテクチャおよびソフトウェア開発プロセスに求められる要件を抽出する．続く 3 章では，提案手法で利用する要求分析法 KAOS について説明する．4 章では，2 章で抽出した要件に基づいて，本論文で提案する self-adaptive システムの構築手法について述べ，5 章では提案手法を利用した self-adaptive システムの

構築実験結果から本手法の実現可能性と有効性を考察する．最後に 6 章で提案手法の自動化の可能性と適用範囲について議論し，7 章でまとめと今後の課題について述べる．

2. self-adaptive システム

2.1 self-adaptive システムに対する要件

self-adaptive とは，みずからの目的を管理し，環境や内部状態の変化に応じてみずからの振舞いを切り替えることのできる特性であり，QoS が要求されるアプリケーションサーバやネットワークシステム，ユビキタス環境下のシステム[8] に対して特に必要とされる特性である．self-adaptive システムに対しては，文献 1)，9) などで様々な定義がなされているが，本研究では文献 1) の定義をもとに以下の 2 つを self-adaptive システムが持つべき特性として定義する．

［定義 1］self-adaptive システムが持つべき特性

- **特性 1：**自身の振舞いや目的・内部状態を評価することができる．
- **特性 2：**振舞いを動的に変化させることができる．

これらの特性を満たすシステムを構築するには，まずシステムアーキテクチャの観点からは，少なくとも自身の状態を評価することのできるアーキテクチャでなければならない．また，振舞いを動的に変化させるためには，各振舞いを実現するモジュールを動的に切り替えることのできるアーキテクチャである必要がある．一方で開発プロセスの観点からは，特性 1 に対しては，特に self-adaptive システムでは単に障害を回避するような振舞いの変化だけでなく，状況によっては達成すべき目的の妥協も考慮した振舞いの変更が期待されることから，システム開発を通じて，目的と振舞いとが対応付けられている必要があると考えられる．また，特性 2 に対しては，振舞い間には共通のリソースを利用するという観点で競合が発生する場合が多く，振舞いの変化を設計・実装するためには振舞い間で発生する競合の回避を特に考慮する必要がある．そこで本研究では，定義 1 の 2 つの特性を満たすために，システムアーキテクチャおよび開発プロセスに求められる要件として以下を定義する．

［定義 2］self-adaptive システムのアーキテクチャおよび開発プロセスに求められる要件

- **要件 1：**自身の状態を評価することができるアーキテクチャである．
- **要件 2：**振舞いモジュールの動的変更が可能なアーキテクチャである．
- **要件 3：**目的と振舞いとを対応付けることができる．
- **要件 4：**振舞い変化時の衝突を考慮することができる．

なお，self-adaptive システムの典型的なフィードバックプロセス[10] として収集（Collect），分析（Analyze），決定（Decide），実行（Act）の 4 つのプロセスが知られているが，収集に対しては要件 1 が，分析には要件 1 と要件 3，決定には要件 3 と要件 4，実行には要件 2 と要件 4 が対応し，本論文で定義した要件はすべてのプロセスを被覆しているといえる．

2.2 関 連 研 究

self-*システムやそのアーキテクチャに関しては数多くの研究がなされているが，定義 2 の要件 1，2 を満たす可能性を持つアーキテクチャとして，1 章でも言及した Kramer らの 3 層アーキテクチャモデル[4] があげられる．このアーキテクチャはロボティクスの分野で提唱された Gat の 3 層モデル[11] を応用したものであり，行動プランを決定するゴール管理層，振舞いの変化を司る変化管理層，コンポーネントの状態を管理するコンポーネント制御層の 3 層から構成される．コンポーネント制御層は複数コンポーネントの接続により構成され，各コンポーネントは内部状態を可視化するための mode と呼ばれる変数を提供することで，外部からのコンポーネント状態の参照を可能としているとともに，変化管理層がコンポーネントの接続を変化させることにより振舞いを変更することから，このアーキテクチャモデルは要件 1，2 を満たす可能性を持つといえる．しかしながら，自己を評価するには状態を参照する側とされる側との関係を考慮したコンポーネント接続を決定する必要があり，特に，複数の目的や振舞いを持つ self-adaptive システムにおいては，必要なコンポーネントを抽出・決定し，参照・非参照の制約を満たす接続関係を決定することは通常容易ではない．また，この 3 層アーキテクチャでは，現在の状態を表す mode は最下層のコンポーネント制御層にあるのに対し，システムの目的は最上位層のゴール管理層で管理されており，現段階ではそれらの対応関係が明示されていないことから，振舞いや状態と目的との対応関係を関連付けることも困難である．同様に衝突への対応についても，衝突が発生する可能性のある箇所を 3 層それぞれで特定し，衝突を回避するためのコンポーネント接続関係や層間の関係を定義することも容易であるとはいえない．以上の観点から，3 層アーキテクチャを用いたとしても，アーキテクチャに関する要件 1，2 を満たす可能性はあるが，その実現は難しく，また，要件 3，4 を満たすことも難しいといえる．

開発プロセスの観点からは，要件 3 を満たす開発プロセスとして，エージェント技術の分野で確立された BDI（Belief，Desire，Intention）モデル[12] を self-adaptive システムに適用した Morandini らの手法[13] があげられる．エージェントは自身の目的を達成するために自律的に動作・意思決定するソフトウェアであり，目的管理のために BDI モデルのような心的モデルが提案されている点や，自律的な動作を実現するために複数のプロセスを並行実行する機構を持つフレームワークが提案されている点などから，近年，self-adaptive

システムの 1 つの実現手段として期待されている．Morandini らの手法では，要求分析結果として得られたツリー上のゴールモデルに記述されたゴールを文献 14) で定義される 3 種類のゴールに分類し，それぞれを実現するためのコードをプランとして実装することで self-adaptive システムを構築する．ゴールの実行を制御することで振舞いを実現することから，振舞い実行にゴールモデル上の分析結果を利用しているといえ，要件 3 の「目的と振舞いとを対応付けることができる」を満たす構築手法といえる．ただし，この手法ではゴール実行のプランを構成要素として self-adaptive システムを実装するが，プランからはシステムアーキテクチャ上での振舞いモジュールの構成を把握することは困難であり，モジュール間に内在する競合を判断することが難しい．したがって，要件 4 を満たしているとはいえない．また，要件 1，2 に対してはアーキテクチャが明示的に対応しているわけではなく，開発者が要件を満たすための機構を独自に構築する必要がある．さらに，この手法で用いるゴールの分類法や BDI モデルの概念はエージェント分野の固有の理論の上に成り立っているもので，モデリングが容易ではなく，また，BDI モデルを前提としたフレームワーク上でのプログラム実装に限定されるため，手法の汎用性も高いとはいえない．

同じく要件 3 を満たす開発プロセスとしては，self-adaptive システムに特化されない汎用的なアプローチとして，要求モデルを利用したアーキテクチャモデル構築手法[15] なども知られている．この手法は，ゴール指向要求分析法 KAOS により構築された要求モデルからアーキテクチャモデルを構築する指針を提供している．しかしながらその構築法は，データフローに基づいてアーキテクチャを構築する手法であり，状況に応じて振舞いを変化させるという意味でシナリオの特定が困難な self-adaptive システムの構築には適しているとはいえない．したがって，振舞いの変化自体を考慮することができないことから，要件 4 を満たしてはいない．

