ゴール指向要求記述に基づいたソフトウェアシステム進化手法に関する研究

Software Evolution Based on Goal-Oriented Requirements Description

2013 年 2 月

中川　博之

ゴール指向要求記述に基づいたソフトウェアシステム進化手法に関する研究

Software Evolution Based on Goal-Oriented Requirements Description

2013 年 2 月

早稲田大学大学院　基幹理工学研究科

中川　博之

目次

図目次

表目次

第 1 章

序論

本研究では，ソフトウェアの更新活動であるソフトウェア進化に対処するための効果的な開発手法として，ゴール指向要求記述を活用したソフトウェアシステムの統合的な開発プロセスを提案する．特に，ゴール指向要求記述に対する整形プロセスを導入し，進化に対応するソフトウェアシステム設計・実装のための各種情報の抽出法を検討する．また，要求記述からシステムの構成に必要なコンポーネントとその接続関係を導出するシステム構成決定法を導入し，これを自動化するコンパイラと，提案する開発プロセスに基づいてシステムを実装するための 1 つの実装基盤を提案する．

本章では，まず本論文の背景であるソフトウェア進化に関する研究動向について述べ，続いて，ソフトウェア進化を実現するためにシステム開発手法に求められる要件を定義する．最後に，本論文の構成について述べる．

1.1　ソフトウェア進化

近年，ソフトウェアの活躍する場面が広がり，オンラインバンキングシステムや電子商取引システムなどの Web 上のシステムだけでなく，携帯端末や自動車の制御システムなど，我々が生活するあらゆる局面において，ソフトウェアシステムが動作している．これらの多くは常に動作していることが前提であり，特に制御ソフトウェアなど，一部のシステムにおいては長期間にわたり高い可用性が求められている．

ソフトウェアとハードウェアとの大きな相違として，ソフトウェアにはハードウェアのような摩耗や劣化，損傷がないという点とともに，ソフトウェアにおいては環境や要求の変化に伴う継続的な変更が期待されているという点が挙げられる．ソフトウェア開発にとって変更は不可避なものであり，複数の調査結果から，ソフトウェアの保守や進化に必要なコストが，ソフトウェアシステムに要するコスト全体の 50%〜90% を占めるということが示されている

[6, 7, 8, 9].

このような特性を持つソフトウェアを，Lehman [10, 11] は S 型，P 型，E 型の 3 種類に分類し，それぞれのソフトウェアが扱うべき変更を明らかにしている．S 型システムは，事前に記述されている仕様を満足するシステムであり，仕様は唯一の完全なものである．数式の演算を実行するようなシステムが該当し，仕様，つまり解くべき問題が変更することはない．P 型システムは，記述された問題を解決できる解法を決定することが要求される種類のシステムである．この種類のシステムは，完全に正確な解の仕様を記述することが難しい問題を扱う場合が多い．仕様に基づいて解を実装した後にその解が受け入れられるかどうかを判断し，もし受け入れられない場合は，受け入れ可能な解が実装されるまで，解の仕様を変更する必要がある．P 型システムとしては，例えば，チェス・将棋を打つシステムなどが挙げられる．E 型システムは，実世界の問題を解くシステムであり，実世界の一部として組込まれるシステムを指す．例えば，OS や航空管制システムや株式市場のシステムなどが該当する．E 型のシステムにおいては，人間の行動の変化により，前提条件となる問題が変化する．従って，問題が変化することで，システムにも変更が求められることを意味している．

Lehman の分類において，近年では E 型のソフトウェアシステムが圧倒的に多い．近年では要求や環境の変化が生じることは前提であり，よってシステム変更を前提としたシステム設計，開発が求められている．このような背景を予想してからか，Lehman は，IBM OS/360 をはじめとする長年のソフトウェアシステムのプログラム変更に対する時系列データ分析結果をもとに，E 型ソフトウェアシステムの進化を支配する法則（Laws of Software Evolution）を下記のように定義している [12, 10, 13, 14, 15] .

- 継続的変更（Continuing change）の法則：E 型システムは継続的に変更されなければならない．そうでなければ，徐々に利用価値が減衰することとなる．
- 複雑さ増大（Increasing complexity）の法則：E 型システムは，進化することにより複雑化する傾向にあるので，構造の複雑さを維持，軽減させるための努力が必要である．
- 自主規制（Self regulation）の法則：E 型システムの進化には自主規制がかかっている．サイズやリリース間の期間, 報告されるエラーは各リリース間においてほぼ一定である．
- 組織安定性不変（Conservation of organizational stability）の法則：E 型システムの残存期間を通じて，開発の占める割合は一定であり，システム開発に必要なリソースとは独立している．
- 精通度不変（Conservation of familiarity）の法則：E 型システムの残存期間を通じて，各リリースでの変化はほぼ一定である．
- 継続的成長（Continuing growth）の法則：ユーザーを満足させるためには，E 型システ

ムが提供する機能を継続的に増やしていく必要がある．

- 品質低下（Declining quality）の法則：E 型システムの品質は，運用環境の変化に適用させない限り，低下していく．
- フィードバックシステム（Feedback system）の法則：E 型システムの進化は，マルチループ，マルチエージェントにより構成されており，大幅な改善を達成できるように扱う必要がある．

同様の背景から，Parnas [16] は，“Software Aging” というタイトルで，ソフトウェアも人間同様に高齢化するという主張を展開している．Parnas はソフトウェアの高齢化の原因として，以下の 2 つを挙げている．

- 動きの欠如（Lack of movement）：頻繁に更新されることが無くなれば，その時がソフトウェアの寿命である．
- 無知な外科手術（Ignorant surgery）：ソフトウェアの変更方法によっても加齢の程度が異なる．初期の設計を理解していない開発者の変更により，システムの変更，理解が困難になり，システムを破壊する変更が繰り返されることとなる．

Lehman が定義した法則と Parnas の “Software Aging” という主張は，いずれもソフトウェアの変更を扱う際に直面する，ソフトウェアの本質的な側面を示している．Lientz [17] は，このようなソフトウェアシステムの変更を伴う活動を以下の 4 種類に分類している．

- 適応（Adaptive）： ソフトウェア環境の変化への対応
- 完全化（Perfective）： ユーザの新しい要求への対応
- 修正（Corrective）： エラーの修正
- 予防（Preventive）： 将来発生し得る問題の未然防止

これらの活動の約 75% を適応，拡張が，約 21% を修正が占めているという調査結果もある [18] が，近年のソフトウェアシステムにおいては，システムの長寿命化により，要求や環境の変化による機能の追加や変更，つまり適応や完全化という**ソフトウェア進化（Software Evolution）**[*1][18, 7] の側面がますます重要となってきている．また，ソフトウェアの長寿化の一方で，ソフトウェアシステムを取り巻く環境の変化も激化していることから，要求や環境の変化に対応するためのソフトウェア進化を確実かつ効率的に実現するためのソフトウェア開発プロセスが求められるようになってきている．ソフトウェア進化は大きく，予期の可否 [19]

[*1] 従来のソフトウェア保守と区分するために，変更を積極的に取り入れる行為をソフトウェア進化と呼ぶようになってきている．

により分類することができる．

- 予期可能な進化 (Anticipated Evolution)：Eclipse Plug-in [20] や Firefox Add-ons [21] に代表されるプラグイン技術や，オブジェクト指向における継承のように進化可能な対象があらかじめ分かっている進化
- 予期不能な進化 (Unanticipated Evolution)：進化の対象があらかじめ分かっていない場合の進化

予期可能な進化は，プログラミング言語やプラットフォームが提供する API を利用して反映させることができる一方で，進化の範囲が限定されるという欠点を持つ．一方の予期不能な進化は，広範囲の進化に対応できる一方で，汎用的な手段が提供されているプラットフォームが広く知られていないという問題がある．Bennett [18] も，「ソフトウェア進化の本質的な問題は，多くの変更が当初設計者が想像もしなかったものである」と予期不能な進化はソフトウェア進化の分野における本質的な課題であることを指摘している．従って，進化を考慮したソフトウェア開発プロセスにおいては，予期不能な進化に対しても開発者を支援すべきである．

1.2　システム開発プロセスに求められる要件

ここで，ソフトウェア進化を考慮した場合にシステム開発プロセスに求められる要件について議論する．まず，ソフトウェアシステムの進化は，要求の変化に起因するものである．従って，ソフトウェア進化においては，要求の変化により生じる変更要求（Change request）に対してソフトウェアシステムがどのような影響を受けるかを正確に分析できなければならない．しかしながら，Lindvall [22] らが実証実験により示しているように，経験を積んだソフトウェア開発者であっても，変化によってソフトウェアシステム上にもたらされる影響を分析するのは容易ではない．よって，ソフトウェア進化を許容する開発プロセスは，ソフトウェアの変更により生じる影響の分析（Impact analysis）[23, 24] を支援するものでなければならない．また，ソフトウェア進化には，小さな機能変更から初期開発時には想定していなかった大がかりな機能追加まで様々なものがあり，また，継続的な進化も考慮できなければならない．従って，影響分析後にコード上に変更を反映（Change implementation）する必要があるが，このとき，変更によるコード上の複雑さ（Complexity）[25, 26] の増加をできる限り抑制するような進化が望ましい．

そこで本研究では，これらの議論をもとに，ソフトウェア進化に対してシステム開発手法に求められる要件として，以下を定義する．

定義 1.2.1　ソフトウェア進化の観点から開発プロセスに求められる要件

- **要件 1：影響分析精度の向上 —** 進化要求によってシステム上で変更すべき箇所をより正確に同定することができる．
- **要件 2：コード複雑化の抑制 —** システム上で変更が必要な箇所を限定し，変更によるコード複雑化を効果的に抑制することができる．

Breivold ら [27] は，ソフトウェア進化に影響すると考えられる特性として，分析可能性（Analyzability），整合性（Integrity），変更可能性（Changeability），拡張容易性（Extensibiilty），移植性（Portability），テスト容易性（Testability）とその他のドメイン固有の特性の 7 つの特性を挙げている．ここで，分析可能性は上記の要件 1 に該当し，変更可能性は上記の要件 2 に関連する特性である．また，変更可能性と拡張容易性は同一の特性と判断している研究が多い．さらに，整合性，移植性，テスト容易性についても，特定の状況を対象としていることや上記の要件の副次的な効果であると考えられることから，本研究では，上述の 2 つの要件を満たすソフトウェアシステム開発プロセスの実現を目指す．

1.3 本研究の目的

本研究では，前節で定義した 2 つの要件を満たす開発プロセスの実現を目指す．まず，定義 1.2.1 を満たすためには，進化により生じる影響と，現在のシステムの構成，実装コードとの間の対応関係が明確でなければならない．また，実装コードの複雑さを抑制するためには，適切に設計モデルや実装コードがモジュール化されている必要がある．

ここで，ソフトウェアに関する記述，モデルとして最初に構成されるべきものは，システムを構築する要因となる要求が記されている要求記述である．ソフトウェアの進化は要求の変化に起因するため，要求記述はソフトウェア進化の妥当性を検証するために重要なモデルとなる．従って，開発プロセスとしては，要求の変化が確実に要求記述上に反映され，この要求記述上の変更が，確実に実装コードにまで継承される開発プロセスが求められる．

そこで，本研究では，要求記述をソフトウェアシステム進化の核とする開発プロセスの実現を目指す．Nuseibeh [28] も指摘するように，ソフトウェアに対する要求とシステムアーキテクチャとは相関関係にあり，要求分析結果をシステムアーキテクチャに反映させることは重要である．

しかしながら，要求記述とシステムアーキテクチャとを明示的に関連付ける研究は多くはない．また，要求記述とシステムアーキテクチャとの関連について言及した既存の各手法においても，ソフトウェア進化を考慮した場合には，必ずしも定義 1.2.1 に示した要件を満たしているとは言えない．そこで，本研究では，このような開発プロセスを実現するための開発手法と

して，要求記述中において進化を考慮したソフトウェア構成要素群を同定可能なモデリング手法を導入し，同定した構成要素の設計・実装を支援する開発プロセスを提案する．これにより，進化に対して頑強な構成要素から構築されるソフトウェアシステムの実現を目指す．

1.4　本論文の構成

本論文では，ソフトウェアの進化を考慮した系統的なソフトウェアシステム開発を実現するための，要求記述に基づいたソフトウェア開発プロセスについて述べる．また，ソフトウェア進化を効率的に実現するために本研究で実装した，支援ツール群について述べる．

本論文の構成は以下のとおりである．まず 2 章では，進化を考慮したソフトウェア開発プロセスを実現するために本研究で導入する進化型開発支援フレームワークの概要について述べる．2 章では，本研究においてソフトウェア構成要素の同定のために着目する Control loop の概念についても説明し，併せて，3 章以降で提案手法を説明するために用いるシステム開発例題についても説明する．

続いて，3 章，4 章において，提案する開発手法について論じる．まず 3 章では，ゴール指向要求記述上での，進化を考慮したソフトウェア構成要素同定法について論じる．本決定法は，Control loop の概念を利用して要求記述を洗練化させ，進化を考慮したソフトウェア構成要素を同定するものであり，3 章では本研究で要求記述として利用するゴール指向記述の概要についても触れる．続く 4 章では，本研究で導入する開発プロセスを支援するツール群について述べる．まず，要求記述を入力として，システムアーキテクチャや，ソフトウェア進化に必要な情報を生成するコンパイラについて述べ，続いて，本研究で導入する開発プロセスに従ったソフトウェアシステムを実現するためのプログラミングフレームワークについて述べる．

5 章では，GUI ベースのモデリングツールと制御システムという 2 種類のソフトウェアを対象とした進化実験の内容とその結果を示し，6 章では，5 章で示した実験結果に基づいて本手法の有効性を評価する．最後に 7 章で関連研究を示し，8 章でまとめと今後の課題，展望について述べる．

第 2 章

ソフトウェア進化を考慮した開発プロセス GCLD

2.1 はじめに

前章ではソフトウェア開発におけるソフトウェア進化の位置づけについて説明し，進化の観点から開発プロセスに求められる要件について論じた．

本研究では，前章で定義した要件を満たす開発プロセスを実現するために，ゴール指向要求記述を核とした開発支援プロセス GCLD (Goal-driven Control Loops Distribution) を提案する．以降本章では，まず，ソフトウェア進化を考慮した開発プロセスの実現に対する本研究のアプローチについて論じる．続いて，本研究においてシステムの構成要素として位置付ける Control loop の概念について説明し，その後，本研究で提案する開発プロセス GCLD の概要について述べる．

2.2 本研究におけるアプローチ

ソフトウェアシステムの進化について考えた場合，開発手法においては，要求や環境の変化に対応するために必要なソフトウェア変更の影響が分析可能であり，また，進化を容易にソフトウェアシステム上に実現できるプロセスであることが求められる．

図 2.1 は，進化を考慮した理想的なソフトウェアシステム開発の流れを示したものである．1 章で定義した要件において，特に影響分析を支援するためには，要求の変化がソフトウェアシステムのどの部分に反映されているかを同定できる必要がある[*1]．この同定を実現するため

[*1] このような性質は，追跡可能性（traceability）[29, 30] とも呼ばれる．

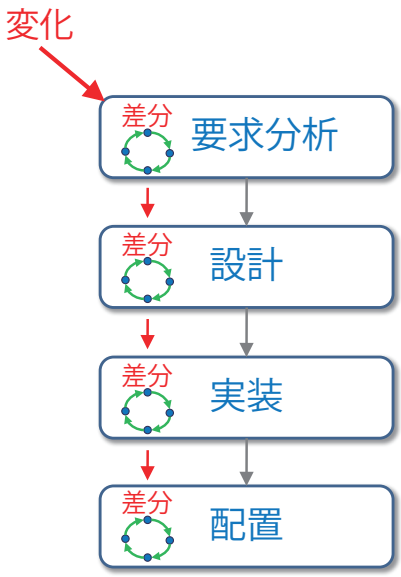

図 2.1. 変化に基づいたソフトウェアシステムの進化

には，コードや設計モデルのみを変更するのではなく，進化時においても初回開発時同様に，要求分析，設計，実装，配置の各フェーズを経て，各フェーズで用いるモデルを適時変更するというアプローチが必要となる．

また，進化を考慮した開発プロセスにおいては，図 2.1 に示すように，変化に対する差分を意識した開発が求められる．まず，要求分析フェーズでは，環境，要求の変化に応じてシステムが新たに満足しなければならない機能要求を抽出する．続いて，設計フェーズにおいては，要求分析フェーズで抽出された変更箇所（差分）のシステム仕様を決定し，実装フェーズにおいて，仕様をもとに変更箇所のコードを作成し，最後に，配置（デプロイ）フェーズにて既存のコードと変更箇所のコードを結合し，システムとして動作させるといった開発の流れが求められる．従って，進化を考慮した開発プロセスにおいては，通常の要求分析，設計，実装，配置といったソフトウェア開発の流れに沿うだけでなく，要求や環境の変化によって生じる各モデルの変更箇所，つまり差分を意識した開発作業が必要となる．

本研究では，このような前提条件から，進化を考慮したソフトウェアシステムの開発プロセスの確立にあたり，要求分析結果を記述する要求記述に着目する．これは，ソフトウェア進化を促すものは環境や要求の変化に起因するものであり，変化の原因をとらえるにはシステムに対する要求やドメイン情報が記述された要求記述が適切であると考えるからである．また，要求記述は，ソフトウェアに関するモデルとして最初に構築されるものであり，変化により生じる進化を適切に分析するためにも相応しいと考えられるからである．さらに，変化に対応する

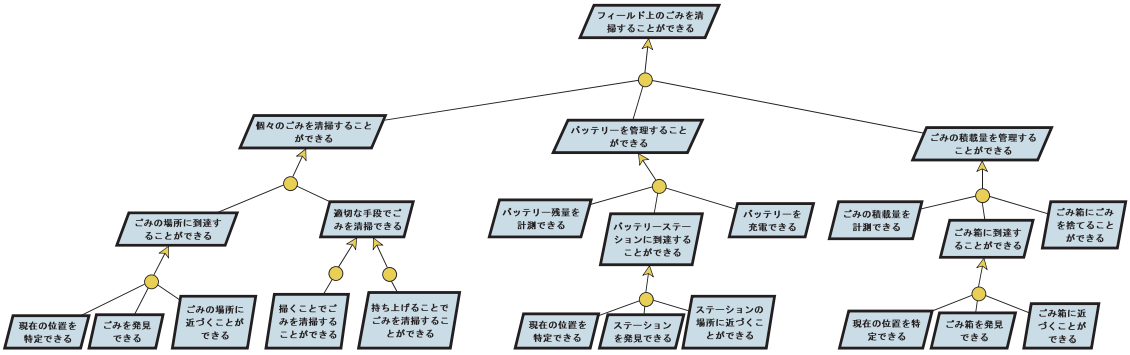

図 2.2. ゴールモデルの一例（清掃ロボットに対する要求記述）

ためにシステムに実装すべき機能の同定や，構成要素への責務割り当て，機能間の競合検出には要求分析時が最も適していると考えられるからである．

特に本研究では，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** [31, 32] に着目し，ゴール指向要求記述を活用した開発手法を検討する．ゴール指向要求分析法とは，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，要求間の関係や，要求と要求を実現するための機能とを関連付ける手法であり，識別したゴールを達成するための機能を系統的に導出する手法である．ゴール指向要求分析法により構築されるモデルはゴール指向要求記述，あるいはゴールモデルと呼ばれる[*2]．ゴールモデルの例を図 2.2 に示す．ゴールモデルにおいては，ゴール間の関係が明確化されるため，進化時における要求変化とその影響範囲の分析に有益であると考えられる．