2.3　本研究のアプローチ

2.2 節で述べた関連研究の現状をもとに，定義 2 で示した 4 つの要件から，追求すべき self-adaptive システムのアーキテクチャおよび開発プロセスについて議論する．まず，要件 1 および要件 2 のアーキテクチャに求められる要件を満たすためには，2.2 節であげた 3 層アーキテクチャの最下層に位置するコンポーネント制御層のような複数コンポーネントの接続により形成されるアーキテクチャが有効であると考えられる．ただし，その場合問題となるのは，状態を参照する側のコンポーネントと参照される側との適切な接続関係の定義である．また，要件 3 を満たすためには，コンポーネントの振舞いと self-adaptive システムの目的とが対応付けられている必要がある．加えて，要件 4 を満たすためには，衝突が発生するモジュールが特定でき，さらに振舞い間だけでなくシステムの目的間における衝突も検出できる必要がある．

そこで本研究では，階層構造を持つ要求記述法といえるゴール指向要求分析法[16] に着目し，ゴール間の階層構造を利用したアーキテクチャモデル構築法を提案する．提案手法では，各コンポーネントに達成すべきゴールあるいは要件を対応付け，各ゴール・要件の達成によりシステム全体の目的が達成されると想定した，コンポーネントにより形成されるアーキテクチャを構築する．これにより，要求分析により獲得されたシステムの目的とコンポーネントが実現する振舞いとを対応付けることが可能となるとともに，コンポーネント間の接続関係を目的の対応関係から決定することが可能となる．本論文では特に，階層構造を実現するためのゴール指向要求分析法として，形式的なゴール記述とゴール間の衝突の明示的な記述が可能である KAOS[6),7)] を用いる．また，得られたアーキテクチャモデルをもとに self-adaptive システムを構築する手段として，すでに多くの言語で提供されているマルチスレッド・プログラミングによる構築指針を示す．

3.　KAOS による要求分析

KAOS はゴール指向要求分析法の一種であり，システムに対する要求（ゴールと呼ぶ）を明示的に記述し，それを詳細化することで，識別したゴールを達成するための要件を系統的に導出する手法である．KAOS 分析ではまずゴールモデルにおいて，システムに対する要求（ゴール）を，すべての達成が必要なサブゴールに分解する AND–分解と，いずれかの達成が必要なサブゴールに分解する OR–分解により，単一アクタ[⋆1]が実現できる大きさにまで細分化する．十分に細分化されたゴールは，構築すべきシステムが達成すべき要件（requirement）とユーザ・他システムといった外部環境のアクタが達成すべき期待（expectation）とに分類される．**図 1** は KAOS ゴールモデルの記述例であり，この例では，要件を割り当てられるエージェント 1 は構築すべきシステムを，期待を割り当てられるエージェント 2 は外部環境のアクタを示したものとなる．

KAOS はゴールモデルのほかにも，それぞれの要件・期待を実現するオペレーションを定義するためのオペレーションモデルや，システムを取り巻くドメイン上に存在するエンティティを定義するためのオブジェクトモデルなどの図形表現モデルを提供しているが，その一方で，各モデル要素を形式的に定義するための論理表現も持つ．この論理表現には時相論理

⋆1 KAOS ではアクタのことをエージェントと呼んでいる．

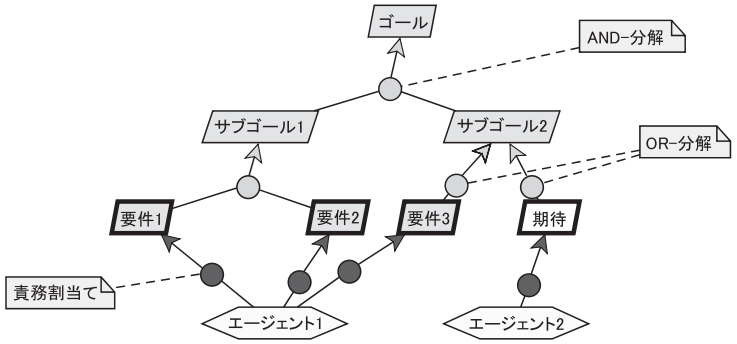

図 1　KAOS ゴールモデルの記述例
Fig. 1　Legend of KAOS goal model.

を用いた記述が可能であり，KAOS ではすでに，対象とするドメインに依存しない記述分類手段や分析手段としていくつかのパターン[7),17),18)] が提供されている．たとえば Darimont らは，文献 17) において，ゴール記述に関する 4 種類のゴールパターンを定義している．

- **Achieve goals：**いずれは満たされなければならない状態を記述したゴール（$P \Rightarrow \Diamond Q$）
- **Cease goals：**いずれは解消しなければならない状態を記述したゴール（$P \Rightarrow \Diamond \neg Q$）
- **Maintain goals：**つねに満たされなければならない状態を記述したゴール（$P \Rightarrow \Box Q$）
- **Avoid goals：**つねに避けなければならない状態を記述したゴール（$P \Rightarrow \Box \neg Q$）

ここで，$\Diamond$ は将来のいずれかの時点で真となることを，$\Box$ は今後つねに真であることを表す時相演算子であり，P には各ゴールに対する事前条件が，Q には各ゴールが満たすべき状態が対応する．KAOS はこのように，図形表現と論理表現の 2 つの記述文法を持つことで要求の系統的な分析を実現する要求分析法であるといえ，提案手法では，この系統的な分析により構築されるゴール間の階層構造をアーキテクチャモデル構築に利用する．

4. KAOS モデルを利用した self-adaptive システムの構築

本研究で提案する self-adaptive システム構築法は以下の 3 つのプロセスにより構成される．

1. **ゴールの記述：**開発者はまず，self-adaptive システムに対する要求をゴール指向要求分析法 KAOS により分析する．この分析は通常の KAOS 分析に従うが，提案手法では，得られた KAOS モデルに本論文で説明する分析を追加することにより，self-adaptive システムのアーキテクチャモデルを生成可能なモデルへと詳細化する．
2. **システムアーキテクチャおよびコンポーネント仕様の決定：**続いて，前プロセスで構築した KAOS モデルを，提案手法に従ってコンポーネントの階層モデルへと変換する．得られたモデルは self-adaptive システムの構造を示すものであり，アーキテクチャモデルに該当する．また，KAOS モデルでの分析結果をもとにコンポーネント実装のためのコンポーネント仕様を決定する．
3. **コンポーネントの実装：**開発者は前プロセスで得られたアーキテクチャモデルとコンポーネント仕様をもとに，各コンポーネントをマルチスレッド・プログラミングをサポートするプログラム言語により実装する．

本章では以降，文献 13) でも self-adaptive システムの例として取り上げている清掃ロボットの構築を例として，提案する self-adaptive システムの構築法について説明する．

4.1　ゴールの記述

提案手法では，形式手法に基づいた論理的なゴール分析が可能である点，衝突を明示的に記述できる点から，ゴール指向要求分析法の中でも特に KAOS を分析・記述法として用いる．提案手法ではさらに，self-adaptive システムに有効なアーキテクチャモデルを導出可能な要求モデルを獲得するために，従来の KAOS 分析に対して以降の各項で述べる分析項目を追加する．

4.1.1　システムの責任範囲

KAOS では，ゴールモデルによりゴールがシステムやユーザなど個々のアクタに責務が割り当てられる粒度にまで分解され，通常はゴールツリーの下層に位置する十分に分解された要件や期待を各アクタに割り当てる．**図 2** は清掃ロボットに対する KAOS モデルの例であるが，従来の KAOS 分析では，要件として記述されている “SweepDust” や “PickUpDust”，“ApproachDust” などが清掃ロボットに割り当てられることとなる．

しかし self-adaptive システムを考えた場合，システムが遭遇するすべての局面や環境を事前に想定することはできず，個々の局面における責務を抽出し，それらを達成するための機能のみを個別に実装するだけでは不十分である．self-adaptive システムにおいては，想定外の環境変化にも対応できるように，割り当てられた機能（要件）だけでなく，機能を実現する振舞いの実行順序や代替となる振舞いの情報を持つという観点から，機能を導出するもととなったゴールとその階層構造も設計時に考慮しておくことが有益であると考えられる．そこで提案手法では，KAOS におけるエージェントの責務を拡張し，以下の定義を定める．