進化を考慮した場合，要求記述の変更結果を効果的に設計モデルに反映する手段が必要となる．ゴール指向要求記述の情報を設計モデルに反映する手段としては，システムの構造を形式的に決定するいくつかの手法が知られている．Lamsweerde [33] は，ゴール指向要求記述から，システム構造を表現するアーキテクチャモデルのスケッチを構築する指針を提案している．この手法においては，各ゴール達成の責務をコンポーネントに割当て，ゴール間のデータフローからコンポーネント間接続を同定することにより，システムの構造を決定する．しかしながらこのような構築法では，まず最初に，ゴール達成の責務を割り当てるコンポーネントを決定する必要があるが，この判断は容易ではない．例えば，過度に多くのゴールを割当ててしまうと，結果としてコンポーネントの単位が大きくなってしまい，変化の影響を受ける範囲を明確に分析できなかったり，新機能の追加など，変更に対する影響がコンポーネント内の他の部分に及んでしまうこととなり，進化に対して十分な設計ができなくなるという問題点があ

[*2] 本研究では以降，ゴールモデルと呼ぶ．

る．また，逆にゴール達成の責務を細分化し過ぎてしまうと，コンポーネント数が多くなり，コンポーネント間接続も複雑化してしまう．これは，進化時に影響を受ける箇所の把握を難しくすることとなる．従って，ソフトウェアシステムの進化プロセスにおいて Lamsweerde の手法を適用した場合，影響分析精度の向上という要件を満足させることが難しいと言える．

同じくゴール指向要求記述を利用したシステム構成決定法として，Yu ら [34] の研究がある．Yu らの手法においては，ゴールモデル中のゴールをコンポーネントに変換し，ゴール間の接続関係をコンポーネント間の接続に変換している．この手法は，ゴールモデルに一対一対応するシステム構成を決定可能であるという点で，ソフトウェアの進化を考えた場合にも，設計モデル上での変更箇所を同定できるという観点から一見有効であるように思われる．しかし，Yu らの手法においては，入力となるゴールモデルの条件や変換対象とするゴールの範囲については言及していないため，コンポーネントの独立性や責務が十分に明確化されないことになる．その結果，ゴールモデルに新たなゴール群を追加したとしても，ゴールモデル上での変更をシステムの設計，実装上でどう扱うべきかは明らかではない．また，ゴールに一対一対応してコンポーネントを変更するため，類似のゴールが複数あった場合に，それらが共通の変数をアクセスすることで競合発生の可能性が生じることとなる．これらの問題は，影響分析精度の向上という観点からも，コード複雑化の抑制という観点からも要件の満足を難しくする要因となるものである．

ここで挙げた 2 つの手法に関する問題は，一つは，ゴールモデル上で機能実現のための責務が明確に分離されていないことに起因するものである．例えは，Lamsweerde の手法においては，ゴールとコンポーネントとの対応関係を決定できないという難しさは，コンポーネントへ割り当てる責務をゴールモデル上では決定できないという点に起因している．また Yu ら手法においては，ゴールの機能実現に対する責務が明確に分離されていないため，変換されるコンポーネントの独立性を判断することが難しく，結果として，ゴールを追加した際に，既存のゴール，つまりコンポーネントに及ぶ変更の影響を判断することが難しくなっている．

また，ゴールモデル上で変数に対する責務が明確に分離されていないことも，ソフトウェア進化を扱う際の難しさとなっている．Yu らの手法では，ゴールとコンポーネントとを一対一対応させているが，この場合，複数のゴールが同一変数に関与する場合，複数のコンポーネントが同一変数に作用することを意味しており，新たなゴールを追加した場合に競合が発生する要因となる．

つまり，既存のゴールモデルからシステム設計モデルを決定する既存手法においては，機能実現や変数に対する責務を適切に分離するのが難しいという点により，進化を考慮したソフトウェア開発において効果を発揮することが難しいといえる．従って，ソフトウェアシステムの進化を扱う場合，進化の要因となる要求変化と，機能実現，変数に対する責務を同時にゴール

モデル上で可視化・分析できることが求められる．

ここで，ゴールモデルにおける責務分析を考えた場合，ゴールモデルはゴール，すなわちシステムの振舞いにより達成される状態に着目したモデルであることから，分析の単位としては，振舞いに着目した単位での責務割り当てが適切であると考えられる．従って，ゴールモデル上で振舞いの視点から責務を分離できる単位でモデリングすることが，進化を考えた場合には有効であると考えられる．

そこで，本研究では，ゴールモデル上で Control loop を同定する手法に基づいた開発プロセスを導入する．次節では，本研究でシステムの構成要素として着目する Control loop の概要について述べる．

2.2.1 Control loop

システムの振舞いに着目したモデルとして，プロセスコントロールモデル [1] がある．プロセスコントロールは，連続的な入力を動的に管理し，出力を特定の目標の範囲内で維持したい場合に利用される，古くから監視制御システムにおいて用いられてきた概念である．Shaw [1] は，このプロセスコントロールを図 2.3 に示すようなソフトウェアアーキテクチャ上のモデルとして定義している．プロセスコントロールモデルにおいては，処理を実現する処理部と処理部を制御する制御部の 2 つの構成要素が存在し，目標の範囲内の出力を維持するために，外部への出力を変化させる処理部に対して，制御部が状態を監視しながら制御を加える．プロセスコントロールモデルにおいては，以下の 3 種類の**プロセス変数 (Process variables)** が定義されている．

- **入力変数（Input variable）：**処理部に入力される変数
- **制御変数（Controlled variable）：**プロセスコントロールにより制御したい対象を示す変数
- **操作変数（Manipulated variable）：**制御のために変更可能な変数

プロセスコントロールモデルは，制御に対するプロセス変数の利用方法により，大きくフィードバック制御とフィードフォワード制御に分類することができる．目標値と，測定した制御変数の値から，操作変数の値を決定し，その値を処理部に与えることで制御を実現するのがフィードバック制御であり，一方，目標値と入力変数から操作変数を決定するのが，フィードフォワード制御である．

プロセスコントロールモデルにおいては，通常 **Control loop** [2, 35, 1] と呼ばれる制御プロセスが形成される．この制御プロセスは，図 2.4 に示されるように，収集（Collect），分析

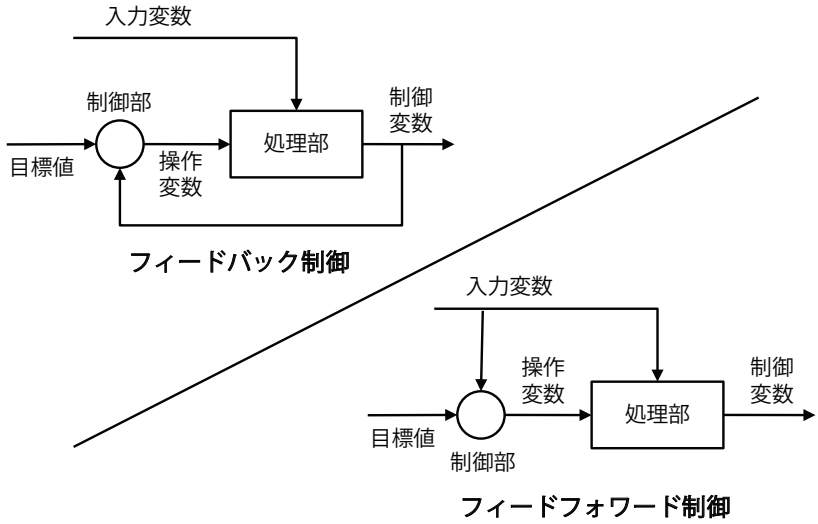

図 2.3. プロセスコントロール [1]（左上：フィードバック制御，右下：フィードフォワード制御）

（Analyze），決定（Decide），実行（Act）の 4 つのアクティビティがこの順序で繰り返し実行されるプロセスである[*3].

- **収集（Collect）：**外部から情報を収集する
- **分析（Analyze）：**情報の収集結果から，現在の状態を分析する
- **決定（Decide）：**環境に適応するためのシステム動作（振舞い）を決定する
- **実行（Act）：**決定結果に基づいた振舞いを実行する

本研究では，ゴールが表現する達成状態を満足するシステムの振舞いとして，収集，分析，決定，実行といった Control loop の観点からシステムをモデリングすることで，入力から出力までの一連のアクティビティを独立して提供可能な範囲をゴールモデル上で決定する．ただし，Control loop の観点から，進化への対応を考慮したシステムの構成要素を同定するが，これは各構成要素の責務を決定するためのものであり，入力変数の値に応じてアクションを決定するという一連のプロセスを実現できれば，各構成要素の設計・実装をプロセスコントロールモデルに束縛するものではない.

[*3] Control loop と同様のプロセスを MAPE (Monitor, Analyze, Plan, Execute) loop と呼ぶ研究グループもある [36, 37].

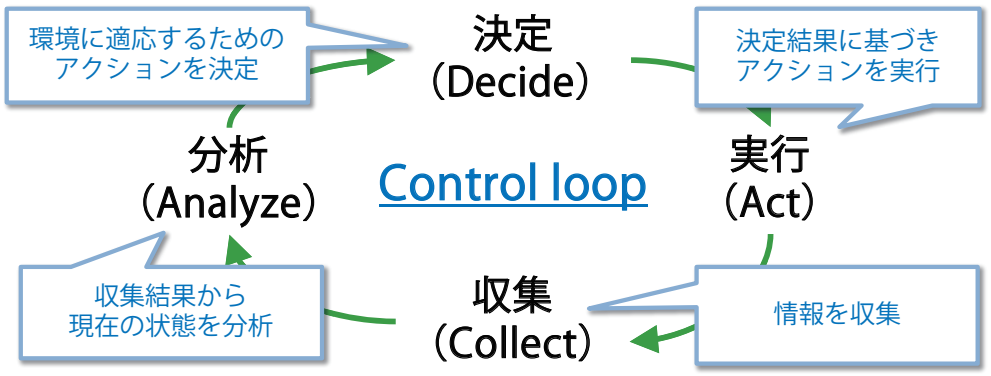

図 2.4. Control loop [2] のアクティビティ

2.2.2 期待できる効果

本研究のアプローチである，ゴールモデル上での Control loop 抽出という観点から，1 章で定義した開発プロセスに求められる各要件に対して期待できる効果について論じる．

まず，影響分析精度の向上に関しては，本研究では，ゴールモデル上でソフトウェアシステムに対する要求を Control loop という制御単位で分割することにより，システムの各機能を独立化させるとともに，システム構成要素の責務を明確化させる．機能の独立化には，関心事単位での分離という分割基準も必要と考えられるため，ゴールモデル上に記述されたゴール間の関係が利用可能であると考える．この機能の独立化により，進化時に，要求の変化によってシステム上に生じる影響の範囲をゴールモデル上で分析，限定化することが可能になると考える．影響分析にあたっては，変更箇所が既存箇所に及ぼす影響として，リソースに対する競合についても考慮する必要がある．本研究では，この競合の解消に，ゴールモデル上での特定構造の発見による，競合発生箇所の検出も試みる．

また，本研究では，ゴールモデル上に記述されたゴールから，システムの構成に必要なコンポーネントとその接続関係を決定する手法を導入することで，要求の変更箇所に対応したシステム構成上での変更箇所を同定する．これにより，要求記述と設計モデル，実装コード間の追跡可能性が向上し，結果として，影響分析の精度向上が期待できる．本研究では，このようなシステムの構成をコンポーネントとその接続関係により定義したモデルを**コンフィギュレーション**と呼ぶ．このコンフィギュレーション決定手法は，各コンポーネントに達成すべきゴールを責務として対応付け，各ゴール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コンフィギュレーションを出力するものである．

上述のような機能の独立化・分解は，実装コード上での変更影響の局所化を促すことから，

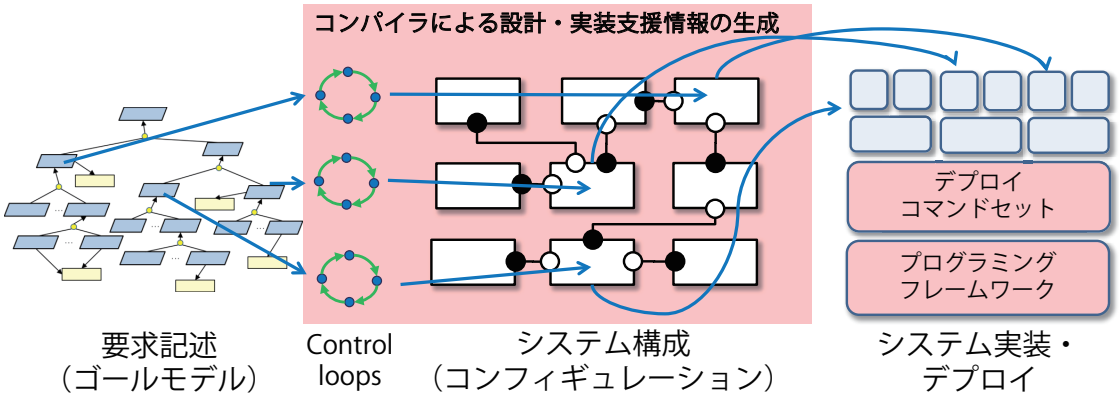

図 2.5. 開発プロセス GCLD (Goal driven Control Loops Distribution) の概要

もう一つの要件であるコード複雑化の抑制も期待できる．コード複雑化の抑制を実現するためには，機能の独立化に従って抽出された Control loop を実装できることが前提となるが，本研究では，ゴールモデル上の情報を利用した Control loop の実装支援手法と，構成要素となる Control loop の実装を支援するためのプログラミングフレームワークを導入し，同フレームワークを利用したシステムの実装指針について検討する．システムを Control loop の集合として実装することで，継続的な進化に対してもコード複雑化の抑制が期待できる．

以上のような，ソフトウェアシステムの進化を実現するための各要件に対する効果を実現するために，本研究では，ゴールモデルを利用した Control loop を構成要素とするシステム開発手法を検討する．

2.3　GCLD の概要

前節で述べたアプローチを実現する開発プロセスとして，本研究ではゴール指向要求分析に基づいた Control loop 配置によるソフトウェア開発プロセス GCLD (Goal driven Control Loops Distribution) を提案する．

本研究で提案する開発プロセス GCLD の概要を図 2.5 に示す．GCLD ではまず，要求が分析，定義されたゴールモデル上からシステムの構成要素となる Control loop を抽出し，システムの構造を表現するコンフィギュレーション上に Control loop を配備する．このコンフィギュレーションの生成は自動化されており，開発者はコンフィギュレーションに従って Control loop を実装する．Control loop の実装は，各システム開発における制約に従うが，本研究では Control loop 実装を支援する 1 つのプログラミングフレームワークを導入した．本フレームワークは，Control loop の実装を支援する API を提供するとともに，実行時に Control loop を

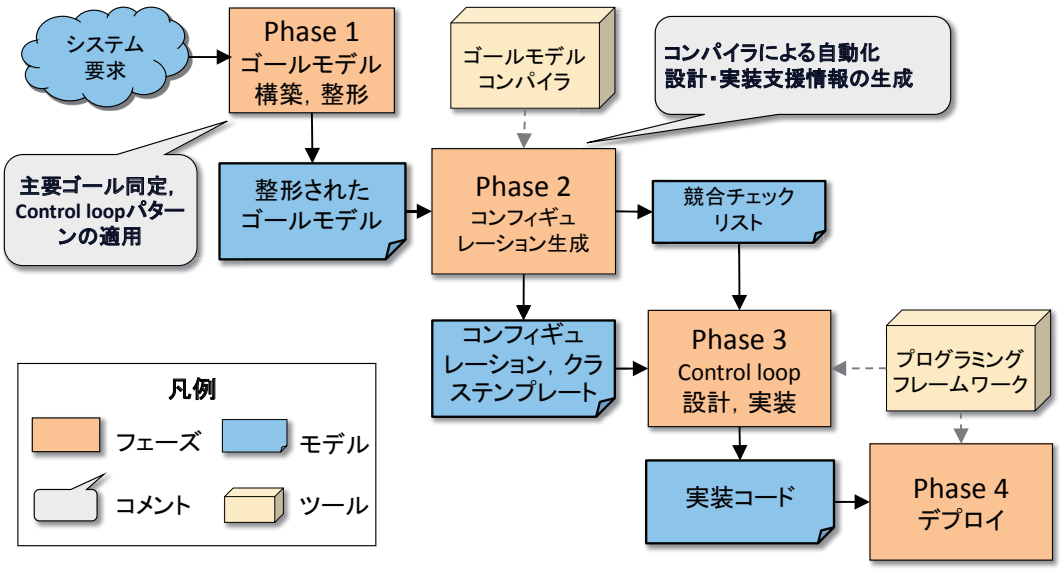

図 2.6. GCLD におけるソフトウェア開発（初期開発時）

ロード，アンロードするコマンドセットも提供している．

GCLD におけるソフトウェア開発プロセスは図 2.6 に示されるように，以下の 4 つのフェーズにより構成される．

1. **ゴールモデル構築，整形：**システム開発者はまず，開発対象となるシステムに対する要求をゴール指向要求分析法に基づいて分析する．このフェーズの目的は，システムに対する要求の分析・定義と，Control loop を抽出するための要求記述の構造化にある．本研究では特に，ゴール指向要求分析法として，設計モデルとの関係が記述可能であり，構造化されたテキストデータである XML へのデータ出力ツールも持つ KAOS [38, 39] を用いる．要求分析法や要求，つまりゴールの記述は通常の KAOS における分析法，記述法に従うが，提案手法では，得られた KAOS のゴールモデルに対して，本研究で提案する整形プロセスに従った整形，詳細化を施すことにより，システム構成を決定可能なゴールモデルへと詳細化する．
2. **コンフィギュレーション生成：**続いて，前フェーズで整形したゴールモデルから，構築するシステムの構成を表現するコンフィギュレーションを決定する．本研究で用いるコンフィギュレーションは，複数のコンポーネントとコンポーネント間接続により定義される，システム構成を表現したモデルである．このコンフィギュレーションは，本研究で導入するゴールモデルコンパイラにより，KAOS のゴールモデルを入力として自動生成される．本研究では特に，ソフトウェア進化に対応するために，複数の Control loop を

持つシステムの構築を目的とし，導入するコンフィギュレーション決定法は整形されたゴールモデルから 1 つ以上の Control loop を含んだコンフィギュレーションを生成する．また，本研究で実装したゴールモデルコンパイラは，コンポーネント実装のためのクラステンプレートや競合リストなど，ソフトウェア進化を支援するための各種情報も生成する．

3.Control loop **設計，実装：**開発者は各 Control loop の動作を決定し，前フェーズで得られたクラステンプレートに対してコンポーネントの振舞いを追加することで，Control loop を構成する各コンポーネントを実装する．本研究では，システムの実装を支援するためのプログラミングフレームワークを導入する．本フレームワークの特徴は，Control loop の実装，制御を支援する点にある．前フェーズで生成されるクラステンプレートも本フレームワークが提供するクラスを継承するものであり，開発者はフレームワークが提供する機能を利用した Control loop の実装が可能である．

4. **デプロイ：**実装した Control loop を実行環境に配備することで，システムを動作させる．本研究で導入するプログラミングフレームワーク上で Control loop を実装した場合は，実装したコンポーネントをプログラミングフレームワーク上にデプロイ，つまり配置することにより，システムを動作させる．

GCLD では，ソフトウェア進化時の開発プロセスも定義し，進化を実装するツール群も提供する．図 2.7 は，ソフトウェア進化時における開発手順を示したものである．以下，ソフトウェア進化時の各フェーズの概要について述べる．

1’. **ゴールモデルの変更：**要求の変化が生じた場合，開発者はまず，前回開発時に構築したゴールモデルに対して，変化分の要求を記述する．新たな要求があれば要求記述を追加し，要求の変更や削除があれば，同ゴールモデル上に記述されている該当部分を変更，削除する．要求の変化をゴールモデル上に反映した後に，再度整形プロセスに従った整形，詳細化を施す．

2’. **差分検出，コンフィギュレーション生成：**続いて，再整形したゴールモデルから，進化後のシステムに対するコンフィギュレーションを決定する．このコンフィギュレーションの決定は，ゴールモデルコンパイラにより自動化されているが，同コンパイラは，進化前のゴールモデルも入力とすることで，進化前後のゴールモデルの差分を検出し差分情報を出力する．また，生成するコンフィギュレーションに対しても構成を比較し，差分情報を出力する．

3’. **差分** Control loop **設計，実装：**ゴールモデルコンパイラが出力する差分情報を利用して，追加，変更の必要がある Control loop を設計，実装する．

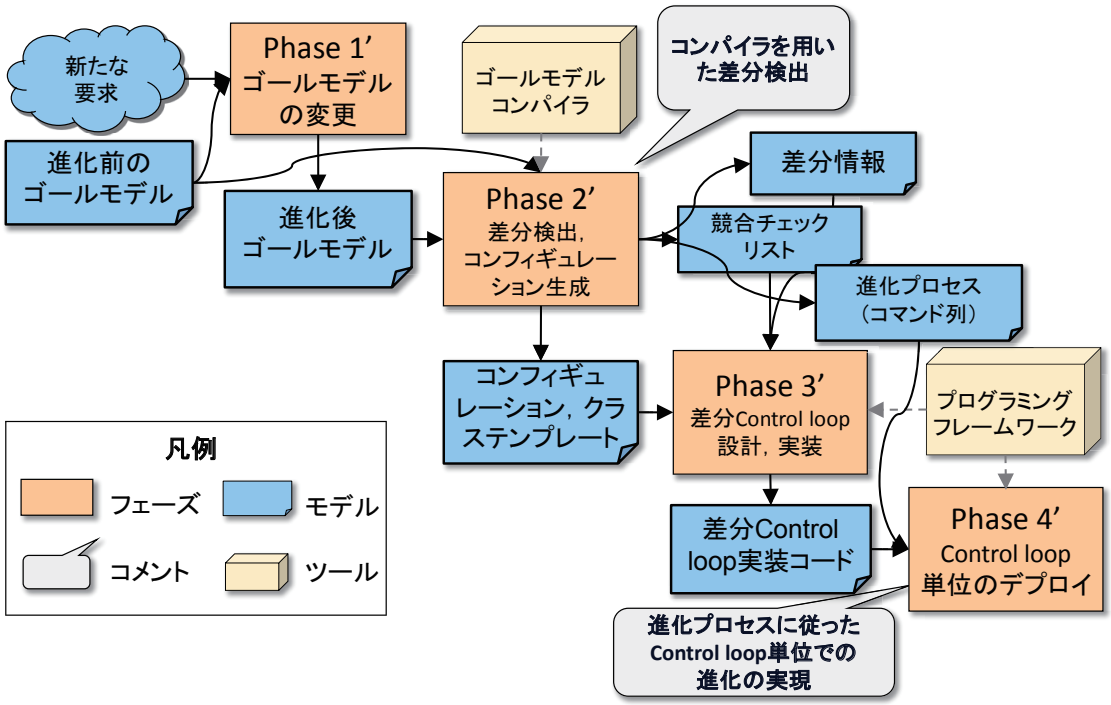

図 2.7. GCLD におけるソフトウェア開発（進化時）

4'.Control loop **単位のデプロイ：**新たに追加，変更実装した Control loop をシステムに配備する．本研究で導入するプログラミングフレームワークにおいては，Control loop をロード，アンロードするデプロイコマンドも提供しているため，同プログラミングフレームワーク上でシステムを動作させている場合は，実行時の動的な Control loop のデプロイも可能である．

2.4 GCLD で扱うモデル

GCLD において構築・生成されるモデルは，大きく，要求記述，コンフィギュレーション，実装コードに分類される．以下，本研究において扱う各モデルについて説明する．

要求記述（ゴールモデル）

本研究では，要求間の関係を明示的に定義することが可能であるゴール指向要求分析法に従った記述スタイル，つまりゴールモデルを要求記述として利用する．ゴールモデルでは，システムに対する要求，つまりゴールが明示的に記述され，ゴールはゴールを達成するためのサ

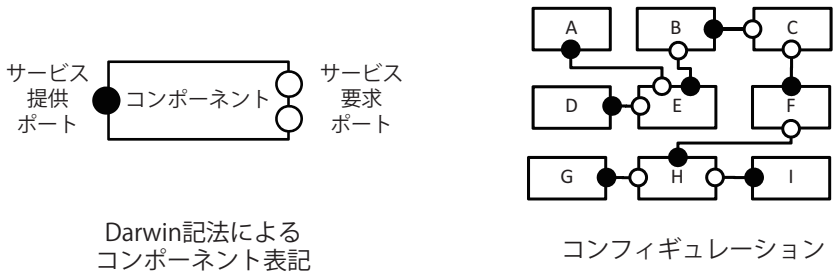

図 2.8. 本研究におけるコンフィギュレーション表記

ブゴールに詳細化される．従って，ゴール間の関係や，ゴールを達成するための中間状態が記載されたモデルであると言える．本研究では特に，ゴールとオペレーションやエンティティといった設計モデルとの関係が記述可能であり，構造化されたテキストデータである XML へのデータ出力機能を持つ KAOS をゴールモデルの基本的な記述文法として用いる．KAOS については，3.2.1 節で述べる．

なお，本研究では，整形プロセスを導入することにより，KAOS のゴールモデルをシステムの構造を決定可能なモデルへと詳細化する．この整形プロセスにおいて，KAOS の文法を一部拡張しているため，最終的に構築されるゴールモデルは拡張された KAOS モデルであると言える．

コンフィギュレーション

ソフトウェアシステムの構造を表現する設計モデルとして，本研究では複数のコンポーネントとそれらの接続関係により定義される**コンフィギュレーション**を用いる．本研究では特に，ソフトウェアシステムの進化を容易にするために Control loop をシステムの構成要素として考え，複数のコンポーネント群により実現される Control loop がシステム内に 1 つ以上定義されるようなコンフィギュレーションを扱う．コンフィギュレーション上のコンポーネントを表現するモデルとしては，コンポーネントの責務を可視化することのできる Darwin モデル [3] を用いる．図 2.8 の左図は Dawrin 記法に従ったコンポーネント表記であり，右図は Dawrin モデルにより記述されたコンフィギュレーションの一例である．Darwin におけるコンポーネントは，それぞれサービス提供の責務を持ち，必要に応じて他のコンポーネントが提供するサービスを要求・利用することで自身の責務を達成する．Darwin においては，このようなコンポーネント間のサービス要求・提供関係により，ソフトウェアシステムの構造を定義する．

本研究では，整形された KAOS モデルからコンフィギュレーションを決定するアルゴリズムと，アルゴリズムを実装したコンパイラを導入する．このコンフィギュレーション決定アル

ゴリズムにおいては，ゴールモデル上に記述されたゴール間の接続関係と，ゴールとエンティティとの関連記述から，ゴールモデル内に存在する Control loop パターンを検出し，実装すべき複数の Control loop を，構成するコンポーネントとコンポーネント間接続によりコンフィギュレーション上に記述する．コンフィギュレーションの詳細と決定アルゴリズムについては，3 章で述べる．

実装コード

本研究では，ソフトウェアシステムの構成要素を Control loop により実現する．本研究のように複数の Control loop によりシステムを構成する場合，並行動作を考慮して，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ミング言語であることが望ましい．本研究においては，マルチスレッドプログラミングをサポートする代表的な言語の一つといえる Java 言語を実装用のプログラミング言語として想定する．

本研究において各 Control loop は，コンフィギュレーション上では Control loop のアクティビティに応じた複数のコンポーネントにより構成される．開発者は各アクティビティに対応するコンポーネントをどのような粒度で実装するかを決定し，Control loop を実装する．実装する各クラスは，コンフィギュレーション上に定義されている各コンポーネントと一対一対応する場合もあれば，複数のコンポーネントを 1 つのクラスとして実装する場合もある．

特に本研究で導入するプログラミングフレームワークを用いて実装する場合には，各コンポーネントを Control loop の各アクティビティに対する基本機能を備えたスーパークラスを継承することで実装することができる．プログラミングフレークワークを利用した実装コードの構築法については，4 章で述べる．

2.5 例題：シミュレータ上での清掃ロボット構築

本研究で導入する開発プロセスを説明するために，3 章以降では，進化するシステムの開発例としてシミュレータ上で動作する清掃ロボットの構築を取り上げ，提案する開発プロセスを論じる．本節では，この清掃ロボットの例題について説明する．

2.5.1 シミュレータ仕様

図 2.9 は清掃ロボットシミュレータの GUI である．シミュレータの主な特徴は以下の通りである．

- シミュレータはフィールド情報を読み込むことで，フィードの状態をキャンバスに描画

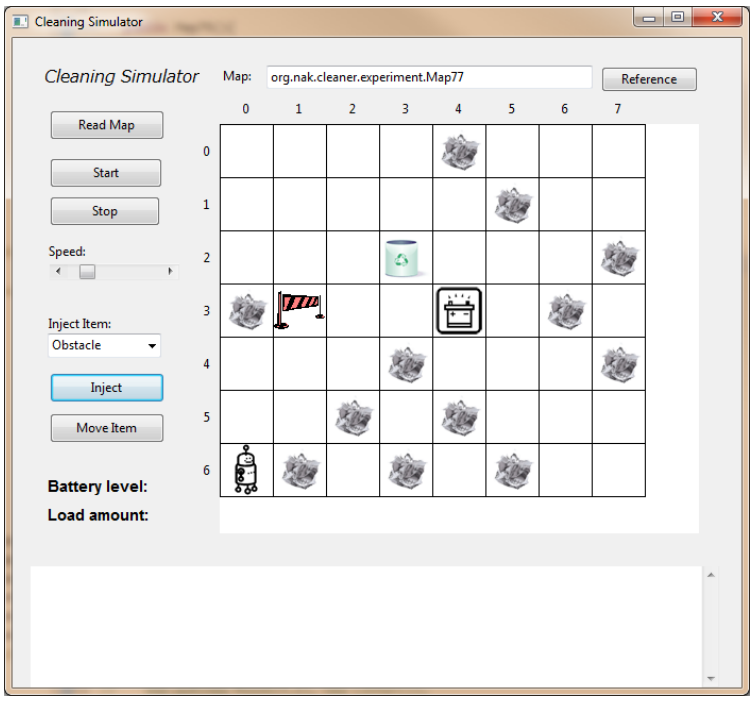

図 2.9. 清掃ロボットシミュレータ

することができる．フィールドの状態は x-y 平面上に表現され，フィールドには各種ごみやバッテリーステーションなどが配置されている．清掃ロボットもフィールド上の一地点に存在している．

- シミュレータは清掃ロボットの動作により，フィールド情報を変更することができる．ここでの変更とは，清掃ロボットの移動や，ごみ清掃によるごみの削除，下記に述べるイベントによるごみなどの追加，移動を指す．
- フィールド上には以下のオブジェクトを配置することができる．
  - **ごみ：**清掃ロボットの清掃対象となるオブジェクトである．ごみは Pile（山），Litter（散乱したもの），Can（缶）などの形状をもつ．図 2.9 中の座標 (4, 0) などに存在するオブジェクトは，Litter の形状をした紙くずを指す．
  - **バッテリーステーション：**清掃ロボットのバッテリーが充電可能なステーションである．清掃ロボットはバッテリーステーションと同一の座標に到達し，充電アクションを実行することにより，バッテリーを充電することができる．図 2.9 中の座標 (4, 3) に位置するオブジェクトが，バッテリーステーションである．以降，バッテリーステーションのことを，単にステーションと呼ぶ．
- シミュレータの GUI からフィールドに対して以下のイベントを発生させることがで

きる．

- **オブジェクトの投入：** “Inject Item” リストでオブジェクトを選択し，“Inject” ボタンを押した後に，フィールド上の一地点を選択することで，フィールド上の選択した座標に，指定したオブジェクトを新たに投入することができる．
- **オブジェクトの移動：** “Move Item” ボタンを押した後にフィールド上のオブジェクトをドラッグすることで，オブジェクトの位置を変更することができる．

- シミュレータの GUI 上で，清掃ロボットのバッテリー残量とごみの積載量を確認することができる．

また，シミュレータ上における清掃ロボットの制約は以下の通りである．なお，図 2.9 中の座標 (0, 6) に位置するオブジェクトが，清掃ロボットである．

- ロボットは，隣接するマス（座標）に対して，1 マスずつ移動できる．
- ロボットは，現在地からもっとも近くにあるごみ，あるいはステーションの座標を検出することができる．
- ロボットは，缶型，山型，ステーション型などの対象物の形状を認識することができる．
- 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上のごみを清掃することができる．
- ロボットが所持するバッテリーは最大容量が 100 であり，初期状態では最大容量まで充電されている．バッテリーは，1 マスの移動に 4，1 回のごみ清掃アクションの実行に 4 消費する．

2.5.2 清掃ロボットに対する要求

前節で示したシミュレータ上において構築する清掃ロボットに対する要求を以下のように定義する．

［清掃ロボットに対する初期要求］清掃ロボットはフィールド上のごみを検知して清掃することができる．ロボットは形状からごみを認識し，その形状から適切な清掃方法を選択する．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上からごみを清掃し，清掃後はごみを積載する．また，ロボットはバッテリー残量が少なくなると，ステーションを発見し，ステーションに移動後，バッテリーを充電することができる．

これらの機能により，ステーションで随時バッテリーを充電しながら，フィールド上のすべてのごみを清掃することができる．

本例題では，上記の初期要求に従って開発された清掃ロボットに対して，以降の 2 つの進化要求が，順次与えられると想定する．

進化 1（ごみ積載量管理機能の追加）：適用フィールドの拡大により，本清掃ロボットに対して以下の要求変化が生じる．

［清掃ロボットに対する要求変化 1］フィールド上のごみの増加により，清掃ロボットのごみ積載量では一度に全てのごみが清掃できない場合が生じることとなった．そこで，清掃ロボットがごみの積載量を管理できる機能を新たに追加したい．清掃ロボットはごみの積載量を監視し，積載量が一定値以上になった場合，ごみ箱の場所を探し，ごみ箱の場所へ移動後，積載しているごみをごみ箱に捨てるという動作が求められる．

ここで，図 2.9 のシミュレータ上の座標 (3, 2) に位置するオブジェクトが，ごみ箱である．

進化 2（障害物回避機能の追加）：続いて，フィールド上に回避して移動しなければならない障害物が追加されたと想定する．シミュレータ上では，図 2.9 中の座標 (1, 3) に位置するオブジェクトが障害物を示す．

［清掃ロボットに対する要求変化 2］フィールド上にごみではない，通過不能であるオブジェクト（障害物）が設置される状況が生じることとなった．そこで，清掃ロボットには進行方向に障害物が存在する場合，迂回して目的の座標に到達する機能が求められることとなった．

本論文では，次章以降，このような清掃ロボットの構築事例を例として，提案する開発プロセスについて論じる．

2.6　まとめ

本章では，ソフトウェア進化に対応するための本研究におけるアプローチについて述べ，本研究で導入する開発プロセス GCLD の概要について説明した．GCLD においては，特に要求分析フェーズに着目し，ゴール指向要求分析法により記述されるゴールモデルを用いた開発プロセスを定義する．また，システム開発を通じて，システムの構成要素を Control loop として定義，構築することによるソフトウェア進化を実現する．

次章以降では，GCLD の各開発フェーズにおいて，提案手法がどのようにソフトウェア進化に対応するかを詳細に論じていく．

第 3 章

ゴール指向要求記述を用いた Control loop の同定

3.1 はじめに

2 章で述べたように，本研究では，進化の影響が分析可能であり，進化を容易に実現できるソフトウェア開発プロセスとして，システムの構成要素を Control loop の観点から同定する開発手法を検討する．Control loop の概念に基づいてシステムを構成する場合，まず，システムが持つべき Control loop が明確に同定されていなければならない．また，各 Control loop が協調して動作するには，これらの関係が明示され，Control loop 間で生じ得る競合についても事前に把握されていなければならない．

本研究では，進化の要因となる要求変化と，機能実現に対する責務を同時に可視化，分析するのは要求分析時が最も適していると考え，要求記述を利用したシステム構成の決定法を提案する．図 3.1 は，本研究における，システム構成要素としての Control loop 同定のイメージを示したものである．本研究ではまず，要求をゴール指向要求記述（ゴールモデル）上に記述し，本研究で導入する整形プロセスを経ることで，システムを構成するために必要な Control loop をゴールモデル上で同定する．その後，システムの構成要素として同定した Control loop を，システム構成を表現するコンフィギュレーション上に配備する．

本章においては，まず，ゴールモデル上で Control loop を同定するためのゴールモデル整形プロセスについて論じる．その後，ゴールモデルからシステム構成を導出するコンフィギュレーション決定法と，これらを用いたソフトウェア開発法について言及する．

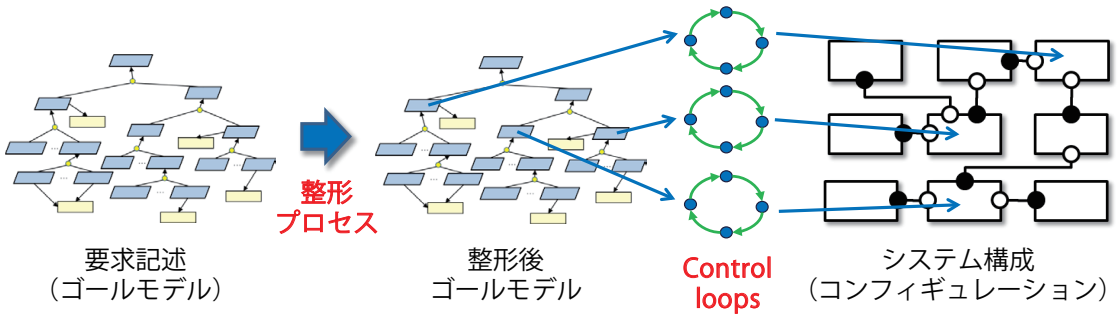

図 3.1. Control loop の位置づけ

3.2　ゴールモデルの構築

本手法ではまず，システムに対する要求をゴールモデル上で分析し，得られたゴールモデルを整形することで，進化による変更や拡張の範囲の限定化を目的とし，Control loop の観点からシステム構成要素を同定するためのゴールモデルへと洗練化させる．本節では，本研究で要求記述として用いる KAOS について説明し，その後，本手法で導入する整形プロセスについて述べる．

3.2.1　KAOS

KAOS[38, 39] は Lamsweerde らによって提案されたゴール指向要求分析法であり，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，要求のどの範囲をシステム化すべきかの判断を助けるとともに，詳細化の段階で要求間の矛盾の発見や解消を実現するなど，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．