［定義 3］self-adaptive システムに対する責務割当て：ゴールモデル上のすべてのゴール

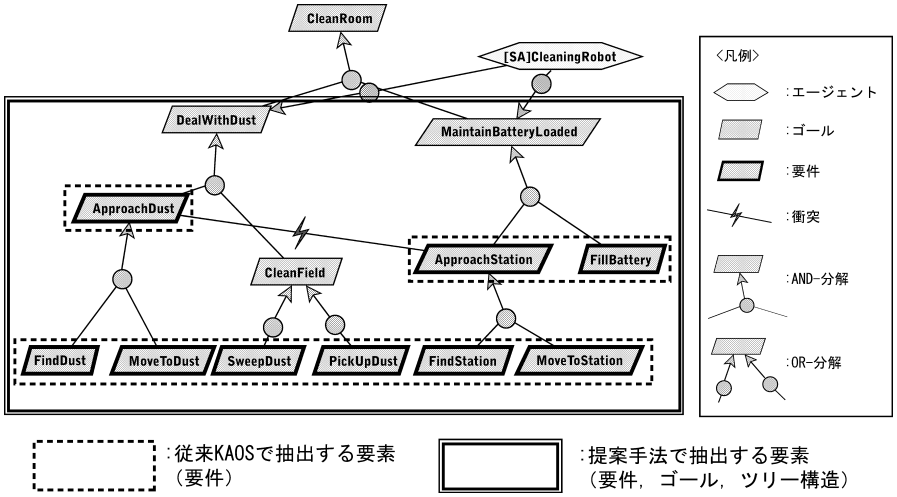

図 2　清掃ロボットに対する KAOS モデル
Fig. 2　KAOS goal model for cleaning robot.

および要件の集合を G とする．self-adaptive システムに該当するエージェントに対しての，次の条件を満たす $n_1, \ldots, n_m \in G$ との関係を，self-adaptive システムに対する責務割当てと定義する．ただし，n_i を根とするサブツリーに包含されるゴールおよび要件の集合を N_i（$\subseteq G$）とする．

- **条件 1．**ゴールモデル上のすべての要件が $\bigcup_{1 \le i \le m} N_i$ に包含されている．
- **条件 2．**いずれの N_i を n_i の各サブゴール $n_{i1}, \ldots, n_{il}$ に対する $N_{i1}, \ldots, N_{il}$ の和集合 $\bigcup_{1 \le j \le l} N_{ij}$ に置き換えても，$(\bigcup N_k (k \neq i)) \bigcup (\bigcup_{1 \le j \le l} N_{ij})$ に包含されない要件が存在する．
- **条件 3．**いずれの N_i も，N_k（$k \neq i$）の真部分集合ではない．

ここで条件 1 は，責務割当てにより抽出されるゴールおよび要件の集合に，システムが満たすべき機能，つまり要件がすべて包含されていることを保証するためのものであり，条件 2，条件 3 は，抽出される集合が最小であり，$n_1, \ldots, n_m$ が冗長でないことを保証するためのものである．本定義から，たとえば図 2 のゴールモデルに対しては，図中に示されるとおり，ゴール “DealWithDust” とゴール “MaintainBatteryLoaded” がエージェントに割り当てられる．従来の KAOS では要件のみがエージェントに割り当てられていたが，本定義により，各要件が導出されたゴール群と，要件やゴール間の階層構造が KAOS モデルから獲得できることとなる．提案手法では獲得された各要素に対して，要件を各振舞い実現のためのコンポーネントに，ゴールを振舞い制御のためのコンポーネントに対応付け，階層構造をコンポーネント間の接続関係定義のために利用する．

Goal Maintain[MaintainBatteryLoaded]
　FormalDef ∀ bt:Battery, cl:CleaningRobot, st:BatteryStation
　　Has(cl,bt) ∧ bt.Level <‘Low’ ∧ Reachable(cl,st) ⇒ □ HeadedForCharge(cl,bt,st)

Goal Achieve[ApproachDust]
　FormalDef ∀ d:Dust, cl:CleaningRobot
　　LaidOn(d) ∧ ¬InRange(cl,d) ∧ Reachable(cl,d) ⇒ ◇ InRange (cl,d)

Goal Achieve[ApproachStation]
　FormalDef ∀ st:BatteryStation, cl:CleaningRobot, bt:Battery
　　LaidOn(st) ∧ ¬InRange(cl,st) ∧ Reachable(cl,st) ∧ Has(cl,bt) ∧ bt.Level<‘Low’
　　⇒ ◇ InRange (cl,st)

図 3　ゴール記述の例
Fig. 3　Examples of goal description.

4.1.2　ゴールパターンの決定

提案手法ではコンポーネントの動作タイプを決定するために，self-adaptive システムが達成する目標に応じて，4.1.1 項で抽出した各ゴールと要件に対してゴールパターンを決定する．ただし，Cease goals と Avoid goals は状態 Q の記述により Achieve goals と Maintain goals により代替できるため，本手法では用いない．**図 3** はゴールパターンを含めた KAOS のゴール記述例である．提案手法ではゴール記述を用いることで，ゴールの特性と期待される状態遷移を明確化し，次項で述べる衝突の分析に利用するとともに，コンポーネントの仕様構築時にも参照する．

4.1.3　衝突の分析

self-adaptive システムでは振舞いを変化させる必要があるため，提案手法ではシステム内の衝突（conflict）に対して特に注目する．衝突とはゴール間で生じる矛盾を表現したものであり，両ゴールの満たすべき状態を同時に達成することができない場合に定義するゴール間関係である．提案手法では以下の手順に従って衝突への対応を検討する．

［定義 4］衝突への対応手順

(1)　［ゴール記述の構築］4.1.1 項で抽出した各ゴールと要件に対してゴール記述を完成させ，達成すべき状態 Q を決定する．

(2)　［衝突の検出］ゴール，要件間で状態 Q が相反するものを抽出し，衝突関係にあるものを同定する．この衝突の検出には，必要に応じて文献 18) などの形式的な手段を用いる．

(3)　［優先順位の決定］(2) で検出された衝突に対して，衝突に関与する要求・ゴールが表現する要求の内容や重要度をもとに，どちらのゴール（要件）を優先して満たすかを決定する．ゴール記述の変更により衝突が解消できる場合は，ゴール記述も変更する．

(4)　［責務の同定・実装］優先順位の高いゴール・要件，あるいはその親ゴールに対応するコンポーネントを衝突に対して責務を持つコンポーネントと同定し，実装段階において，競合を回避するための実行制御を同コンポーネントに実装する．

たとえば図 3 の例では，ゴール “ApproachDust” が達成すべき状態 InRange (cl,d) とゴール “ApproachStation” が達成すべき状態 InRange (cl,st) とはごみ d とバッテリステーション st がともに清掃ロボット cl の手の届く範囲に配置されていない限り同時に満たすことができず，衝突の関係にある．この衝突の検出は形式的にも可能である．たとえば文献 18) では，一方のゴール記述の否定をとり，他ゴール記述とドメイン知識を利用して後ろ向き推論（backward-chaining）により衝突が発生する状態を検出するが，本例では清掃ロボット cl が同時に手の届く範囲にごみとバッテリステーションが配置されていない場合があるというドメイン知識[⋆1]を追加することで，バッテリ残量が少ない状況で衝突が発生することが検出できる．

続いて，得られた衝突関係に対して優先順位を決定する．優先順位とは，同時に満たそうとすると衝突が発生する，つまり衝突関係にあるゴール対に対して，どちらのゴールを優先して満たすかを示したものであり，ゴールが表現する要求の内容やその重要度をもとに意味的に判断する．本例の衝突に関しては，バッテリが切れると清掃ロボットが停止してしまうことから，ごみの清掃よりもバッテリ補給を優先すべきと考えられ，したがってゴール “ApproachDust” よりもゴール “ApproachStation” を優先させる．この例では，バッテリ残量が少ない場合にごみへの移動を発動しない，つまりゴール “ApproachDust” に対するゴール記述の条件部に ¬(bt.Level<‘Low’) を追加することで，衝突状態においてバッテリステーションへの移動を優先させる．

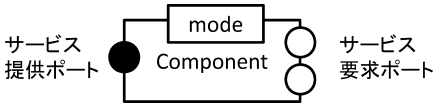

図 4　コンポーネントモデル[19)]
Fig. 4　Component model [19)].

この衝突に関する分析結果は，コンポーネントの実装の段階で，振舞いを変化させる際にどちらの振舞いを優先させ，また，どのコンポーネントでその制御を実現するかを決定するために用いる．

4.2　システムアーキテクチャおよびコンポーネント仕様の決定

KAOS モデルに対して 4.1 節で定めた分析が完了すると，提案手法では，得られた KAOS モデルからアーキテクチャモデルを構築する．アーキテクチャモデルの構成要素となるコンポーネントには，2.3 節で言及した 3 層アーキテクチャのコンポーネント制御層においても利用されているコンポーネントモデル[19)] を用いる（**図 4**）．本モデルの特徴は，コンポーネントの外部から内部状態を可視化するための mode を提供するとともに，各コンポーネントがサービス提供ポートとサービス要求ポートを持ち，それらを接続することによりシステムアーキテクチャを表現するところにある．

提案手法では 4.1.1 項で述べたゴール割当ての定義に従って，self-adaptive システムに割り当てられたゴールと要件の集合を抽出し，各ゴール・要件に対してそれぞれ 1 つのコンポーネントを割り当てる．その後，親ゴールと子ゴールそれぞれに対応するコンポーネント間のポートを接続することで，コンポーネントの階層関係を定義する．この際，KAOS モデルでエージェントとして表現される self-adaptive システム自身もルートのコンポーネントとして抽出し，KAOS モデル上で明示的に割り当てられたゴールに対応するコンポーネントと接続する．コンポーネントの接続形態は KAOS モデルでの分解方法から決定し，AND–分解に対しては親ゴールに対応するコンポーネントが子ゴールに対応するコンポーネントのサービスを利用する形態で接続し，OR–分解に対しては親コンポーネントが子コンポーネントを内部に包含し，サービス提供ポートを統一した形態で接続する．付録 A.1 に，提案手法における KAOS モデルからアーキテクチャモデルへの変換アルゴリズムを示す．**図 5** は，図 2 の KAOS モデルをもとに生成された清掃ロボットに対するアーキテクチャモデルである．図 5 の例では，たとえばコンポーネント “DealWithDust” とその子コンポー

⋆1 この例では，ドメイン知識として
∃ st:BatteryStation, d:Dust, cl:CleaningRobot
LaidOn(st) ∧ LaidOn(d) ∧ InRange(cl,d) ⇒ ¬InRange(cl,st)
を追加する．

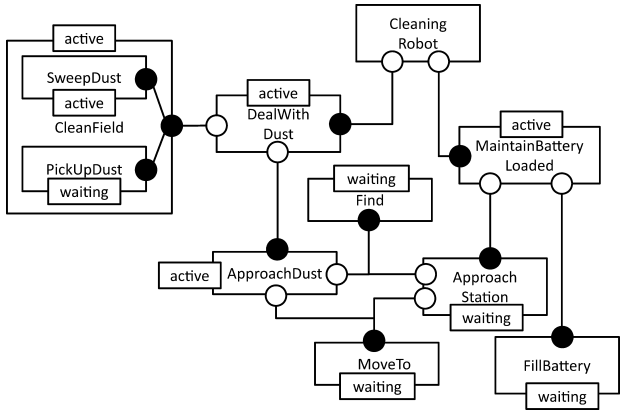

図 5　清掃ロボットに対するアーキテクチャモデル
Fig. 5　Architecture model for cleaning robot.

ネントとは AND–分解に対応した接続に該当し，コンポーネント “CleanField” とその子コンポーネントとは OR–分解に対応した接続に該当する．このような変換により，KAOS モデルの階層構造を反映したアーキテクチャモデルの構築が可能となる．

また，アーキテクチャモデル構築にあたっては，類似の達成状態を持つゴールをコンポーネント化の段階で 1 つのコンポーネントに集約する．たとえば，図 2 の例では，ゴール “FindDust” と “FindStation” は図 5 におけるコンポーネント “Find” に，ゴール “MoveToDust” と “MoveToStation” はコンポーネント “MoveTo” にそれぞれ集約されている．

提案手法ではさらに，コンポーネントの状態と目的に対する達成状況の可視化を目的として，コンポーネントの内部状態を汎用的に定義する．図 4 における mode はコンポーネントの内部状態を可視化したものであるが，提案手法では**図 6** に示すコンポーネントのライフサイクルを定義し，mode がとりうる値として，*waiting*，*active*，*achieved*，*not achieved* の 4 種類を定める．ここで，*waiting* はコンポーネントのサービスが待機状態であることを表し，*active* はコンポーネントが稼働中であり，サービスが提供されている状態であることを表す．一方，*achieved* と *not achieved* は Achieve goals に対応するコンポーネントのみがとりうる状態であり，コンポーネントの動作の結果，各ゴールあるいは要件が目標とする状態に到達したかどうかを表現するものである．図 6 中の `activate()` と `passivate()` はそれぞれコンポーネントのプロセスを活性化，待機状態化させるメソッドであり，他コンポーネントに制御される可能性のあるコンポーネントに対してはこの 2 つのメソッドを実装する．提案手法ではこのように，コンポーネントのとりうる状態と状態遷移のメソッドを統一することで，振舞い変更時のコンポーネント状態の参照・制御を簡易化するとともに，目的に対する達成状態の参照も可能としている．

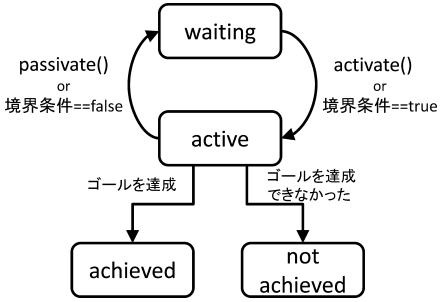

図 6　コンポーネントのライフサイクル
Fig. 6　Component life-cycle.

表 1　コンポーネント仕様
Table 1　Component specifications.

属性名	記述内容
Component	コンポーネント名
Type	コンポーネントの型（Maintain or Achieve）
AchievementCond	達成条件：達成すべき状態を記述する（Achieve 型のみ）
BoundaryCond	境界条件：活性化する条件を記述する（Maintain 型のみ）
RequiredComp	必要とするコンポーネント

さらに提案手法では，コンポーネントの仕様を決定するための要素として，**表 1** に示す属性を定義する．表 1 の属性により構成されるコンポーネント仕様は，ゴール記述とアーキテクチャモデルから実装のための情報を抽出するためのものである．コンポーネントの型を表す Type はゴールパターンに対応し，AchievementCond と BoundaryCond は，コンポーネントが達成すべき状態とコンポーネントが活性化する境界条件を表す．また，RequiredComp はコンポーネントの接続関係に対応する情報である．コンポーネント仕様のうち，RequiredComp に関する情報は，アーキテクチャモデル，つまり KAOS ゴールモデルの構造から形式的な取得が可能であり，Type と AchievementCond に関しては，ゴール記述からの形式的な取得が可能である．BoundaryCond に対しては開発者の判断が必要とな

Component MaintainBatteryLoaded
Type Maintain
BoundaryCond bt.Level < 'Low'
RequiredComp GoToStation, FillBattery

図 7　コンポーネント仕様の例
Fig. 7　Example of component specification.