図 3.2 は KAOS ゴールモデルの記述例である．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，つまり**ゴール**を単一アクタが実現できる大きさにまで細分化する．この細分化には，すべての達成が必要なサブゴールへと詳細化する **AND-refinement** と，いずれかの達成が必要なサブゴールへと詳細化する **OR-refinement** の 2 種類がある．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．KAOS ではアクタを**エージェント**（*Agent*）により表現し，構築すべきシステムを表現するエージェントには要件を割り当て，その他の外部環境アクタを表現するエージェントには期待を割り当てる．例えば図 3.2 の例では，要件を割り当てられている Agent1 は構築すべきシステムを，期待を割り

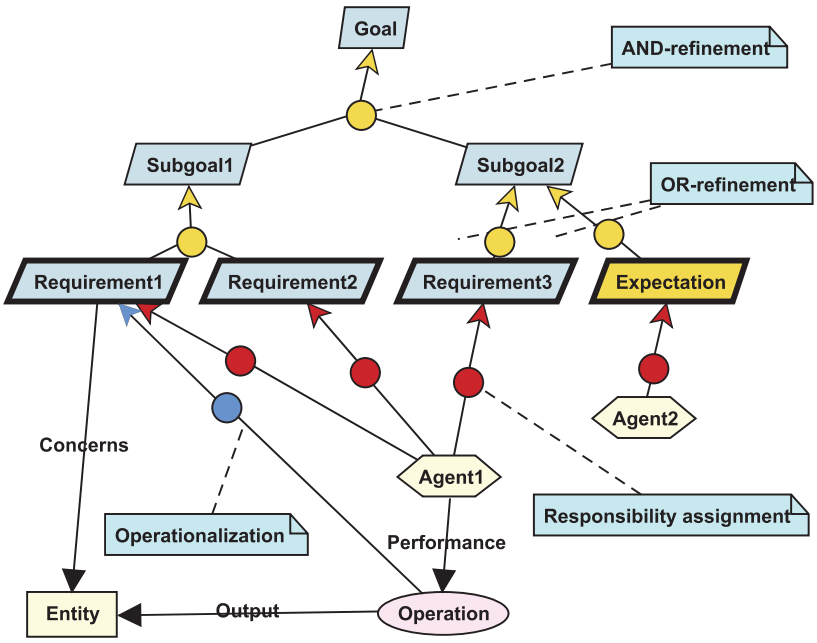

図 3.2. KAOS ゴールモデルの記述例

当てられている Agent2 は外部環境アクタを示すこととなる．ここで，要件，期待はゴールの一種であり，本研究では以降，特に断らない限り，ゴールと要件とを総称してゴールと呼ぶ．

KAOS では，これらの要求を分析・記述するモデル要素の他に，エージェントへの責務割り当て（*Responsibility assignment*）や，ゴールとエンティティとの関連（*Concerns*）などを定義するためのリンク，つまり関係線を提供している．また，ゴールが表現する状態を達成するための動作を**オペレーション**として定義し（*Operationalization*），ゴールと関連付けることも可能である．このように，KAOS においては，ゴールモデルをもとに分析を進めることで，システムに実装すべき操作を抽出・記述することも可能であり，ゴール以外の様々なモデル要素を図形表現で記述することが可能である．

KAOS はこれらの図形表現モデルを提供する一方で，各モデル要素を形式的に定義するための論理表現も提供している．この論理表現には時相論理に基づいた記述が想定されており，KAOS では既に，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 [40, 39, 41] が提供されている．例えば Darimont らは，文献 [40] 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 ($P \Rightarrow \Diamond \neg Q$)

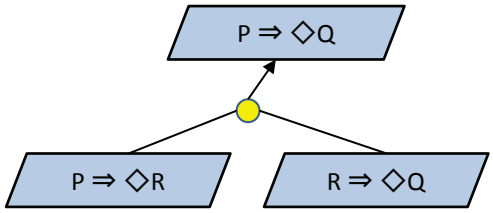

図 3.3. Milestone-driven パターン

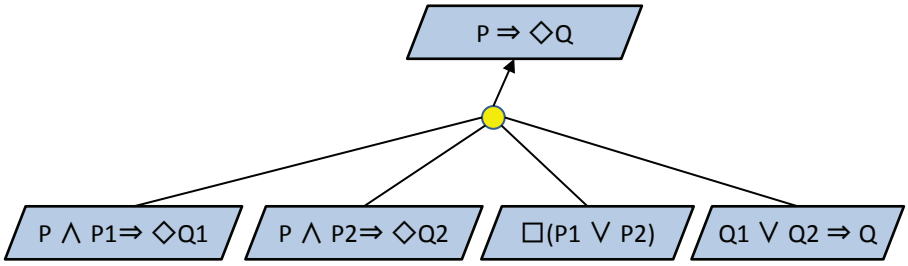

図 3.4. Case-driven パターン

- **Maintain goals:** 常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:** 常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，◇ は将来のいずれかの時点で真となることを，□ は今後常に真であることを表わ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．

一般にゴール指向要求分析法においては，ゴール分解（詳細化）の難しさが指摘されているが，KAOS においては，ゴールの詳細化を支援するために各種の詳細化パターンも提案されている．代表的な詳細化パターンとして，ゴール達成のために経由しなければならない状態群への分解を促す Milestone-driven パターン [42]（図 3.3）や，ゴールを実現するための事前条件を分解することで，場合分けによるゴールの詳細化を促す Case-driven パターン [40]（図 3.4）などが知られている．

KAOS はこのように，ゴール間の階層構造が記述可能な図形表現だけでなく，背後に論理表現も併せ持つもつことで要求の系統的な分析を支援する要求分析法であり，また，エンティティやオペレーションなどのシステム設計の概念も記述できることから，本研究では，要求記述に KAOS のゴールモデルを利用し，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 Control loop の同定とコンフィギュレーションの構築に活用する．

3.3 ゴールモデル整形プロセス

本研究では，進化に有効なシステム構成をゴールモデルから決定することを目的として，ゴールモデルの整形プロセスを導入する．図 3.5 は，本研究で導入するゴールモデル整形プロセスの概要を示したものである．本研究では，ゴールモデル上で Control loop を抽出・同定するために，以下の手順で，ゴールモデルを整形する．

定義 3.3.1 **ゴールモデル整形プロセスの流れ**

1. **エンティティの追加：**ゴールモデルにエンティティを追加し，ゴールとの関係を定義する．このゴールとエンティティとの関係は，以降のステップにおいて，Control loop を構築すべきゴール群の同定と，Control loop 間の競合検出において利用する．
2. **共通ゴールの集約：**共通ゴールを集約し，分離して記述するとともに，ゴール間の依存関係に基づいた階層構造を明確化する．
3. **主要ゴールの同定：**ガイドラインに従って，主要ゴールとなり得るゴール群を抽出し，それらの候補群から主要ゴールに該当するゴール群を同定する．
4. **Control loop の構築：**各主要ゴールが Control loop を形成するようにゴールモデルを構造化する．構築した各 Control loop をシステムに責務として割り当てる．
5. **競合の検出：**ゴールモデル上で Control loop 間に競合が発生し得るかを判定し，必要に応じてゴールモデルを再整形する．

以降，整形プロセスにおける各ステップの内容を，図 3.6 に示す清掃ロボットに対するゴールモデルを例に，手順に沿って説明する．

3.3.1 初期ゴールモデル

本研究では，まず従来の KAOS 分析に従って，開発対象システムに対するゴールモデルを構築するが，整形プロセス適用前のゴールモデルが満たすべき要件として，以下を定義する．

定義 3.3.2 **初期ゴールモデルが満たすべき要件要件：** システムに対する機能要求が，すべてゴールモデル上にゴールとして記述されている □

これは，整形プロセス適用前のゴールモデル上で要求がすでに十分に詳細化されていることを前提とするためものであり，整形プロセスでは，このようなシステムに実装すべき機能を表現した要求，つまり機能要求をもとに Control loop の同定を進める．機能要求は，通常，ゴー

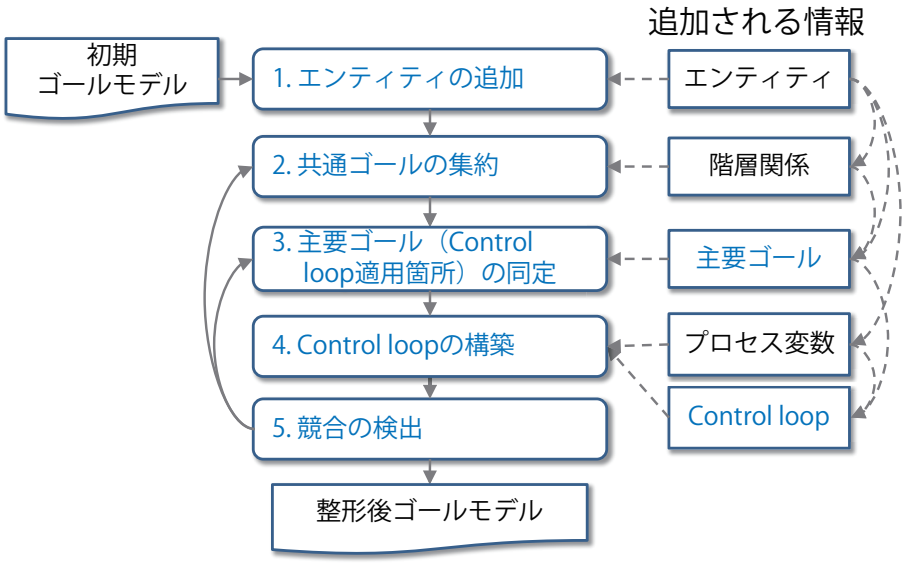

図 3.5. ゴールモデル整形プロセスの概要

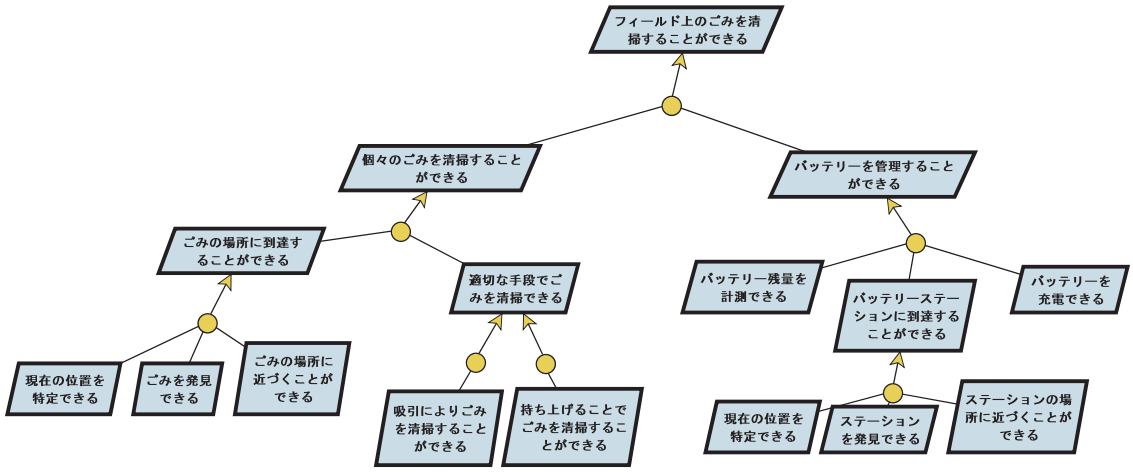

図 3.6. 整形プロセス適用前の清掃ロボットに対するゴールモデル

ルモデルの末端の葉ノードに該当するゴールとして記述され，例えば，図 3.6 の清掃ロボットの例では，ゴール「現在の位置を特定できる」,「ごみを発見できる」などが機能要求に該当する．

3.3.2 エンティティの追加

整形プロセスでは，まず最初に，Control loop を形成させるべきゴールを発見することを目的として，初期ゴールモデルに対して各ゴールに関連するエンティティを追加し，ゴールとエンティティとを Concerns 関係により関連付ける．従来の KAOS 分析においては，ゴールモデルを記述後，システムに関連するオブジェクトを導出するために，各ゴールに関与するエンティティを記述するが，本研究では，ゴールの共通化・集約のためにエンティティとの関係，つまり Concerns 関係を利用する．また，本研究では，このエンティティとの関係を Control loop 同定のためだけでなく，Control loop 間の競合検出にも利用する．

Example 1 — 以降，整形プロセスの適用法を，清掃ロボットシミュレータに対するゴールモデルを用いて説明する．図 3.6 の清掃ロボットのゴールモデルに対しては，例えば，ゴール「現在の位置を特定できる」については，エンティティ「現在地の座標」が関連エンティティとして考えられる．また，ゴール「バッテリー残量を計測できる」に対しては，計測対象となる「バッテリー残量」が関連エンティティとして考えられる．従って，これらのエンティティを定義し，各ゴールと Concerns 関係により関連付ける．図 3.7 はエンティティ追加後のゴールモデルを示したものである．

以降，本研究では，ゴールモデル内に定義されている Concerns 関係を，ゴール *goal*，エンティティ *ent* の関係 $(goal, ent)$ を元に持つ集合 *Concerns* により定義し，$(goal, ent)$ が集合 *Concerns* に含まれている時に真となる述語 *concerns* を導入する．

$$(goal, ent) \in Concerns \iff concerns(goal, ent)$$

また，本研究では以降，複数のサブゴールとその親ゴールが関与すると判断したエンティティに対しては，親ゴールに Concerns 関係を集約して記述するものとする．例えば，図 3.7 のゴールモデルにおけるエンティティ「ごみ」は，ごみ清掃に関与するすべてのゴールに関与するため，上位ゴールである「個々のごみを清掃することができる」との関係に集約して記述されていると解釈できる．

3.3.3 共通ゴールの集約

ソフトウェアシステムの進化を考えた場合，要求の変化が実装コードに及ぼす影響を考慮する必要がある．つまり，追加される機能や変更が必要な各機能に対してそれぞれ独立に変更要求を記述すると，同様の機能や共通の機能に関する要求記述が分散化，冗長化する可能性があ

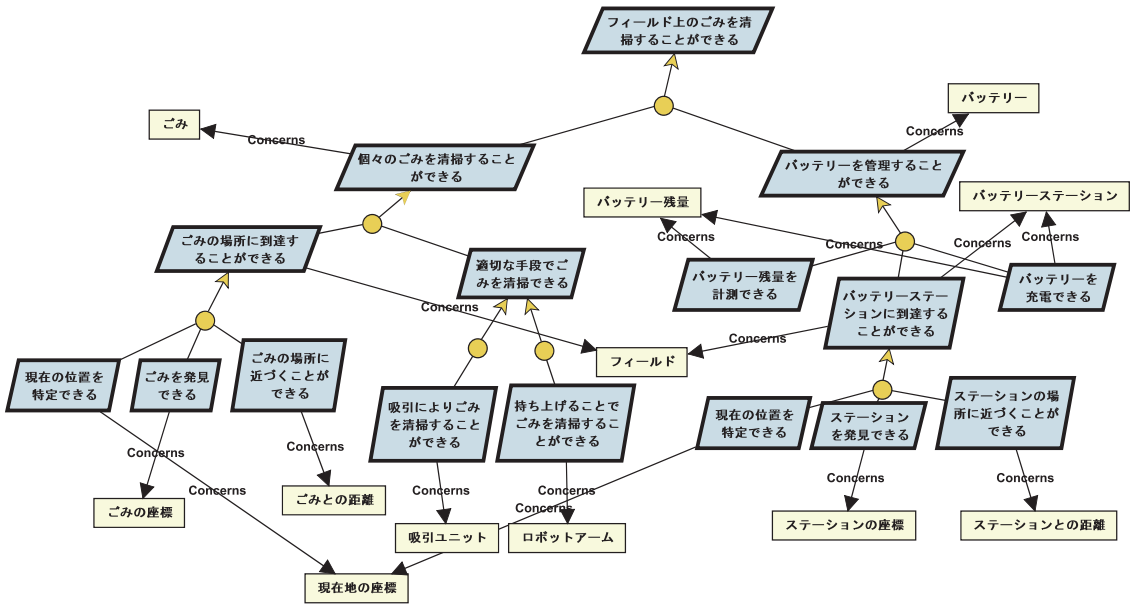

図 3.7. エンティティを追加したゴールモデル

る．例えば，清掃ロボットの例では，ごみ清掃機能やバッテリー管理機能においては，ごみやバッテリーステーションといった対象に向かって移動する機能や，これらの対象を発見する機能が必要となるが，ごみ清掃やバッテリー管理に対する記述内でそれぞれの要求を記述をすると，記述が分散化，冗長化することとなる．

従って提案する整形プロセスでは，複数箇所に出現する可能性のあるゴールの冗長な定義を避けるために，これらのゴールを集約して共通化する．ゴールを完全に集約，つまり集約先に移動可能である場合は移動させ，移動できない，つまり移動により親ゴールの達成に明示的に影響を及ぼす場合は，本研究で導入する “Uses” ラベルを用いて，抽出された共通ゴールへの依存関係を記述する．後者の場合は，例えばゴール A の達成にゴール B の達成が必要であり，かつ，他の機能要求の達成を必要としない場合に，ゴール A に対して “Uses B” ラベルを付与する．共通ゴール集約の手順を Algorithm 1 に示す．開発者は Algorithm 1 に従って，ゴールモデルのルートノードから再帰的に共通ゴールとなり得るゴールであるかどうかを検査することで，共通ゴールを分離させる．ここで，共通ゴールとして抽出すべきかどうかの判断には，達成すべきゴール，つまり実現すべき機能の類似性や，エンティティとの関係の類似性を用いる．

Example 2　— 図 3.7 のエンティティ追加後の清掃ロボットゴールモデルの例では，ごみを発見してごみの場所まで移動することにより達成されるゴール「ごみの場所に到達することが

Algorithm 1 共通ゴールの集約：*aggregateCommonGoals(n)*

[入力] *n:* 検査対象のゴール

```
1:  for all childGoal in n.childrenList do
2:      if childGoal が共通ゴールとして抽出できる then
3:          if 対応する共通ゴール commonGoal がまだ定義されていない then
4:              commonGoal ← generalize(childGoal); // generalize(): ゴールの記述を一般化する処理
5:          end if
6:          commonGoal.addAllDescendent(generalize(childGoal.allDescendent));
            // 新たに作成した共通ゴール (commonGoal) に childGoal のサブゴール, 子孫のゴールを
            // 一般化して移動する
7:          commonGoalRoot.childrenList.add(commonGoal); // commonGoalRoot: 共通ゴールの集約先
8:          childGoal.removeAllDescendent();
9:          if childGoal の移動が親ゴールの達成に影響しない then
10:             n.removeChild(childGoal);
11:         else
12:             childGoal.putLabel("Uses" commonGoal.getName());
13:         end if
14:     end if
15:     aggregateCommonGoal(childGoal);
16: end for
```

できる」と，バッテリーステーションを発見して到達することにより達成されるゴール「バッテリーステーションに到達することができる」の 2 つのゴールを，対象物への到達という観点から集約可能と判断することができる．従って，これらを共通ゴール「目標物に到達することができる」として集約し，サブゴールも含めて記述を一般化する．その一方で，これらの 2 つのゴールは，ごみ清掃やバッテリー充電を達成するために経由すべきゴール（状態）であり，必要不可避であるため，サブゴールを除いて 2 つのゴールは残し，共通ゴールに対する “Uses” ラベルを付与する．共通ゴール集約後のゴールモデルを図 3.8 に示す．図 3.8 においては，ゴール「ごみの場所に到達することができる」,「バッテリーステーションに到達することができる」を残し，共通ゴールとして集約されたゴール「目標物に到達することができる」への依存関係を示す，“Uses 目標物に到達することができる” ラベルを付与している．

本研究では，この “Uses” ラベルを，Control loop 間の依存関係を表現する情報として利用する．また，“Uses” ラベルを付与したゴールは，以降のプロセスにおいて機能要求とみなす．

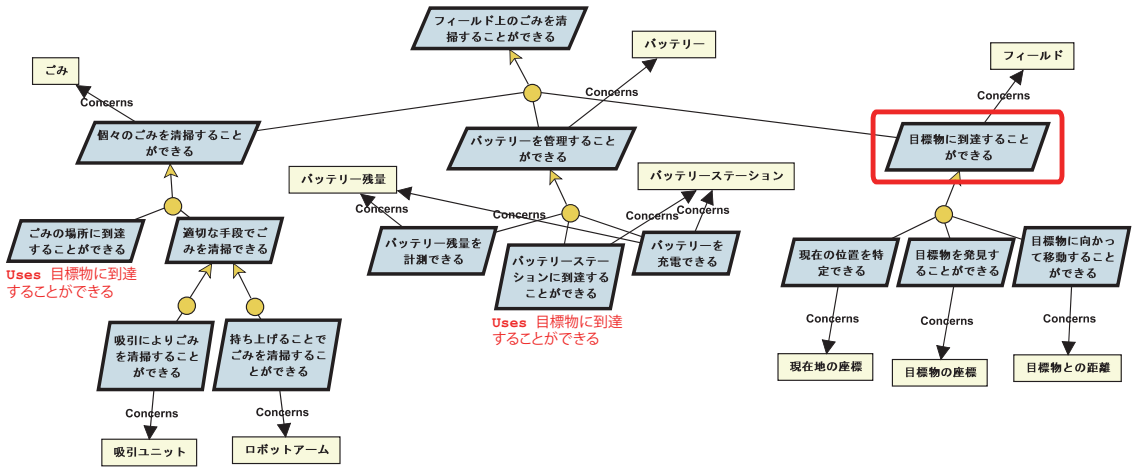

図 3.8. 共通ゴール集約後のゴールモデル

3.3.4　主要ゴールの同定

共通ゴールが分離して記述されると，続いて，ゴールモデル上に記述されたゴール群から**主要ゴール（Prime goals）**となり得るゴールの集合を同定する．主要ゴールとは，ゴールの中で，特にシステムが持つべき機能により達成が明示的に期待されているゴールであり，本研究においては，ソフトウェアの進化・変更を考慮して，固有の Control loop を割り当てる対象とするゴールを指す．主要ゴール単位で Control loop を割当て，これを動作の単位とすることで，進化時に他の主要ゴール，つまり他 Control loop により構成される他のシステム構成要素との依存関係を Control loop 間の関係のみに限定することが可能になる．

主要ゴール候補の同定

本研究においては，まず，主要ゴールの候補となり得るゴールを同定し，その後，主要ゴール候補のいくつかを選択し，充足性を判定することで，主要ゴール集合を決定する．定義 3.3.3 は，本研究で導入する，主要ゴール候補同定のためのガイドラインである．

定義 3.3.3　主要ゴール候補同定のガイドライン

以下のいずれかの指針を満たすゴール g_i を主要ゴール候補の集合 *PGoalCand* の元とする．ここで，$Children_{g_i}$ はゴール g_i を親とするゴールの集合であり，また，$uses(g_x, g_y)$ はゴール g_x の達成にゴール g_y の達成が必要である（ゴール g_x 上に “Uses g_y” が定義されている）ときに真となる述語である．

- **指針 1：** Uses ラベルにより参照されているゴールである

$$\exists g_i\ (\exists g_j\ uses(g_j, g_i)) \Rightarrow\ g_i \in PGoalCand$$

- **指針 2：** 複数のサブゴールが同一エンティティと Concerns 関係にある

$$\exists g_i\ (\exists g_{ik}\ \exists g_{il}(g_{ik}, g_{il} \in Children_{g_i} \wedge \exists ent(ent \in Entity \wedge concerns(g_{ik}, ent) \wedge concerns(g_{il}, ent)))) \Rightarrow\ g_i \in PGoalCand \qquad \square$$

定義 3.3.3 で示した 2 つの指針は，主要ゴールの可能性のあるゴールを決定するためのものである．まず，指針 1「Uses ラベルにより参照されているゴールである」は，共通ゴールを主要ゴールとして同定するためのものである．共通ゴールは，ゴールモデル上で共通の機能を集約したゴールであり，システムが提供すべき複数の機能の実現に不可避のゴールと判断できることから，主要ゴールとして抽出すべきゴールであるといえる．

一方の指針 2「複数のサブゴールが同一エンティティと Concerns 関係にある」については，Control loop の単位で関心事や操作変数を集約化させるためのものである．指針 2 においては，既に定義されているゴールとエンティティとの関連に着目し，エンティティとの関連をサブツリー内で包含できるようなゴールを主要ゴールの候補として同定する．このような同定は，複数の Control loop が同一の操作変数を扱うことを避ける効果がある．

Example 3 — 図 3.8 のゴールモデルでは，まず，ゴール「目標物に到達することができる」は，他ゴールから Uses 関係によって利用されているゴールであるため，指針 1 を満たす．また，指針 2 の「複数のサブゴールが同一エンティティと Concerns 関係にある」については，ゴール「個々のごみを清掃することができる」がエンティティ「ごみ」に対して合致し，ゴール「バッテリーを管理することができる」がエンティティ「バッテリー」,「バッテリーステーション」,「バッテリー残量」に対して，さらにゴール「目標物に到達することができる」がエンティティ「フィールド」に対して合致することが分かる．従って，清掃ロボットのゴールモデルでは，図 3.9 に示すように，ゴール「個々のごみを清掃することができる」,「バッテリーを管理することができる」,「目標物に到達することができる」を主要ゴール候補として同定することができる．

充足性判定

本研究では，ゴールモデル上の主要ゴールに Control loop を割り当てることで，ソフトウェアの構成要素を決定する．提案する整形プロセスでは，Control loop を割り当てる主要ゴールの集合がソフトウェアの構成要素として十分であるかを判断するために，主要ゴール集合の同定に対する充足性判定法を導入する．定義 3.3.4 に主要ゴール同定に対する充足性の判定法を

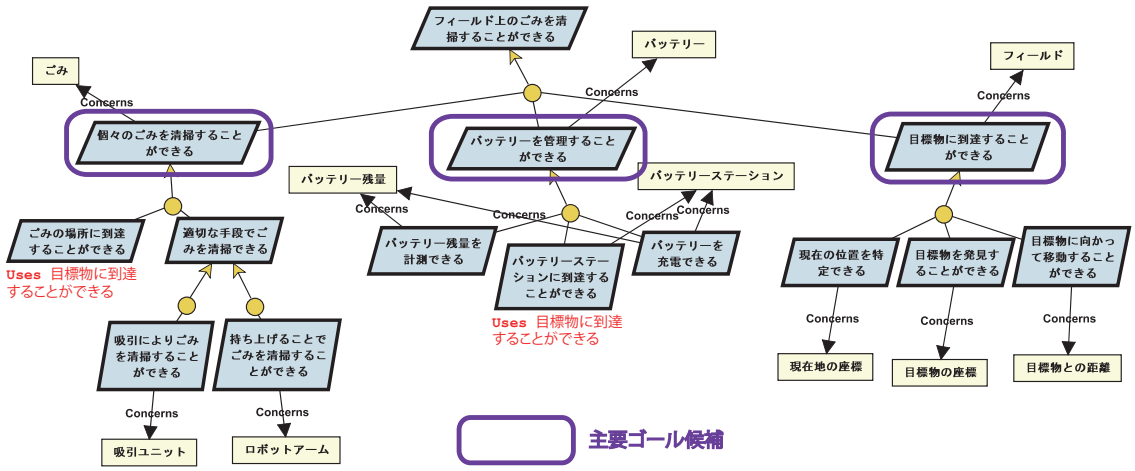

図 3.9. 主要ゴール同定後のゴールモデル

示す.

定義 3.3.4　**充足性の判定**

主要ゴール候補のいくつかを選択し，下記の条件 1，条件 2 がいずれも満たされるとき，ゴールモデルが主要ゴールの同定に関して充足されているという．また，このとき，選択された主要ゴール候補を主要ゴールとする．ここで，*FuncReq* は機能要求の集合であり，*PGoal* ($\subseteq$ *PGoalCand*) は主要ゴール候補から主要ゴールとして選択したゴールの集合，G_i はゴール g_i を根とするツリーに含まれるゴールの集合である．

- **条件 1：**すべての機能要求がいずれかの主要ゴールを根とするツリーに包含されている

$$\forall g(g \in FuncReq \wedge \exists i(g_i \in PGoal \wedge g \in G_i))$$

- **条件 2：**いずれの主要ゴールも，他の主要ゴールを根とするツリーに包含されていない

$$\forall i \forall j \ (g_i, g_j \in PGoal \wedge i \neq j \Rightarrow g_i \notin G_j \wedge g_j \notin G_i) \qquad \square$$

ここで，主要ゴールは Control loop を割り当てられるゴールであることから，条件 1 はすべての機能要求がいずれかの Control loop に包含されていることを，条件 2 は Control loop が他の Control loop を包含していないことを保証するものである．

この充足性判定においては，主要ゴール候補群から主要ゴールの集合を定め，各条件が満足されるかどうかを検査する．まず，選択した主要ゴール集合に対して，条件 1，条件 2 が満足されるかどうかを判定する．すべての主要ゴール候補の組合せにおいて条件 1 を満足しない場合は，いずれの主要ゴール候補にも含まれていない機能要求が存在することとなる．この場合，該当する機能要求，あるいはこれらを包含するゴールを主要ゴール候補として追加し，充

足性を再度判定する．

Example 4 — 図 3.9 の清掃ロボットのゴールモデルにおいては，主要ゴール候補集合から，「個々のごみを清掃することができる」,「バッテリーを管理することができる」,「目標物に到達することができる」のすべてを選択することで，条件 1，条件 2 を満足する主要ゴール集合を得ることができる．

3.3.5 Control loop の構築

充足性判定条件を満たす主要ゴールが同定されると，各主要ゴールに対して Control loop を割り当てる．本ステップでは，各主要ゴールのサブゴールを Control loop の観点から整理・詳細化することで，Control loop 設計のための情報を抽出する．ゴールモデルの意味論に照らし合わせると，各主要ゴールに Control loop を対応付け，Control loop に関連する状態群をサブゴールとして記述することは，Control loop の概念に基づいて主要ゴールを達成するということを意味しており，また，Control loop が主要ゴールとして記述されたゴールの達成に対する責務を持つことを意味している．本研究では，このサブゴールの抽出，つまり主要ゴールにおける Control loop に関するアクティビティの抽出のために，ゴール記述パターンである**Control loop パターン**を導入する．

Control loop パターン

本研究で導入する Control loop パターンを図 3.10 に示す．Control loop パターンはツリー構造で表現され，Control loop のアクティビティに対応する 3 種類のゴールタイプをツリー上に明示的に記述するものである．Control loop パターンの根には主要ゴールが対応付けられ，Control loop のアクティビティにおける Analyze と Decide の責務を割り当てる（Analyze & Decide タイプゴールとして同定する）．Analyze & Decide タイプゴールのサブゴールは Collect タイプと Act タイプに分類される．入力変数の扱いに関するゴールは Collect タイプに該当し，振舞いによる状態達成を表現したゴールは Act タイプに分類される．

図 3.11 の上図は Control loop パターンの一例を形式的に記述したものである．ここで，P, Q はそれぞれ，主要ゴールに対する事前条件，主要ゴールがいずれ満たすべき状態を，R_j は $Q\ (= R_m)$ を満たすために達成しなければならない状態（マイルストーン）を表す．また，M_i は入力変数が取り得る値に対応する状態であり，S_{ji} は状態 M_i におけるシステムの振舞いである．$\Diamond$ は将来のいずれかの時点で真となることを，$\Box$ は今後常に真であることを表わす時相論理演算子である．Control loop パターンの根に該当するゴールは先に述べたとおり Analyze

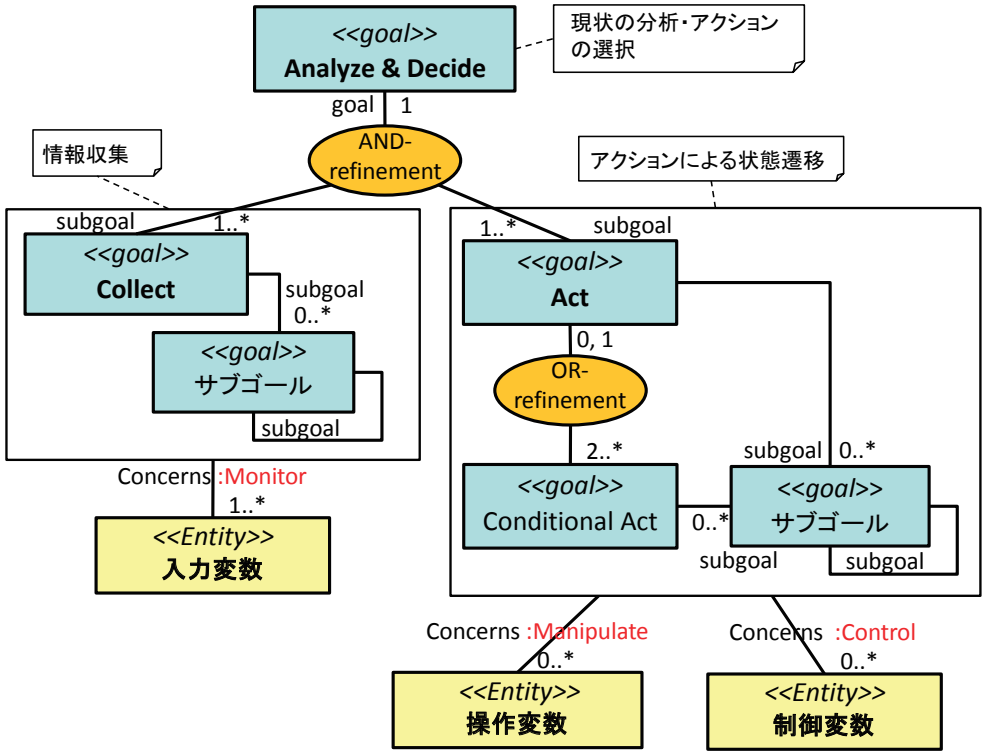

図 3.10. Control loop パターン

& Decide タイプに該当し，“$P \Rightarrow \Diamond Q$” は，主要ゴールの達成を意味している．サブゴールのうち，入力変数に関与し，入力変数の現在の値が同定されていることを表現するゴールは Collect タイプに該当し，“$\Box(M_1 \vee \ldots \vee M_n)$” で示される．一方，Act タイプのゴールは，入力変数の値と現在の目標達成状態，選択された振舞いによって次の状態に遷移することを表すゴール “$R_j \wedge (\vee_{i=1\ldots n}(M_i \wedge S_{j+1_i})) \Rightarrow \Diamond R_{j+1}$” により表現することができる[*1]．

本ステップでは，Control loop パターンの根のゴールに各主要ゴールを対応付け，Control loop パターンのゴール分割に従って，主要ゴールのサブゴールをリファクタリング，つまり整理する．図 3.11 の下図は，Control loop パターンにおける各ゴールの役割を Control loop の各アクティビティに対応付けて示したものである．まず，主要ゴールは，Control loop における分析（Analyze）と決定（Decide）の責務を持ち，現状の分析と次に取るべきアクションの選択や，アクションの制御の役割を担う．

ここで，Control loop を構築する場合，Control loop が扱うプロセス変数も明確化する必要がある．2 章で述べたとおり，Control loop には，Control loop への入力情報となる入力変数，

[*1] ここで，$0 \leq j \leq m$ となる m が存在し，$R_0 = P$，$R_m = Q$ である．

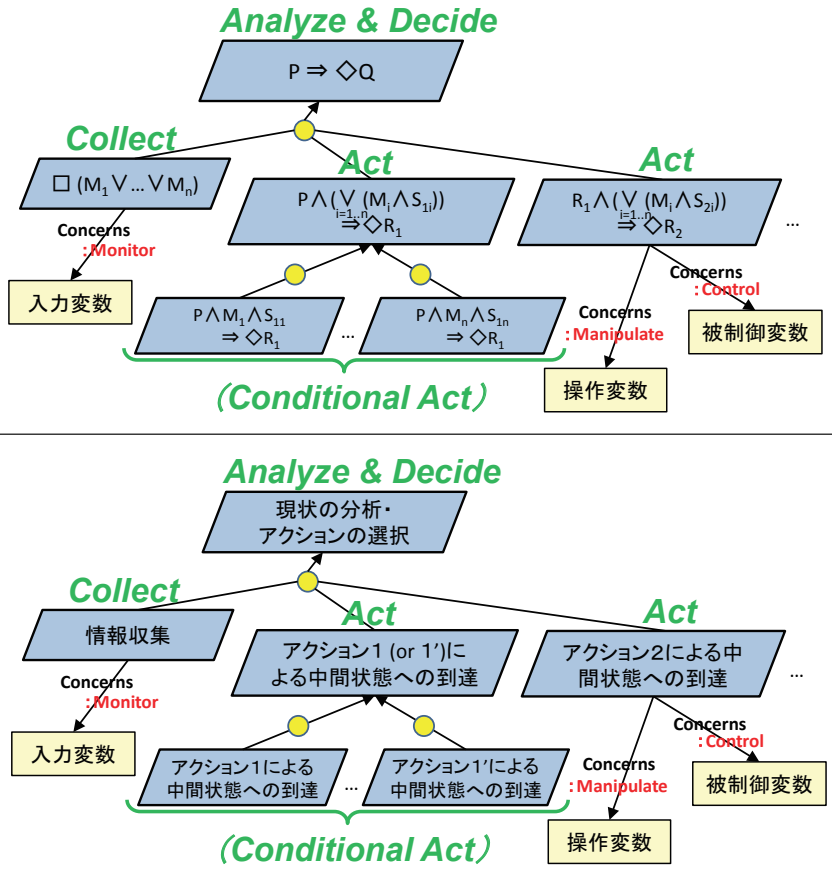

図 3.11. Control loop パターンの一例（上図：形式的記述，下図：各ゴールの役割）

Control loop の制御対象となる制御変数，Control loop が変更（操作）可能な操作変数の 3 種類のプロセス変数が存在する．既に述べたように，Analyze & Decide タイプゴールは 2 種類のサブゴールにより詳細化されるが，情報収集の責務を持つ Collect タイプゴールは，現在の状態を同定することが可能なエンティティである入力変数と関連付けられる．もう一方の，アクション実行による状態遷移を責務として持つ Act タイプゴールは，必要に応じて，決定結果からアクションを制御するために与えられる操作変数と，主要ゴール達成のために変更可能である制御変数を表現するエンティティに関連付けられる．このように，Control loop パターンに従ったゴール整形においては，プロセス変数を同定しながら，Control loop の各アクティビティに該当するゴールを明確化する．

ここで，Control loop の各アクティビティとゴールタイプとの対応関係の妥当性について議論する．まず，Collect と Act は環境に関与するアクティビティであるのに対し，Analyze, Decide はシステム内部で閉じたアクティビティである．また，Analyze, Decide がゴールモデ