るが，ゴール記述を参照することで，境界条件に該当する論理式を決定することができる．コンポーネント仕様の例として，ゴール "MaintainBatteryLoaded" に対応するコンポーネント仕様を**図 7** に示す．

4.3　コンポーネントの実装

各コンポーネントの仕様が決定すると，仕様に従ってコンポーネントを実装する．本研究では，コンポーネントの実装に JADE（Java Agent DEvelopment framework）[20] を用いる．JADE は Java 言語により実装されたエージェントプラットフォームであるが，その特徴は，各エージェントが実行するそれぞれのタスクを Behaviour（ビヘイビア）クラスのサブクラスとして実装し，これらをエージェントに割り当てることでエージェントの動作を定義する点にある．本研究においては，次に述べるモデル間関係を定義することで，コンポーネントを JADE 上で実装する．まず，表 1 における Type はコンポーネントの振舞いのスタイルを表したものであり，4.1.2 項で決定した KAOS のゴールパターンが対応する．JADE では，何度も繰り返し実行される命令ブロックを記述するためのビヘイビアとして CyclicBehaviour クラスが提供されており，また，終了条件を定義し，終了条件が真になるまで命令ブロックを繰り返すビヘイビアとして SimpleBehaviour クラスが提供されている．したがって，コンポーネントが定常的な状態維持を必要とする Maintain 型であれば CyclicBehaviour クラスを，記述された状態を 1 度でも達成すればよい Achieve 型であれば SimpleBehaviour クラスを構築するコンポーネントのスーパクラスとして利用する．

AchievementCond に関しては Achieve 型のコンポーネントにおいてビヘイビアの終了条件として利用し，BoundaryCond に関しては Maintain 型のコンポーネントにおいて，命令ブロックを囲んだ条件式として利用することで，自発的な活性化を実現する．RequiredComp は構築するコンポーネント内で参照（利用）するコンポーネントを実装段階で事前に把握するために用いる．

コンポーネントの枠組みが構築できると，続いて，外部からの状態参照を可能とするための mode 変数を実装する．この mode 変数はコンポーネントの処理とその結果に応じて値を変化させる必要がある．コンポーネントのプロセス開始時には mode を active に変更する必要があり，Achieve goals に対応するコンポーネントの場合は，達成条件の成否により値を achieved，not achieved のいずれかにセットする．さらに，他コンポーネントからプロセス状態を変化させることができるように passivate，activate メソッドを実装する．

以上の項目を各コンポーネントごとに実装するが，特に衝突に対して責務を持つコンポーネントにおいては，衝突に関与するコンポーネントの activate，passivate メソッドを呼び出すことでコンポーネントの実行状態や実行順序を制御し，衝突の回避を実現するプログラムを実装する．

5.　self-adaptive システム構築実験

提案手法の実現可能性を検証するために，本研究では清掃ロボットの構築を想定したシミュレーション実験を実施した．

本実験では，要求分析により構築された図 2 の KAOS モデルに対して図 5 で示したアーキテクチャモデルを構築し，その後，提案手法に従って JADE 上で動作する清掃ロボットをシミュレータ上に実装した．清掃ロボットの実装にあたっては，コンポーネントの切替えが分かるように，コンポーネントが active になった段階でコンポーネント名をログ出力するようプログラム実装した．なお，本実験においては，以下の環境を想定したシミュレータを構築し，利用している．

- ロボットは現在地から最も近くにあるごみを見つけることができる．
- ロボットはごみに対して，ほうきで掃くか，アームで掴むことによりフィールドを清掃する．一方で，ごみはその特性により，ほうきで掃くことができるものとできないもの，アームで掴めるものと掴めないものとがある．
- ごみには Pile（山），Litter（散乱したもの），Can（缶）の 3 種類の形状があり，ロボットはその形状のみでごみを認識することができる．
- ロボットが所持するバッテリは最大容量が 100 であり，初期状態では最大容量まで充電されている．バッテリは，1 マスの移動に 4，周囲の探索に 3 消費する．

実験では，3×3 から 7×7 のフィールドまでのいくつかの配置パターンを用意し，それぞれにおいて清掃ロボットがごみを収集するプロセスを観測した．

5.1　実 験 結 果

実験結果の 1 つとして，**図 8** (a) のフィールドに対する清掃ロボットの移動経路を図 8 (b) に，ログ出力結果の一部を**図 9**，**図 10** に示す．まず，図 9 は座標 $(1, 4)$ に配置された Junk を清掃する局面を示したものであるが，清掃ロボットは当初，Junk の外見が Pile であるこ

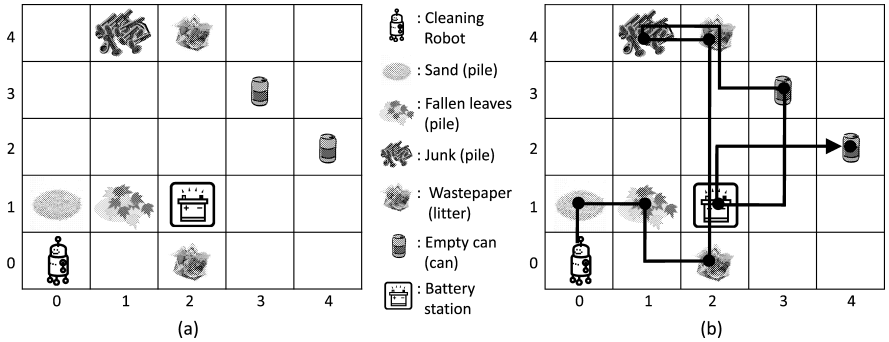

図 8　実験結果：(a) 5 × 5 フィールドへのオブジェクト配置例，(b) 清掃ロボットの移動経路
Fig. 8　Results of experiments: (a) Arrangement of objects in field map gridded 5 × 5 and (b) trajectory of cleaning robot.

```
 1: [java] Component: Move
 2: [java]   Moved to (1,4).
 3: [java]  Rest battery: 49
 4: [java] Component: CleanField
 5: [java]  Identified object type :PILE
 6: [java] Component: SweepDust
 7: [java]  Sweep dust at (1,4).
 8: [java]  Dust at (1,4) can not be swept.
 9: [java] Change to pick up.
10: [java] Component: PickUpDust
11: [java]  Pick up dust at (1,4).
12: [java]  Dust at (1,4) was picked up.
```

図 9　清掃方法の変更
Fig. 9　Log for modifying of cleaning method.

とから，ほうきでごみを掃くためのコンポーネント “SweepDust” を起動してほうきによる清掃を実施したが，失敗して（6〜8 行目），アームでの清掃に切り替えていることが分かる（10，11 行目）．この振舞いの変化は，図 5 のアーキテクチャモデルにおける “CleanField” コンポーネント，つまり図 2 の KAOS モデルにおける “CleanField” を親ゴールとする OR–分解の部分に該当し，子コンポーネントにより実現される複数の清掃手段から親コンポーネントが適切な手段を選択し，タスクの達成状況をもとに振舞いを切り替えることができたことを示している．

```
 1: [java]  Dust is found at (4,2).
 2: [java] Component: Move
 3: [java]   Moved to (3,2).
 4: [java]  Rest battery: 27
 5: [java] Battery is low ...
 6: [java] Component: ApproachStation
 7: [java] Component: Find
 8: [java]  Rest battery: 24
 9: [java]  Station is found at (2,1).
10: [java] Component: Move
11: [java]   Moved to (3,1).
12: [java]  Rest battery: 20
13: [java] Component: Move
14: [java]   Moved to (2,1).
15: [java]  Rest battery: 16
```

図 10　バッテリ残量低下にともなう移動目標の変更
Fig. 10　Log for changing target caused by battery decreasing.