ル上で状態としての明示的な表現が難しいアクティビティであるのに対して，Collect, Act が明示的に示されていれば，Control loop の明示的な役割を同定することができると考えられる．従って，Collect タイプと Act タイプのゴールを集約，つまりサブゴールとして持つゴールに Control loop を割り当てることと，加えて，同ゴールが Collect，Act タイプゴールの達成状態を利用するゴールであることから，同ゴールに Analyze と Decide の役割を割り当てることは妥当であると考えられる．また，プロセスコントロールモデルにおける制御部と処理部の役割を考えると，Analyze と Decide は制御部の役割となり，システム上では同一コンポーネントとして実装される場合も多い．この設計モデルとの対応関係の観点からも，Analyze と Decide の役割を同一ゴールに割り当てることは妥当であると考えられる．

Collect タイプの決定

Control loop パターンの適用にあたっては，まず，主要ゴールが Analyze & Decide タイプに該当することから，主要ゴールのサブゴールとして Collect タイプを同定するところから始める．ここで，Collect を実現，つまり Collect タイプのゴールを達成するためには，入力変数が必要になる．入力変数とは Control loop に対しての入力となる情報であり，Control loop に割り当てられるゴールを達成，あるいはゴールの達成を判断するために収集する必要のある情報である．従って，ゴールモデル上においては，まず，入力変数がエンティティとして定義されている必要がある．もし入力変数が定義されていなければ，新たに入力変数をゴールモデル上に定義する．

次に，入力変数を収集するゴールが既に主要ゴールのサブゴールとして記述されているかどうかを確認し，もし記述されていなければサブゴールとして新たに定義する．最後に，このサブゴール，つまり Collect タイプのゴールと入力変数との間に Concerns 関係を定義する．KAOS の Concerns 関係は，単にゴールとエンティティとの関係を定義するに過ぎないため，本研究では，どのような関係であるかを分類可能な属性情報を Concerns 関係に付与する．表 3.1 は Concerns 関係を詳細化するラベルの一覧である．Collect ゴールと入力変数間の Concerns 関係に対しては，`：Monitor` のラベルを付与する．

Example 5　— 図 3.12 は，図 3.9 で示した清掃ロボットのゴールモデルに対して，主要ゴール「個々のごみを清掃することができる」に対応する Control loop の Collect タイプの決定過程を示したものである．図 3.9 においては，同主要ゴールを詳細化したゴール群に Collect タイプに該当するゴールと入力変数は存在しない．この主要ゴールを達成するためには，ごみの清掃手段を決定するために「ごみの形状」と，清掃の成否を判断するために「清掃結果」に関する情報が必要であると考えられる．従って，ここでは，入力変数として「ごみの形状」と「清

表 3.1. Concerns 関係詳細化ラベル

ラベル	ゴールタイプ	エンティティ	Concerns 関係の意味付け
`:Monitor`	Collect	入力変数	入力変数の値を決定する情報収集
`:Manipulate`	Act	操作変数	操作変数の変更に関与するアクション
`:Control`	Act	制御変数	実行結果により，制御変数を変更させる可能性のあるアクション

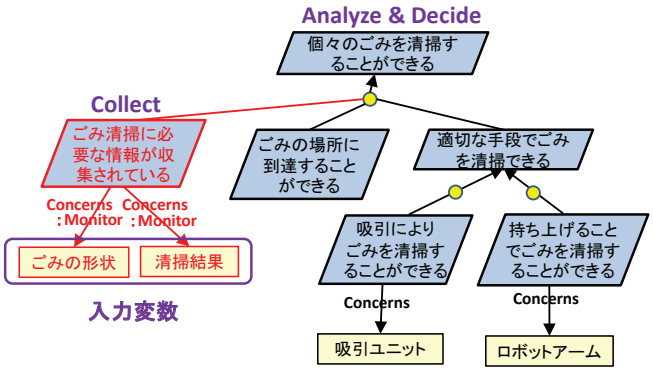

図 3.12. Collet タイプ，入力変数の追加

掃結果」を表現するエンティティを追加し，これらを収集するためのゴール「ごみ清掃に必要な情報が収集されている」を Collect タイプのゴールとして追加する．さらにこれらの入力変数と Collect タイプゴールを，`：Monitor` ラベルの付与された Concerns 関係で関連付ける．

図 3.13 は，図 3.9 で示した清掃ロボットのゴールモデルに対して，すべての Control loop の Collect タイプ決定後のゴールモデルである．主要ゴール「バッテリーを管理することができる」に対しては，「バッテリー残量」がゴール達成の判断に必要な入力情報となり，主要ゴール「目標物に到達することができる」については，「現在地の座標」と「目標物の座標」に関する情報が，目標物へ到達するために必要であると考えられる．これらはいずれもすでにエンティティとして記述されているため，ここでは既に定義されている各ゴール–エンティティ間の Concerns 関係に対して，`：Monitor` ラベルを付与する．

Act タイプの決定

Collect タイプのゴールと入力変数が同定されると，次に，状態変化を実現するアクションを同定する．これらはゴールモデル上ではアクション実行による状態遷移を責務とする Act タイプゴールとして表現される．もし，複数の振舞いにより Act タイプゴールが達成できる場

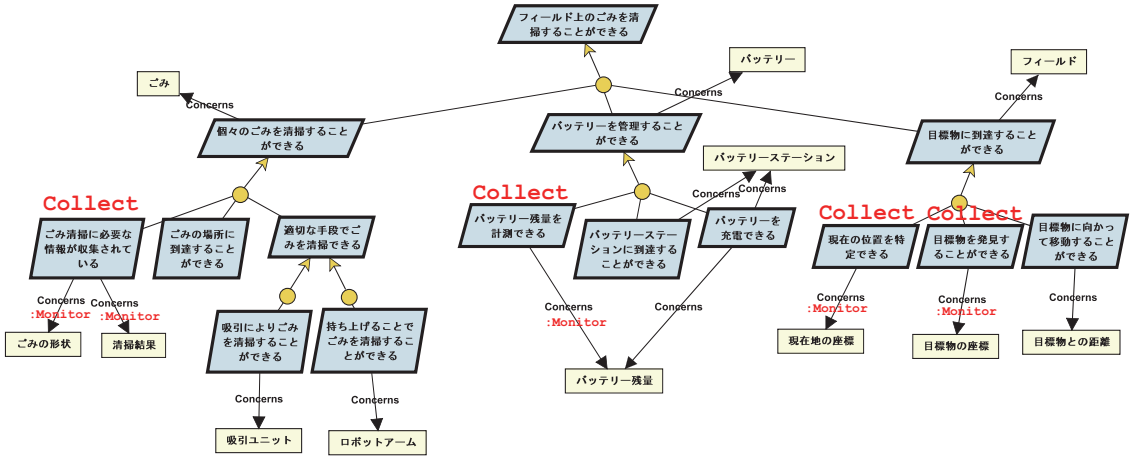

図 3.13. Collect タイプ同定後のゴールモデル

合は，OR-refinement を用いて複数の振舞いに該当するゴール（図 3.11 において *(Conditional Act)* とラベル付けされたゴール）を記述する．主要ゴールのサブゴールとして Act タイプゴールが記述されていない場合は追加する．

Act タイプゴールが同定されると，Control loop のプロセス変数である制御変数，操作変数を同定する．各 Act タイプゴールが表現するアクションを実行するにあたり，変更可能，つまり操作可能なエンティティがあれば，操作変数として定義し，該当する Act タイプゴールに Concerns 関係を定義する[*2]．また，Control loop の制御対象となる制御変数をエンティティとして定義し，制御変数を変更させる可能性のあるアクションに該当する Act タイプゴールに Concerns 関係を定義する[*3]．Control loop は通常，操作変数，制御変数を持つが，提案手法においてはゴール間の利用関係，つまり Control loop の利用関係も考慮するため，Uses ラベルを持つ Control loop においては，関連する操作変数や制御変数が被利用側の Control loop 内に記述される場合もある．

Example 6　— 図 3.14 は，主要ゴール「個々のごみを清掃することができる」に対応する Control loop の Act タイプゴール決定過程を示したものである．同主要ゴールに対しては，まず，既に同定した Collect タイプゴール「ごみ清掃に必要な情報が収集されている」を除くサブゴールであるゴール「ごみの場所に到達することができる」と「適切な手段でごみを清掃することができる」が Act タイプのゴールであるかを判断する．この場合はいずれもアクションにより達成できる状態を表現しているため，Act タイプゴールとして同定する．続いて，操作

[*2] 表 3.1 に基づいて，Concerns 関係に`:Manipulate` ラベルを付与する．
[*3] 同様に，`:Control` ラベルを付与する．

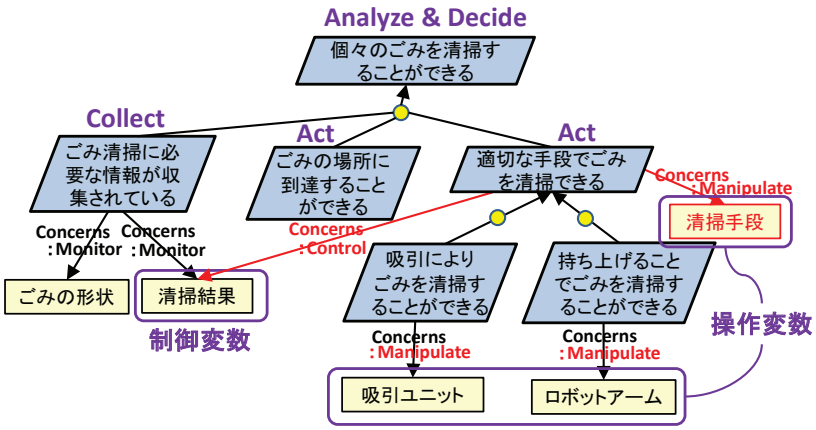

図 3.14. Act タイプ，操作変数，制御変数の同定

変数と制御変数を同定するが，この例では，前者のゴール「ごみの場所に到達することができる」については，Uses ラベルにより，他の主要ゴール，つまり Control loop を利用していることから，ここでは操作変数は必要ないと判断できる．一方，後者のゴール「適切な手段でごみを清掃することができる」については，操作変数として，ごみの清掃方法を選択する「清掃手段」が同定され，そのサブゴールにはそれぞれ「吸引ユニット」と「ロボットアーム」が操作変数として同定される．また，本 Control loop の制御変数としては，清掃による成否を表わす「清掃結果」が考えられるが，清掃結果はごみの清掃を実施することにより値が変化するため，後者のゴールに関連付ける．

図 3.15 は，清掃ロボットのゴールモデルに対して，すべての Control loop の Act タイプゴールを同定した結果である．主要ゴール「バッテリーを管理することができる」に対しては，ゴール「バッテリーステーションに到達することができる」とゴール「バッテリーを充電できる」が Act タイプのゴールとなり，後者に関しては，制御変数として「バッテリー残量」が同定される．一方，主要ゴール「目標物に到達することができる」については，ゴール「目標物に向かって移動することができる」が Act タイプのゴールである．同 Act タイプゴールに関する操作変数として「移動方向」を新たに追加し，また，目標物との距離は移動結果に伴い変化するため，「目標物との距離」を制御変数として追加し，同 Act タイプゴールに関連付ける．

Uses の詳細化

Control loop 内のプロセス変数が同定されると，続いて，Control loop 間で受け渡しをすべき変数を同定する．本整形プロセスでは，構築された Control loop に対して，3.3.3 節の共通ゴール集約時に定義した Uses 関係を詳細化させる．まず，利用側 Control loop が被利用側

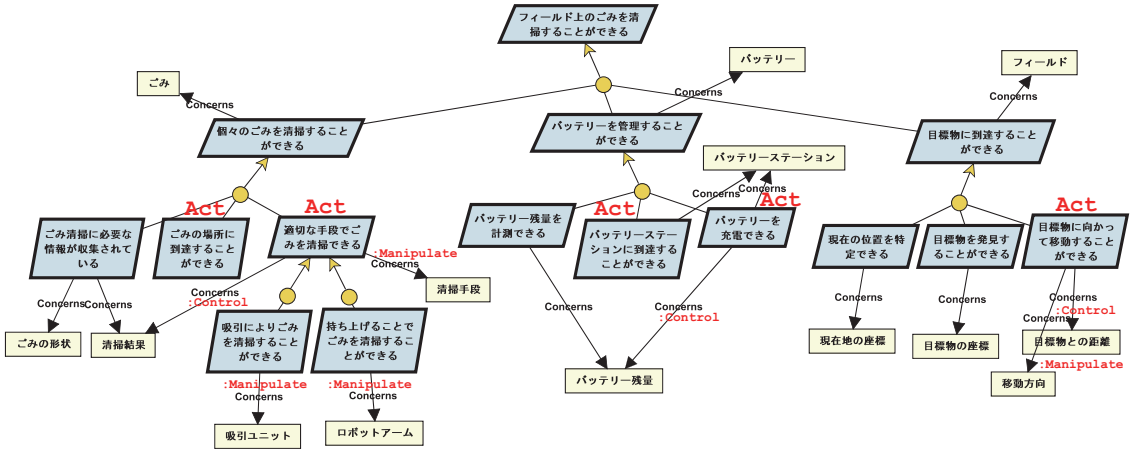

図 3.15. Act タイプ同定後のゴールモデル

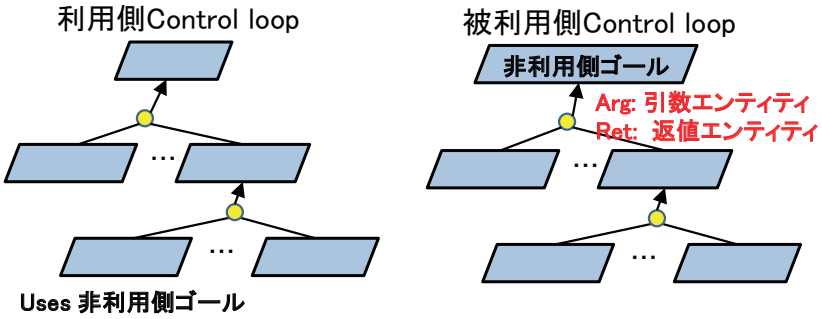

図 3.16. Uses の詳細化

Control loop へ送信する変数（エンティティ）が存在する場合は，被利用側 Control loop における `Arg` 属性の値として定義する．一方で，非利用側 Control loop が処理結果を利用側 Control loop に返す場合は，処理結果となるエンティティを被利用側 Control loop における `Ret` 属性の値として定義する（図 3.16）．

Example 7　— 図 3.17 は，清掃ロボットの例における詳細化された Uses 記述を示したものである．この例では，非利用側となる，目標物への到達を実現する Control loop が動作するためには，ゴミやバッテリーステーションなどの目標物が明示される必要があるため，被利用側 Control loop への引数として，「目標物タイプ」を記述する．一方で，被利用側 Control loop の提供サービスは発見した目標物への到達であるため，返すべき処理結果は不要であり，従って，この例では返値，つまり RET 属性の値は指定しない．

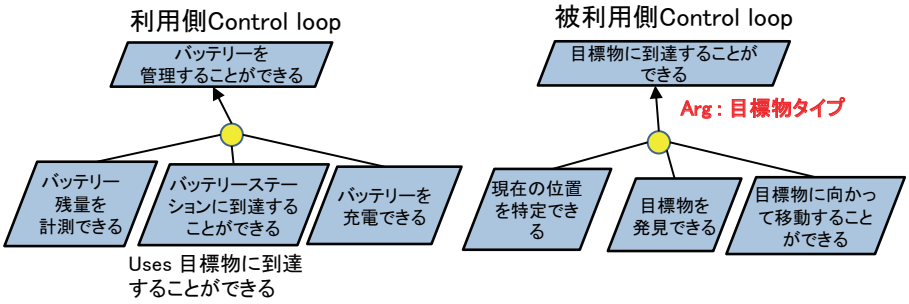

図 3.17. 清掃ロボットの例における Uses の詳細化

責務割り当て

通常，ゴールモデルでは，ゴールがシステムやユーザなど個々のアクタ（エージェント）に責務が割当てられる粒度にまで詳細化，つまり分解される．従って，KAOS においては，ゴールモデルの下層に位置する十分に分解されたゴールが機能要求として抽出され，システムに対する責務として各アクタに割り当てられる．例えは，図 3.15 の清掃ロボットの例では，ゴールモデルの下層に記述されているゴール「ごみ清掃に必要な情報が収集されている」や「ごみの場所に到達することができる」，「吸引によりごみを清掃することができる」などが，提供すべき機能として構築対象のシステムである清掃ロボットに割当てられることとなる．

しかしながらシステムの進化を考えた場合，単に個別の機能要求を抽出してシステムに割り当てただけでは機能間の依存関係の情報を抽出することができず，進化時の影響範囲を分析することができない．従って本研究では，システム，つまり KAOS におけるエージェントの責務割り当てを定義 3.3.5 のように拡張する．

定義 3.3.5 **システムに対する責務割当て：**システムを表現するエージェントには，機能要求ではなく，主要ゴールを割り当てる．

$$\forall pg(pg \in PGoal \Rightarrow \exists st(st \in Agents \wedge Responsibility(st,pg))) \qquad \square$$

システムに主要ゴールを責務として割り当てることは，Control loop を包含するサブツリーを責務として割り当てることを意味し，従って，システムに Control loop 単位での責務を割当てることに該当する．これは，ゴールモデル上でのシステム設計・実装の範囲を Control loop の単位で明示するものである．

Example 8 — 図 3.15 のゴールモデルでは，主要ゴールである「個々のごみを清掃することができる」，「バッテリーを管理することができる」，「目標物に到達することができる」の 3 つ

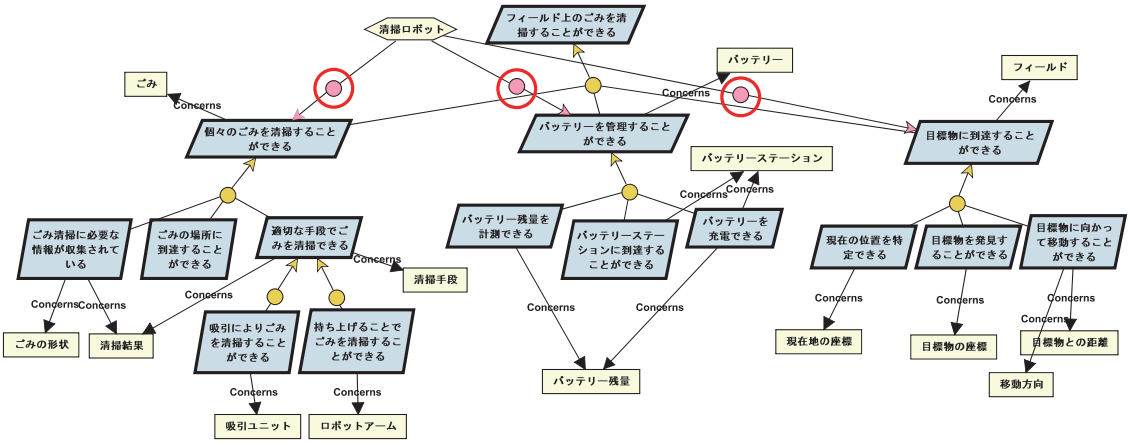

図 3.18. 責務の割り当て

のゴールを，清掃ロボットを表現するエージェントに割り当てる．責務を割り当てた後のゴールモデルを図 3.18 に示す．

3.3.6　競合の検出

複数の Control loop をシステムに配置した場合，Control loop 間で**競合**（conflict）が引き起こされる可能性がある．ここでいう競合とは，複数の Control loop が各自のゴールを達成するために共通変数の値を相互に変更する状態を指す．本研究では，ゴールモデルの構造から形式的に競合の可能性のある箇所を検出し，競合が存在する場合には，それを解消するようゴールモデルを整形する．以降，ゴールモデル上での競合の検出法と，競合が発生する場合の解消法について述べる．