一方の図 10 は，清掃ロボットが移動目標をごみからバッテリステーションへと切り替えた局面に対応するログである．このログからは，次のごみがある座標 $(4, 2)$ に向かって移動を開始したが（1〜3 行目），バッテリが低下したため（5 行目），バッテリステーションへ接近するためのコンポーネント “ApproachStation” が活性化し，ステーションへの移動へとプランを変更している（6 行目以降）ことが分かる．また，本実験ではバッテリ充電後，ごみの清掃（この例では座標 $(4, 2)$ の空き缶の除去）プロセスを再開することも確認できた．これらは図 2 の KAOS モデルにおけるゴール “ApproachDust” と “ApproachStation” 間の衝突に対応した動作であり，内部状態であるバッテリ残量がトリガとなって，競合する振舞いが切り替えられた結果によるものである．

次に，図 10 の局面に対応する振舞いを実装したプログラムの一部を示す．**図 11** はコンポーネント “MaintainBatteryLoaded” の実装内容である．このコンポーネントは，KAOS 分析の結果により衝突を制御するコンポーネントとして検出され，衝突回避のための制御が実装されたものである．本実験では，バッテリ残量が境界条件である閾値を下回ると（1 行目），競合の対象となるコンポーネント “DealWithDust” を待機状態にして（3 行目），子ゴールに対応する 2 つのコンポーネントを起動するという形態で実装することができた．一方で，passivate メソッドが呼び出されるコンポーネント “DealWithDust” では，passivate メソッド内に子ゴールに対応するコンポーネントの passivate メソッドを記述することで，

```
 1:  if(((CleaningRobot)myAgent).robot.getBattery() < CleaningRobot._BATTERY_THRESHOLD_){
 2:     this.mode = Mode.ACTIVE;
 3:     ((CleaningRobot)myAgent).dealWithDust.passivate();
 4:     if(approachStation == null){
 5:        System.out.println("Battery is low ...");
 6:        approachStation = new ApproachStation((CleaningRobot)myAgent);
 7:        addBehaviour(approachStation);
 8:        approachStation.activate();
 9:     }
10:     if(approachStation.mode==Mode.ACHIEVED){
11:        if(fillBattery == null){
12:           fillBattery = new FillBattery((CleaningRobot)myAgent);
13:           addBehaviour(fillBattery);
14:           fillBattery.activate();
15:        }
16:     }
17:  }
18:  if(fillBattery != null && fillBattery.mode == Mode.ACHIEVED){
19:     ((CleaningRobot)myAgent).dealWithDust.activate();
20:     approachStation = null;
21:     fillBattery = null;
22:     mode = Mode.PASSIVE;
23:  }
     ...
```

図 11　コンポーネント MaintainBatteryLoaded の実装内容
Fig. 11　Implementation of the “MaintainBatteryLoaded” component.

階層的に子ゴールの非活性化を実現することができた．このように提案手法の開発スタイルを導入することで，衝突の制御を実現するコンポーネントを特定するとともに，activate および passivate メソッドを用いることで，コンポーネントの切替えによる衝突の解消を実現することができるといえる．

また，図 11 の 10 行目や 18 行目は，子ゴールの状態に応じた分岐処理を実現したものであるが，これはコンポーネントのライフサイクルを導入することで実現できた記述である．つまり，提案手法に従った self-adaptive システムの実装は，コンポーネントの現在の状態や，コンポーネントに割り当てられたゴールの達成状態を評価できる機構を提供しているといえる．

5.2 考　　察

構築した清掃ロボットの挙動と清掃ロボットの構築過程をもとに，本手法の有効性を 2.1 節で定義した各要件に対して評価し，本手法の適用可能範囲について議論する．まず，要件 1 の「自身の状態を評価することができるアーキテクチャである」に関しては，実験結果からも分かるように図 4 に示したコンポーネントモデルを用い，また，3 層アーキテクチャのコンポーネント制御層と同様のコンポーネント接続形態をとることで，mode の参照による各コンポーネントの内部状態の評価が可能である．提案手法ではさらに，KAOS で記述された要求モデルからアーキテクチャモデルを生成することで，各コンポーネントにその目的が明確に割り当てられ，また，mode のとりうる状態として目的に対する達成状態も追加することで，目的の達成に対する評価も実現されたといえよう．

要件 2 の「振舞いモジュールの動的変更が可能なアーキテクチャである」に関しても，提案手法では，3 層アーキテクチャのコンポーネント制御層と同様のコンポーネント接続形態をとることにより，コンポーネントの切替えによる振舞いの切替えを実現している．加えて，提案手法では KAOS モデルの階層構造から変更や実行順序制御の責務を持つコンポーネントを特定し，またコンポーネントを活性化・非活性化させるメソッドを利用することで，動的変更における各コンポーネントの役割・記述内容を明確に分離することも可能となったといえよう．なお，本特徴に関してはオブジェクト指向設計原則の 1 つである単一責任の原則[21] の観点から，コンポーネントの設計にも有益であるものと考えられる．

要件 3 の「目的と振舞いとを対応付けることができる」に関しては，提案手法では，ゴールや要件をコンポーネントと対応付けることにより，KAOS モデル上で記述された要求，つまり self-adaptive システムが持つべき目的が，各コンポーネントに対応付けられることとなる．この際，単に KAOS モデル上の要件に対応した，機能を実現するためのコンポーネントだけでなく，子ノードの達成状態を監視し，実行を制御するような親ノードに対応するコンポーネントにもその役割が対応付けられるため，環境変化やタスク実行結果によって振舞いを変える必要のある self-adaptive システムの開発には特に有効であると考えられる．また，実験結果が示すように，提案手法では目的の達成状況をコンポーネントの状態にも対応付けることで，mode 値の参照により，振舞いが実現すべき目的の達成状況の参照も可能であるといえる．目的と振舞いとの対応付けに関しては，本手法に対して今後，ゴール記述に制約，あるいは詳細なパターンを追加することで，下位ノードが実現する振舞いに対する実行条件や実行制御を決定するなど，目的と振舞いとのさらなる対応付けも可能であると考える．

最後に要件 4 の「振舞い変化時の衝突を考慮することができる」に関しては，実験における清掃からバッテリ補給への振舞い変更に該当する箇所の開発プロセスが該当する．提案

手法では，まず，KAOS モデルを利用することで要求分析の段階で競合（衝突）を検出し，衝突するゴールあるいは要件間の優先順位を決定することで，衝突の存在を確認し，対応指針を決定することが可能であるといえる．また，衝突に対応すべきゴール・要件を同定し，1 対 1 対応するコンポーネントも特定することが可能であることから，衝突を回避する責務を持つコンポーネントを同定することができ，衝突に関連するコンポーネントを限定できるともいえよう．さらに実装においても，関連するコンポーネントの activate，passivate メソッドを用いることで，コンポーネントの切替えによる衝突の解消を実現することも可能である．

以上の考察から，提案する開発手法は 2.1 節で定義した 4 つの要件を満たす self-adaptive システム構築手法であるといえよう．

6. 自動化の可能性と適用範囲

最後に，提案手法の自動化の可能性と適用範囲に関して議論する．まず，KAOS モデルからアーキテクチャモデルへの変換に対しては，コンポーネント間の接続関係は一意であることから，変換の自動化は可能である．この際，ソースモデルとなる KAOS モデルからはゴールモデル構造の取得が必要となるが，KAOS のモデリングツールとして知られる Objectiver [22] などは構築モデルの XML 形式での出力が可能であり，ゴールモデル構造の取得が可能である．筆者らはすでに文献 23)，24) においても，この階層構造を利用した変換手法を提案している．また，コンポーネント仕様の構築に関しても，4.2 節で述べたように大半の属性は自動取得が可能である．

一方で，KAOS モデルと実装モデルの構築に対しては現状自動化が難しい．KAOS モデル構築に関しては，主にサブゴールへの詳細化とゴール記述の構築，衝突の検出を開発者の手作業により実施する必要がある．ただし，これらに対してはいくつかの既存研究やガイドラインを提示することで，開発者の負担を軽減することができると考えられる．まず，サブゴールへの詳細化に関しては，提案手法では KAOS モデルとアーキテクチャモデルが対応付けられるため，反復的な分析を実践することでアーキテクチャモデルがゴールモデル洗練化の 1 つのガイドラインとなりうるであろう．また，文献 17) に示されたゴール洗練化のパターンや，self-adaptive システムに対するアナリシスパターンの提供も有効であろう．ゴール記述に関しても，本論文で利用した Maintain や Achieve などのパターンを効率的に利用することで，開発者の負担を軽減することは可能と考えられる．

衝突の検出に関しては，文献 7)，18)，25) において形式的な検出法が提案されているが，検出のために適当なドメイン知識を用意する必要があり，KAOS の検証ツールとして知られる FAUST [26] においても衝突への対応は十分にサポートされていない[27] など，現段階では手動による検出が必要である．ただし，衝突の検出に関しては，上述の形式的な検出法に基づいた衝突のパターンやヒューリスティック[18] が提案されており，当分野における今後の成果が期待される．

実装モデル構築に対する自動化に関しては，コンポーネントの Type に応じた継承クラスの記述など，一部の構造に関するテンプレート生成は可能と考えられる．振舞い実装に対するコードの自動生成は一般に難しいが，限定されたコード生成や検証機能の提供に関しては，モデル駆動アーキテクチャ[28] で用いられる制約記述言語 OCL [29] や，形式仕様記述言語[30),31)] に分類される VDM [32),33)]，JML [34] などをコンポーネント仕様の記述に利用することで一定の支援は期待できる．ただし，実装モデル構築の自動化を推進するためには，本論文で提案するコンポーネント仕様に対して，属性追加や記述制約などの拡張が必要となる．

続いて，適用範囲に関して議論する．まず本手法では，要求モデル上で抽出された機能や目的をコンポーネントに割り当てることを前提としているため，少なくとも KAOS モデルで最下層に位置する要件に関しては，その達成状態や状態遷移を明確に記述できる必要がある．また，システムの振舞いを厳密に検証し，衝突を正確に検出するためには，ゴールの分解に関しても，ゴールツリーにおける親のゴールと子のゴールあるいは要件間で状態の事前条件と事後条件（達成状態）が等価になるような分解が必要となる．したがって，特に本論文で実験対象としたロボットの分野や，明示的なサービスを提供するユビキタスコンピューティング，あるいは Web サービスの分野，ワークフローを基盤としたビジネスアプリケーションにおいては本手法の適用が有効であると考えられる．

一方で，システムに割り当てる機能に対して達成状態を明示的に記述できない場合や，システム構築時に利用すべきコンポーネントが定められている場合，マルチスレッドプログラミングに従ったコーディングをサポートしていないなどプログラム言語に制約がある場合は，本手法の適用は限定的となる．ただし，既存コンポーネントの利用に関しては，KAOS モデル上で該当する要件が果たすべき機能にそのコンポーネントが包括される範囲であれば，ラッパとなるコードの追加により，本手法の適用は可能であると考えられる．

実装の観点からは，スレッドの実行周期に対する制約も考慮する必要がある．たとえば，非常に細かい時間単位でのスループット監視が要求される場合などは，Maintain 型のコンポーネントにおける境界条件のチェックが，保証すべき単位時間内に必ず実行されなければ

ならない．このような厳しい時間制約がある問題においては，そもそも本手法の適用が困難な場合もあるが，不要なコンポーネントを待機状態化させることにより同時実行されるスレッド数を削減したり，コンポーネントの粒度を詳細化し，各スレッドの実行時間を短縮したりするなどの手段を検討する余地はある．

実装フレームワークに関しては，本論文では JADE を用いた実装手法を示したが，JADE のビヘイビアはスレッドの概念を利用して実装されたものであり，本論文で示した実装手段は一般のマルチスレッド・プログラミング環境においても適用可能である．ただし，提案手法を適用するためには，CyclicBehaviour や SimpleBehaviour に該当するスレッドを用意するために，スレッドを繰り返し実行するための機構や終了条件によりスレッドを終了させるための機構を別途実装する必要がある．

7. まとめと今後の展望

本研究では，近年注目されている self-adaptive システムの効率的な構築法として，ゴール指向要求分析法 KAOS を利用したアーキテクチャモデルの構築法と，得られたアーキテクチャから実装コードを構築するための実装ガイドラインを提案し，その有効性と実現可能性を self-adaptive システム構築実験を通じて評価した．本手法により，要求に適合し，柔軟に振舞いを変化させることのできる self-adaptive システムの系統的な構築手段が提供されると考える．

今後は以下の観点から本手法を洗練化させる予定である．まず，振舞い間に複雑な衝突が存在する問題への適用を目的として，本手法におけるゴール記述法を厳密に定義し，衝突の形式的な検出機能について検討を進める予定である．あわせて，衝突に対するコンポーネントの設計・実装指針も示したい．さらに，各モデル間の形式的な変換ルールを定義することによる，モデル変換に基づいた self-adaptive システム構築手法についても検討する予定である．この手法により，要求分析結果に矛盾しない self-adaptive システムの構築と，衝突に対する形式的な検証および制御の実装が期待できるが，本ツールに関しては，すでに提案している KAOS モデルをソースモデルとしたモデル変換手法[23),24)] の成果が利用できるものと考える．self-*システムに関しては，実現に向けてまだまだ解決すべき課題が多く残されているが，本研究の試みが実世界に適応するソフトウェアの構築に対する 1 つの有効な手段となれば幸いである．

参 考 文 献

1) *The 1st IEEE International Conference on Self-Adaptive and Self-Organizing Systems* (*SASO*), Boston, MA, USA (July 2007).
2) Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., et al.: Software engineering for self-adaptive systems: A research road map, *Dagstuhl Seminar Proceedings 08031* (2008).
3) Nuseibeh, B.: Weaving together requirements and architectures, *IEEE Computer*, Vol.34, No.3, pp.115–117 (2001).
4) Kramer, J. and Magee, J.: Self-managed systems: An architectural challenge, *Future of Software Engineering* (*FOSE '07*), pp.259–268 (2007).
5) Sykes, D., Heaven, W., Magee, J. and Kramer, J.: From goals to components: A combined approach to self-management, *Proc. International Workshop on Software Engineering for Adaptive and Self-managing Systems* (*SEAMS '08*), Leipzig, Germany, pp.1–8, ACM (2008).
6) Dardenne, A., van Lamsweerde, A. and Fickas, S.: Goal-directed requirements acquisition, *Science of Computer Programming*, Vol.20, No.1-2, pp.3–50 (1993).
7) Letier, E.: Reasoning about Agents in Goal-Oriented Requirements Engineering, Ph.D. thesis, Universite Catholique de Louvain (2001).
8) Satyanarayanan, M.: Pervasive computing: Vision and challenges, *IEEE Personal Communications*, Vol.8, pp.10–17 (2001).
9) Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S. and Wolf, A.L.: An architecture-based approach to self-adaptive software, *IEEE Intelligent Systems*, Vol.14, No.3, pp.54–62 (1999).
10) Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N. and Zambonelli, F.: A survey of autonomic communications, *ACM Trans. Autonomous and Adaptive Systems* (*TAAS*), Vol.1, No.2, pp.223–259 (2006).
11) Gat, E., Bonnasso, R.P., Murphy, R. and Press, A.: On three-layer architectures, *Artificial Intelligence and Mobile Robots*, pp.195–210, AAAI Press (1998).
12) Rao, A.S. and Georgeff, M.P.: BDI agents: From theory to practice, *ICMAS*, pp.312–319, The MIT Press (1995).
13) Morandini, M., Penserini, L. and Perini, A.: Towards goal-oriented development of self-adaptive systems, *Proc. International Workshop on Software Engineering for Adaptive and Self-managing Systems* (*SEAMS*), Leipzig, Germany, pp.9–16 (2008).
14) Dastani, M., van Riemsdijk, M.B. and Meyer, J.-J.Ch.: Goal types in agent programming, *ECAI*, Brewka, G., Coradeschi, S., Perini, A. and Traverso, P. (Eds.), pp.220–224, IOS Press (2006).