エンティティに関する競合

本研究では，Control loop 間で生じる競合を検出する手段として，複数 Control loop が共通して値を変更する変数が存在するかどうかをゴールモデル上で検査する．このような検査に対しては，Control loop に関与するゴールを決定し，それらのゴールに対してエンティティとの関連をチェックすることで，形式的な検査が可能となる．本研究では，ゴールモデル上での形式的な競合検出を目的として，Entity-conflict パターンを定義する．

定義 3.3.6　Entity-conflict パターン

以下の条件を満たすエンティティ ent を，競合の可能性のあるエンティティとする．ここで，g_i および g_j は主要ゴールである．

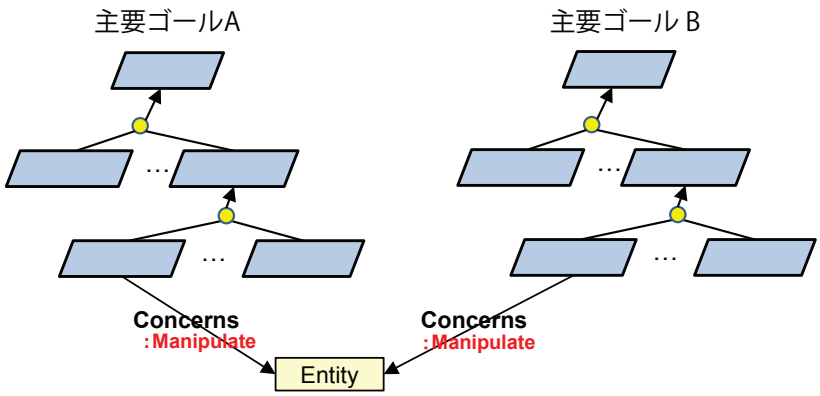

図 3.19. Entity-conflict パターンのイメージ

$$\exists i\, \exists j (g_i, g_j \in PGoal \wedge \exists ent (ent \in Entities \wedge \exists k\, \exists l (g_{ik} \in G_i \wedge g_{jl} \in G_j \wedge concerns(g_{ik}, ent, Manipulate) \wedge concerns(g_{jl}, ent, Manipulate))))) \quad \square$$

図 3.19 は Entity-conflict パターンを図示したものである．Entity-conflict パターンは，複数の主要ゴールを根とするサブツリーから，Manipulate 属性の `Concerns` 関係が定義されているエンティティを検出するものであり，各主要ゴールに Control loop を割当てた場合に，複数の Control loop に共通の操作変数が割り当てられる状態を示したものである．Entity-conflict パターンに合致する場合，次節で述べる解消法に基づいてゴールモデルを整形する．

競合の解消法

Entity-conflict パターンにより検出された競合をゴールモデル上で解消するには，エンティティとの関係を複数の主要ゴールを根とするサブツリーからではなく，唯一の主要ゴールからの関係に集約する必要がある．そこで，本研究では，Entity-conflict パターンにより検出された競合に対して，以下のいずれかの手段によりゴールモデルを整形する．

定義 3.3.7 ゴールモデル上での競合の解消法

Entity-conflict パターンにより検出された競合の解消法として，以下の 2 種類の手段を定義する．

- **解消法 1（共通ゴールの導入）：**エンティティに唯一関係するゴールを共通ゴールとして新たに定義する
- **解消法 2（ゴールの集約）：**競合する主要ゴールを 1 つのゴールツリーとして集約し，集約したゴールを新たな主要ゴール候補とする □

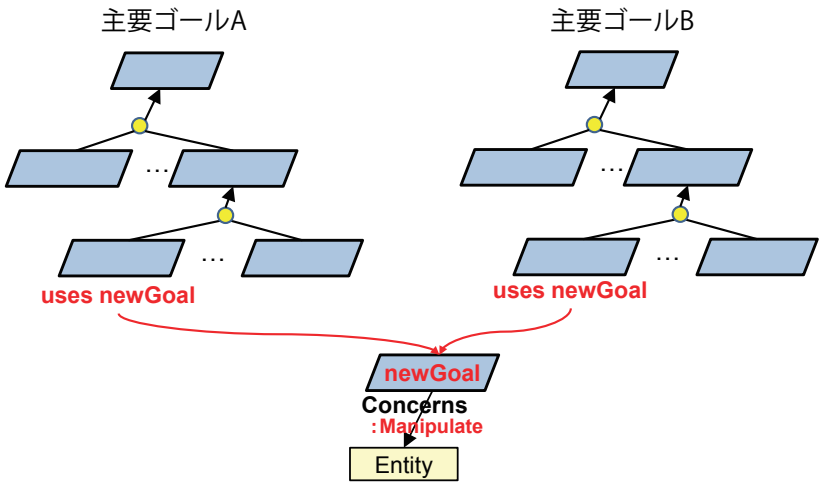

図 3.20. 解消法 1：共通ゴールの導入

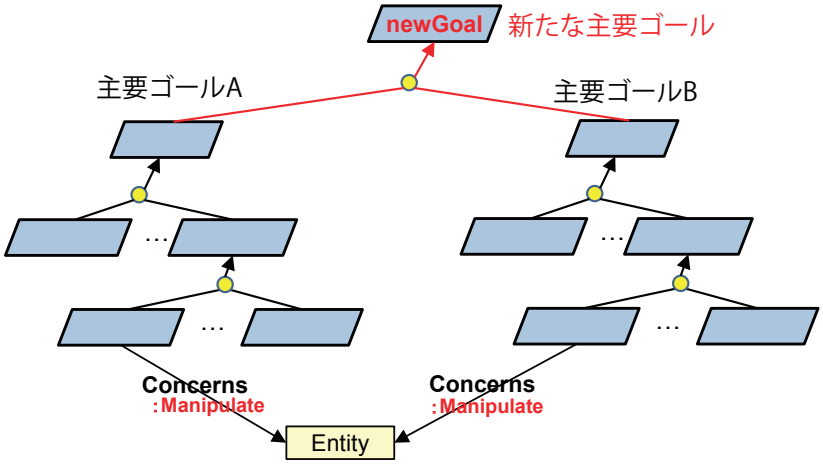

図 3.21. 解消法 2：ゴールの集約

解消法 1，解消法 2 のイメージをそれぞれ図 3.20，図 3.21 に示す．解消法 1 は，競合するエンティティに対して，アクセスの責務を持つ唯一のゴールを共通ゴールとして新たに導入するものであり，競合が発生した主要ゴールは新たに定義された共通ゴールをゴール達成のために利用する，つまり Uses ラベルにより共通ゴールを参照するように修正する．これは，共通ゴールの導入と同様の整形に該当し，共通のエンティティに対する関与を集約することで，Control loop からエンティティへのアクセスの責務集約を目的としたものである．

もう一方の解消法 2 に関しては，競合が発生している主要ゴール群を集約して，これらを包含する新たな主要ゴールを定義するというものである．この修正は，競合 Control loop 群を統

図 3.22. Darwin コンポーネントモデル [3]

合することで，共通エンティティへのアクセスを制御するものであり，新たな主要ゴール，つまり統合した Control loop がエンティティに対してのアクセスの責務を持つことになる．

いずれの解消法を利用するかは，各主要ゴールの意味を考慮する必要がある．解消法 1 は，独立した複数の主要ゴールが競合している場合の解消法であるのに対して，解消法 2 は，同様の目的を持つ主要ゴールを統合する場合に有効である．従って，考慮するべき競合が存在する場合は，各主要ゴールの持つ意味や目的を考慮して，いずれかの解消法を実施し，解消法の適用後は，新たな主要ゴール群に対して，再度充足性を判定し，主要ゴールを同定する．

本研究においては，以上の整形プロセスによりシステムに必要な Control loop を同定し，整形されたゴールモデルをシステム構成決定のための入力情報として用いる．

3.4 ゴール指向要求記述に基づいたコンフィギュレーション決定

一般に，システム開発プロセスにおいては，要求記述に記載された情報は後継フェーズにおいて開発すべきシステムの拠り所となる．本研究では，1.2 節で述べた開発プロセスに求められる要件を満たすために，前節までで定義した整形プロセスにより構築されたゴールモデルを用いたシステム構成（コンフィギュレーション）の決定法を導入する．システム構成を表現するモデルとしては，コンポーネントとその接続（コネクタ）により表現されるコンフィギュレーションを用い，特に，コンポーネントの責務を可視化することのできる Darwin モデル [3]（図 3.22）をコンフィギュレーションを表現するためのコンポーネントモデルとして用いる．

3.4.1 コンポーネントのタイプ

本研究で導入するコンフィギュレーション決定法では，Control loop の構成要素として，ゴールモデル上で定義した Collect タイプ，Analyze & Decide タイプ， Act タイプの 3 種類に対応するコンポーネントを抽出することとする．Collect タイプと Act タイプのコンポーネン

トを分離して抽出するのは，まず Collect タイプについては，情報収集においては定常的なモニタリングなど，Control loop の制御タイミングとは独立したサイクルでモニタリングを実行する場合があることと，ソフトウェア進化を考慮した場合，Control loop の制御の変更と情報収集メカニズムの変更のタイミングは必ずしも一致しないことによるものである．また，Act タイプに関しても，Collect タイプ同様にソフトウェア進化を考えた場合，Control loop の制御の変更と処理部の変更のタイミングは必ずしも一致しないと考えられる点，Shaw のプロセスコントロールモデルにおいても制御部と処理部が分離されている点から，独立したコンポーネントとして抽出する．一方で，通常は分析結果に対応してどのように制御するかを判断するため，分析（Analyze）と決定（Decide）は同一コンポーネントの責務として集約する．

3.4.2　コンフィギュレーションへの変換

本研究で導入するコンフィギュレーション決定法では，整形されたゴールモデルをもとに，ゴールモデル上のゴールをシステムコンフィギュレーション上のコンポーネントへと変換する．変換の対象となるゴールは，整形プロセスにより責務割り当てをした主要ゴール以下のゴール群，つまり Control loop を構成するゴール群となる．本手法においては，コンフィギュレーション変換に 2 つのアルゴリズムを用いる．まず Algorithm 2 により，ゴールモデル上のゴールのうち，Collect タイプ，Analyze & Decide タイプ， Act タイプ，および Conditional Act タイプに該当するゴールをコンポーネントに変換し，ゴール間の関連，つまり AND/OR-refinement リンクをコンポーネント間の取り得る接続関係に変換する．Algorithm 2 において，AND-refinement 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，一対一の複数の接続関係に変換する．一方の OR-refinement に対しては，親コンポーネントのサービス利用ポートに対して子コンポーネントのそれぞれのサービス提供ポートとを接続する．図 3.23 は AND-refinement と OR-refinement の変換イメージを示したものである．

Example 9　— 図 3.18 のゴールモデルを入力としたときに，Algorithm 2 を適用して生成される第一段階のコンフィギュレーションを図 3.24 に示す．このコンフィギュレーションでは，例えばコンポーネント「個々のごみを清掃することができる」と，その子コンポーネント「ごみ清掃に必要な情報が収集されている」,「適切な手段でごみを清掃できる」,「ごみの場所に到達することができる」との接続は AND-refinement リンクに対応した接続であり，コンポーネント「適切な手段でごみを清掃できる」と「吸引によりごみを清掃することができる」および「持ち上げることでごみを清掃することができる」との接続は，OR-refinement リンクに対応した接続である．

Algorithm 2 ゴールモデルからのコンフィギュレーション生成（第一段階）

[入力] *gModel*: ゴールモデル

[出力] *config*: （第一段階の）システムコンフィギュレーション

```
1: assignedGoals ← gModel 上でシステムに責務割り当てされているゴール群;
2: for all goal in assignedGoals do
3:     goal.putLabel("AD");
4:     comp ← config.Components.add(goal.getOperation());
       /* コンポーネントの生成 */
5:     if goal.refinement=AND-refinement then
6:         for all cGoal in goal.getChildren() do
7:             pPort ← comp.addRequiredPort();
8:             cComp ← config.Components.add(cGoal.getOperation());
9:             cPort ← cComp.addProvidedPort();
10:            config.connectPorts(pPort, cPort);
11:            if cGoal.hasConcerns(":Monitor") then
12:                cGoal.putLabel("C"); /* Collect タイプと同定 */
13:            else
14:                cGoal.putLabel("A"); /* Act タイプと同定 */
15:                if cGoal.refinement = OR-refinement then
16:                    pPort ← comp.addRequiredPort();
17:                    for all condGoal in cGoal.getChildren() do
18:                        condComp ← config.Components.add(condGoal. getOperation()); /* Conditional Act
                           タイプ */
19:                        condPort ← condComp.addProvidedPort();
20:                        config.connectPorts(cPort, condPort);
21:                    end for
22:                end if
23:            end if
24:        end for
25:    end if
26: end for
```

得られたコンフィギュレーションを Darwin モデルの意味論に照らし合わせると，AND-refinement リンクに該当する変換は，親ゴールに対応するコンポーネントが全ての子ゴールに対応するコンポーネントのサービスを要求，つまり利用することで，自身のサービスを提供することを意味している．また，OR-refinement リンクに該当する変換は，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いずれかのサービスを利用することで，自身のサービスを提供可能であることを意味している．従って，変換前後において，

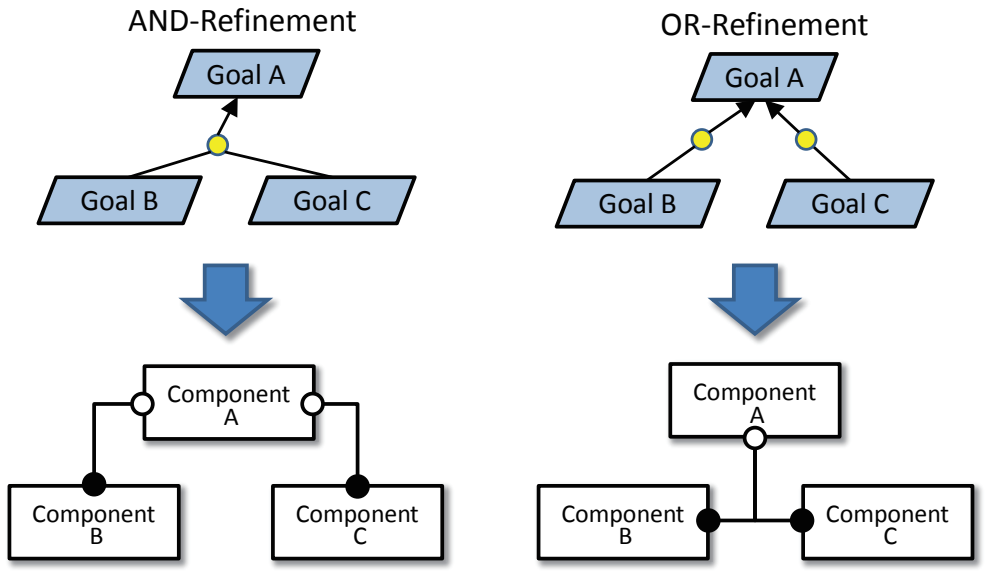

図 3.23. Algorithm 2 における AND/OR-refinement のコンフィギュレーションへの変換イメージ

ゴールモデルとコンフィギュレーション間での意味上での不整合は生じないと考えることができる．

ゴールモデルからコンフィギュレーションへの変換が終わると，生成したコンフィギュレーションを Algorithm 3 に従って洗練化する．本アルゴリズムは，`Uses` ラベルに従って，利用関係にあるコンポーネントを接続するとともに，Control loop における Analyze & Decide タイプのコンポーネントと，OR-refinement で複数記述されている Conditional Act タイプのコンポーネントとを直接接続するよう変更する．後者は，仲介している Act タイプのコンポーネントを削除することを意味する．

Example 10　— Algorithm 3 適用後のコンフィギュレーションを図 3.25 に示す．図 3.25 は，図 3.24 と比較して，コンポーネント「適切な手段でごみを清掃できる」が削除され，Analyze & Decide タイプのコンポーネント「個々のごみを清掃することができる」と Conditional Act タイプのコンポーネント「吸引によりごみを清掃することができる」および「持ち上げることでごみを清掃することができる」とが直接接続されるように変更されている．また，`Uses` ラベルに従って，コンポーネント「ごみの場所に到達することができる」と「バッテリーステーションに到達することができる」が，コンポーネント「目標物に到達することができる」のサービスを利用するための接続が追加されていることが分かる．これは，ごみ処理に関する Control loop とバッテリー管理に関する Control loop が，対象物への到達に関する Control