15) van Lamsweerde, A.: From system goals to software architecture, *Proc. Formal Methods for Software Architectures*, LNCS 2804, pp.25–43, Springer (2003).
16) van Lamsweerde, A.: Goal-oriented requirements engineering: A guided tour, *5th IEEE International Symposium on Requirements Engineering* (*RE'01*), Toronto, Canada, pp.249–262 (2001).
17) Darimont, R. and van Lamsweerde, A.: Formal refinement patterns for goal-driven requirements elaboration, *SIGSOFT FSE*, pp.179–190 (1996).
18) van Lamsweerde, A., Letier, E. and Darimont, R.: Managing conflicts in goal-driven requirements engineering, *IEEE Trans. Softw. Eng.*, Vol.24, No.11, pp.908–926 (1998).
19) Hirsch, D., Kramer, J., Magee, J. and Uchitel, S.: Modes for software architectures, *EWSA*, LNCS, pp.113–126 (2006).
20) Telecom Italia: JADE. http://jade.tilab.com/
21) Martin, R.C.（著），瀬谷啓介（訳）：アジャイルソフトウェア開発の奥義，ソフトバンククリエイティブ (2004).
22) CEDITI: Objectiver. http://www.objectiver.com/
23) 中川博之，吉岡信和，本位田真一：IMPULSE：KAOS を利用したマルチエージェントシステムの分析モデル構築，情報処理学会論文誌，Vol.48, No.8, pp.2551–2565 (2007).
24) 中川博之，田口研治，本位田真一：モデル変換に基づく要求記述を利用した形式仕様の構築，情報処理学会論文誌，Vol.49, No.7, pp.2304–2318 (2008).
25) van Lamsweerde, A. and Letier, E.: Handling obstacles in goal-oriented requirements engineering, *IEEE Trans. Softw. Eng.*, Vol.26, No.10, pp.978–1005 (2000).
26) CETIC: FAUST. http://faust.cetic.be
27) Ponsard, C., Massonet, P., Molderez, J.F., Rifaut, A., van Lamsweerde, A. and Van, H.T.: Early verification and validation of mission critical systems, *Formal Methods in System Design*, Vol.30, No.3, pp.233–247 (2007).
28) OMG: Model Driven Architecture. http://www.omg.org/mda/
29) OMG: OCL 2.0 specification (2005). http://www.omg.org/docs/ptc/05-06-06.pdf
30) 荒木啓二郎，張　漢明：プログラム仕様記述論，オーム社 (2002).
31) 中島　震：ソフトウェア工学の道具としての形式手法，Technical Report, NII テクニカル・レポート (2007).
32) Fitzgerald, J. and Larsen, P.G.: *Modelling Systems, Practical Tools and Techniques in Software Development*, Cambridge University Press (1998).
33) Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N. and Verhoef, M.: *Validated Designs for Object-oriented Systems*, Springer (2005).
34) The Java Modeling Language (JML). http://www.cs.ucf.edu/~leavens/JML/

付　　録

A.1　KAOS モデルからアーキテクチャモデルへの変換アルゴリズム

[入力] KAOS ゴールモデル

[出力] アーキテクチャモデル

[初期化] *parentsList* ⇐ *null*;
childrenList ⇐ *null*;

[手順]

agent ⇐ self-adaptive システムに該当する KAOS ゴールモデル上のエージェント;
parentsList.add(*agent*);
アーキテクチャモデル上に *agent* に対応するコンポーネントを追加;
for all *parent_node* in *parentsList* **do**
　parentComponent ⇐ *parent_node* に対応するコンポーネント;
　if *parent_node* はエージェントである **then**
　　childrenList ⇐ *parent_node* に責務割り当てされたゴール群;
　else
　　childrenList ⇐ *parent_node.getChildren*(); {*parent_node* がゴール・要件であれば，ゴールモデル上のサブゴール群をリストにセット }
　end if
　for all *child_node* in *childrenList* **do**
　　if (*parent_node* はエージェントである) **or** (*parent_node* と *child_node* とが AND–分解の関係にある) **then**
　　　アーキテクチャモデル上に *child_node* に対応するコンポーネントを追加;
　　　childComponent ⇐ *child_node* に対応するコンポーネント;
　　　parentComponent にサービス要求ポートを追加する;
　　　childComponent にサービス提供ポートを追加する;
　　　追加したサービス要求ポートと提供ポートとを連結する;
　　else if *parent_node* と *child_node* とが OR–分解の関係にある **then**
　　　アーキテクチャモデル上の *parentComponent* の内部に，*child_node* に対応するコンポーネントを追加;
　　　childComponent ⇐ *child_node* に対応するコンポーネント;

childComponent にサービス提供ポートを追加する;
追加したサービス提供ポートと *parentComponent* のサービス提供ポートとを連結する;
end if
parentsList.add(*child_node*);
end for
parentsList.remove(*parent_node*);
end for

(平成 20 年 12 月 11 日受付)
(平成 21 年 7 月 2 日採録)

中川　博之（正会員）
1974 年生．1997 年大阪大学基礎工学部情報工学科卒業．同年鹿島建設（株）に入社．2007 年東京大学大学院情報理工学系研究科修士課程修了，2008 年同大学院博士課程中退．同年より電気通信大学助教，現在に至る．要求分析，形式手法，エージェント技術の研究に従事．電子情報通信学会，IEEE 各会員．

大須賀昭彦（正会員）
1981 年上智大学理工学部数学科卒業．同年（株）東芝入社．同社研究開発センター，ソフトウェア技術センター等に所属．1985〜1989 年（財）新世代コンピュータ技術開発機構（ICOT）出向．2007 年より，電気通信大学大学院情報システム学研究科教授．工学博士（早稲田大学）．主としてソフトウェアのためのフォーマルメソッド，エージェント技術の研究に従事．1986 年度情報処理学会論文賞受賞．電子情報通信学会，日本ソフトウェア科学会，人工知能学会，IEEE CS 各会員．

本位田真一（フェロー）
1953 年生．1978 年早稲田大学大学院理工学研究科修士課程修了．（株）東芝を経て 2000 年より国立情報学研究所教授，2004 年より同研究所アーキテクチャ科学研究系研究主幹を併任，現在に至る．2008 年より同研究所先端ソフトウェア工学・国際研究センター長を併任，現在に至る．2001 年より東京大学大学院情報理工学系研究科教授を兼任，現在に至る．現在，早稲田大学客員教授，英国 UCL 客員教授を兼任．2005 年度パリ第 6 大学招聘教授．工学博士（早稲田大学）．1986 年度情報処理学会論文賞受賞．日本ソフトウェア科学会理事，情報処理学会理事を歴任．ACM 日本支部会計幹事，情報処理学会フェロー，日本ソフトウェア科学会編集委員長，FIT2009 プログラム委員長，日本学術会議連携会員．