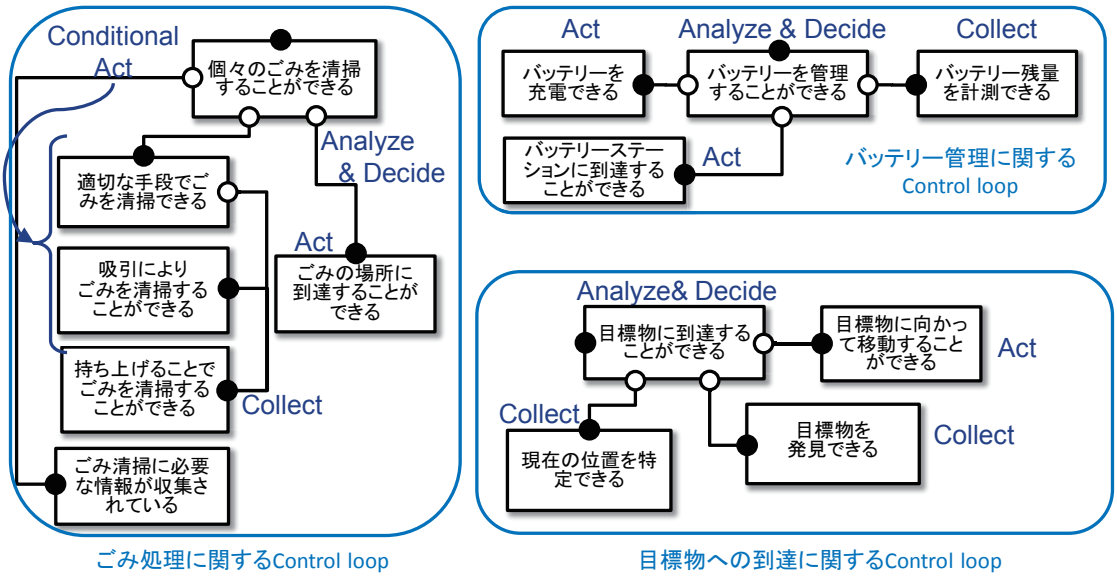

図 3.24. 生成されるコンフィギュレーション（第一段階）

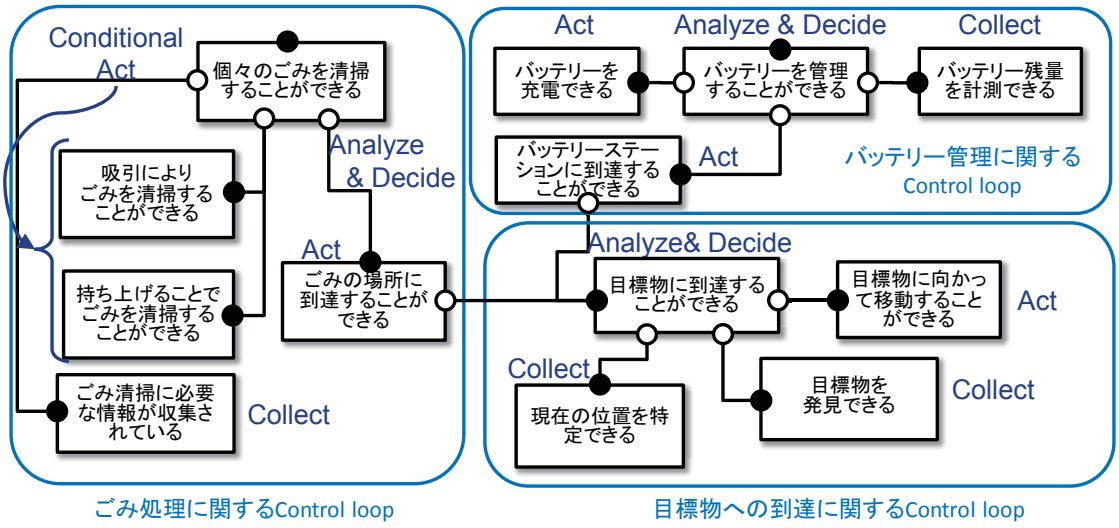

図 3.25. 生成されるコンフィギュレーション

loop の提供サービスを利用することを表現している．

このような変換により，本手法では，ゴールモデルの階層構造と Control loop パターンとを利用して，システムのコンフィギュレーションを決定する．

Algorithm 3 コンフィギュレーションの洗練化

[入力] *config*: Algorithm 2 により生成されたコンフィギュレーション

[出力] *config*: 洗練化後のコンフィギュレーション

```
 1: Components ← config.Components;
 2: Connections ← config.Connections;
 3: /* "Uses" ラベルで定義されたコンポーネント間を接続する */
 4: for all comp in Components do
 5:     if comp.associatedGoal.hasLabel("Uses <usedGoal>") then
 6:         usedComp ← Components.get(<usedGoal>);
 7:         rPort ← comp.addRequiredPort();
 8:         config.connectPorts(rPort, usedComp.providedPort);
 9:     end if
10: end for
11: /* Conditional Act を直接接続（仲介する Act 型コンポーネントを削除）する．
12:    Requesters(comp) : comp に対して reqPort ポートで接続するコンポーネントの集合 */
13: for all comp in Components do
14:     goal ← comp.associatedGoal;
15:     if goal.hasLabel("A") ∧ goal.refinement = OR-refinement then
16:         for all requester in Requesters(comp) do
17:             Connections ← Connections − {con | con ∈ Connections ∧ con.reqPort = requester.reqPort ∧
              con.provPort = comp.provPort};
18:             for all con in Connections do
19:                 if con.reqPort = comp.reqPort then
20:                     con.reqPort ← requester.reqPort;
21:                 end if
22:             end for
23:         end for
24:         Components ← Components − {comp};
25:     end if
26: end for
```

3.5　ソフトウェア進化を考慮した開発プロセス

前節までに，システム要求に対する要求記述としてのゴールモデルとその整形プロセス，整形後のゴールモデルに対するコンフィギュレーション決定法について述べた．本節では，整形プロセスやコンフィギュレーション決定法を用いた，ソフトウェア進化を考慮した開発プロセスについて述べる．本研究では，1.2 節で定義した開発プロセスに求められる要件に考慮し，

ゴールモデル整形プロセスを用いた以下のようなソフトウェア開発プロセスを定義する．

(1) ゴールモデルの構築，整形：システム開発者はまず，開発対象となるシステムに対する要求をゴールモデル上で分析する．このフェーズの目的は，システムに対する要求の分析・定義と，Control loop を抽出するための要求記述の構造化にある．要求分析法や要求，つまりゴールの記述は通常の KAOS などのゴール指向要求分析法に従うが，本開発プロセスでは，得られたゴールモデルに対して，本研究で導入する整形プロセスに従った整形，詳細化を施すことにより，システム構成を決定可能なモデルへと詳細化する．ソフトウェア進化時においても，機能追加などの進化に関する要求をゴールモデル上に記述し，追加した要求に対して，整形プロセスを再適用する．

(2) Control loop の設計：前フェーズで整形したゴールモデルから，システム構成を表現するコンフィギュレーションを決定する．3.4 節で述べたコンフィギュレーション決定法により，整形されたゴールモデルから，1 つ以上の Control loop を含んだコンフィギュレーションの決定が可能である．

コンフィギュレーションが決定すると，続いて各 Control loop の設計に移る．個々の Control loop の設計は，構造設計と振舞い設計に分類することができる．構造設計としては，Control loop 構成要素の決定，つまりコンフィギュレーション上のコンポーネントと実装クラスとの対応関係の決定が該当する．Control loop を構成する 3 つのコンポーネントタイプの責務の大きさなど対象システムの特徴を考慮し，コンフィギュレーション上の各コンポーネントタイプにそれぞれ個別のクラスを割り当てたり，複数のコンポーネントタイプを統合して 1 つのクラスとして実装するなどの対応付けを検討する．

振舞い設計については，入力変数の値に応じてアクションを決定・実行するという一連の処理の実現方法を検討する．ただし，本研究では変更影響の局所化を目的として Control loop の観点からシステムの構成要素を決定しているため，プロセスコントロールモデル [1] などの Control loop の典型的なモデルに束縛した設計・実装を義務付けるわけではない．各 Control loop の振る舞いは該当する主要ゴール以下のゴールを達成するものであればよく，ゴール群や関連エンティティとの関係情報をもとに設計する．ゴール記述から振る舞いモデルを構築する手段としては，例えば，文献 [43] などの手法が参考になる．この手法を用いる場合は，ゴールを時相論理記述に形式化し，振る舞いモデルとして LTS (Labeled Transition System) を構築することとなる．

併せて，Control loop 間のインタラクションについても設計する．このインタラクションは，コンフィギュレーション上における Control loop 間でのポート接続によるやり取りが該当し，データの送受信方法やサービスの提供手段を検討する．また，Uses ラベルにより被利用側となる Control loop については，優先度によるサービス提供や依頼順序に従ったサービス提供な

ど，利用側 Control loop を決定するための手段やインタフェースを決定させる．

進化時においては，進化前後のゴールモデルや生成されるコンフィギュレーションの差分をとることで，進化に対する要求の変化やコンフィギュレーションの変化の情報を明確化することができるため，これらを実現するための設計・実装モデル上の変更箇所を同定する．

(3) Control loop の実装：各 Control loop の動作と Control loop 間のインタラクションが決定すると，設計結果に基づいて Control loop を実装する．また，実装した Control loop を実行環境に配備（デプロイ）することでシステムを動作させる．本研究では，4 章で導入するプログラミングフレームワークにより，Control loop および Control loop 間インタラクションの実装とデプロイを支援する．

進化時には，同定された変更箇所を実装し，変更に関与する Control loop のみを変更する．つまり該当 Control loop の追加，削除，修正により，変更を実装コード上に反映させる．

3.6　進化時のゴールモデル整形

続いて，要求が変化した際の対処法としてのゴールモデル変更法について，特に主要ゴールが新たに追加される場合と，既存の主要ゴール内の一部が変更される場合の 2 通りの進化について説明する．以降本節では，清掃ロボットシミュレータの例を用いて進化への対応法を説明する．

3.6.1　進化 1：ごみの積載量管理機能の追加

まず，主要ゴールが新たに追加される場合の進化として，清掃ロボットに対するごみ積載量管理機能の追加を例に挙げ説明する．ごみ積載量管理機能の追加に関する新たな要求は，2.5 節で示した通りである．

［清掃ロボットに対する要求変化 1］フィールド上のごみの増加により，清掃ロボットのごみ積載量では一度に全てのごみが清掃できない場合が生じることとなった．そこで，清掃ロボットがごみの積載量を管理できる機能を新たに追加したい．清掃ロボットはごみの積載量を監視し，積載量が一定値以上になった場合，ごみ箱の場所を探し，ごみ箱の場所へ移動後，積載しているごみをごみ箱に捨てるという動作が求められる．

このような要求を記述すために，進化前の清掃ロボットに対応する図 3.18 のゴールモデルに対して，新たに追加するゴールツリーを図 3.26 に示す．このゴールツリーは，ごみの積載

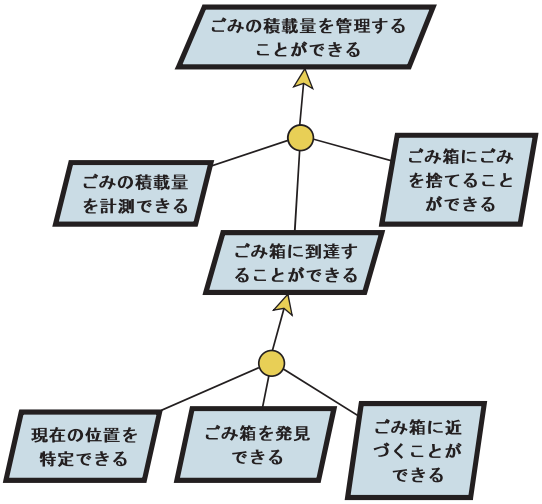

図 3.26. ごみ積載量管理機能の追加に伴い追加するゴールツリー

量を管理するためには，ごみ積載量の計測，ごみ箱への到達，ごみ捨ての 3 つの機能[*4]が必要であることを示し，ごみ箱への到達に関しては，さらにゴールが詳細化されている．

提案する開発プロセスにおいては，ソフトウェア進化時にも，まずゴールモデル上に新たな要求を記述し，その後，整形プロセスを再度適用することによりゴールモデルを洗練化させる．以下，本例における整形プロセスを概説する．

初期ゴールモデルの確認

まず，要求の変化に対して，定義 3.3.2 で示した要件「システムに対する機能要求が，すべてゴールモデル上にゴールとして記述されている」を満たすかどうかを判断する．本進化要求に対しては，ゴールモデルに図 3.26 のサブツリーが追加されることとなるが，ごみ積載量管理に必要と考えられる積載量計測，ごみ箱への移動，ごみ捨ての機能が機能要求として記述されているため，初期要求モデルが満たすべき要件は満足していると判断できる．もし，十分にゴールが分解されていない，あるいは，追加・変更すべき機能要求をすべてゴールモデル上に記述できていないと判断すれば，ゴールモデルを修正する．

エンティティの追加

次に，変化した要求に関連するエンティティをゴールモデルに追加する．図 3.26 で示したゴール群に対して，関連エンティティとその関係を追加した後のゴールモデルを図 3.27 に示

[*4] それぞれ，ゴール「ごみの積載量を計測できる」，「ごみ箱に到達することができる」，「ごみ箱にごみを捨てることができる」が対応する．

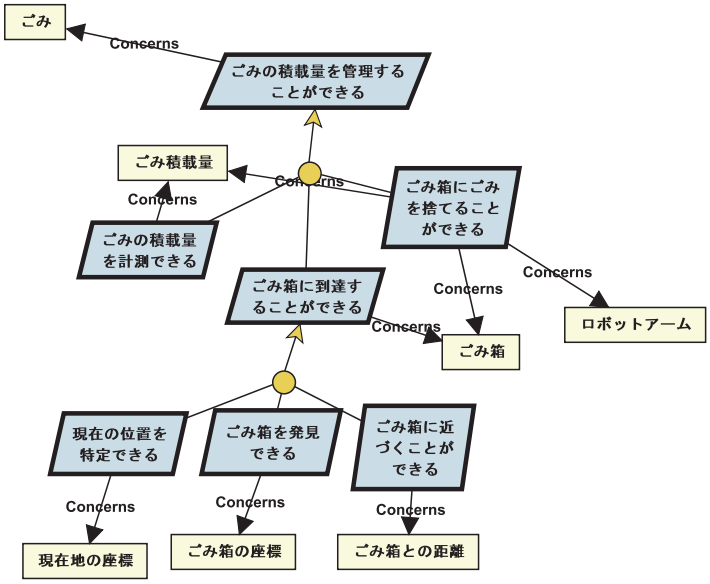

図 3.27. 追加部分に対するエンティティの追加

す．本例では，ごみ積載量管理機能の追加に対して，「ごみ」，「ごみ積載量」，「ごみ箱」，「ロボットアーム」，「現在地の座標」，「ごみ箱の座標」，「ごみ箱との距離」を関連エンティティとして同定し，追加記述する．

共通ゴールの集約

エンティティを追加すると，新たに追加したゴール群に関しても，共通ゴールとして集約できるゴールや，共通ゴールを利用することで達成できるゴールがあるかどうかを判断する．ごみ積載量管理機能の追加により追加されるゴールモデルにおいても，ごみ箱への到達については，既に共通ゴールとして集約されている対象物への移動に関するゴールを利用することで達成できると考えられるため，ごみ箱への到達に関する詳細記述をゴールツリーから削除し，ゴール「目標物に到達することができる」の利用を表現する Uses ラベルを付与する（図 3.28）．

主要ゴールの同定

共通ゴール集約後は，再度主要ゴールを同定する．本例における共通ゴール集約後のゴールモデルを図 3.29 に示す．このゴールモデルにおいては，進化前と同様に，ゴール「個々のごみを清掃することができる」，「バッテリーを管理することができる」，「目標物に到達することができる」を主要ゴール候補として同定することができる．加えて，今回の進化に伴うゴール

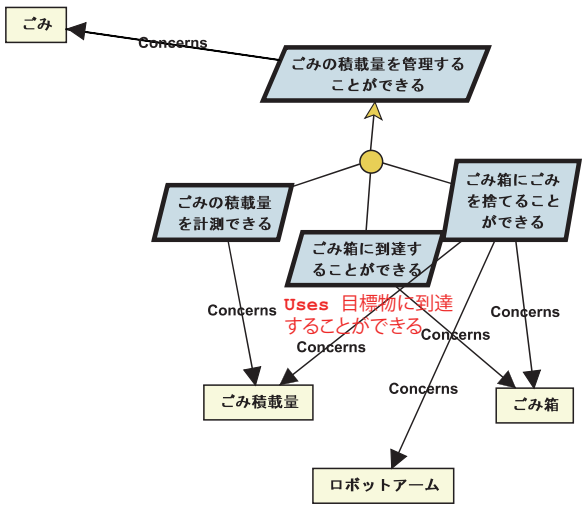

図 3.28. 共通ゴールの集約

モデルの変更により，指針 2 の「複数のサブゴールが同一エンティティと Concerns 関係にある」について，ゴール「ごみの積載量を管理することができる」がエンティティ「ごみ積載量」,「ごみ箱」,「ごみ」に対して合致し，ゴール「フィールド上のごみを清掃することができる」がエンティティ「ごみ」に対して合致することとなる．

以上から，要求変化後のゴールモデルでは主要ゴール候補として，図 3.29 中の（A)〜(E) に示される，「フィールド上のごみを清掃することができる」,「個々のごみを清掃することができる」,「バッテリーを管理することができる」,「ごみの積載量を管理することができる」および「目標物に到達することができる」が同定される．

続いて，これらの主要ゴール候補から主要ゴール集合を決定する．本進化後のゴールモデルにおいては，図 3.29 中の（A)〜(E) としてラベル付けされた主要ゴール候補の中から，以下の 2 通りの組み合わせにより，充足性判定条件 1 の「すべての機能要求がいずれかの主要ゴールを根とするサブツリーに包含されている」と，条件 2 の「いずれの主要ゴールも，他の主要ゴールを根とするサブツリーに包含されていない」が満たされる．

- **集合 1：** (A)
- **集合 2：** (B), (C), (D), (E)

本例においては，集合 1 を選択すると，システム中に唯一の Control loop しか抽出されなくなってしまい，進化に対しての変更影響が大きくなってしまうことと，進化前においては (B), (C), (E) の構成で主要ゴール集合を同定していたことから，進化前の主要ゴール集合に「ごみの積載量を管理することができる」を追加した集合に該当する集合 2 を，主要ゴール集合とし

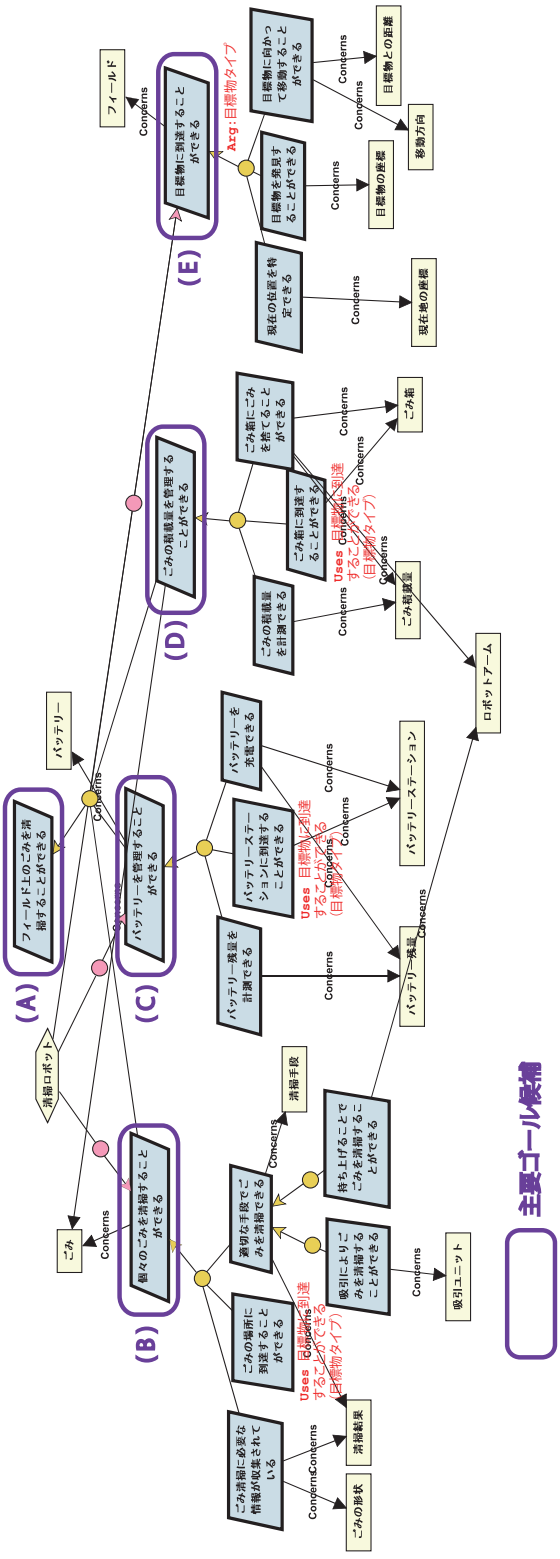

図 3.29. 共通ゴール集約後のゴールモデル

て採用する．

Control loop の構築

ゴールモデル上において主要ゴールが同定されると，各主要ゴールに対して Control loop を割り当てる．本例では，追加された主要ゴールに対して Control loop を新たに構築する．新たに追加された主要ゴール「ごみの積載量を管理することができる」に対しては，まず，入力変数としてエンティティ「ごみ積載量」を同定することができ，ごみ積載量を収集するゴールとして，既に記述されている「ごみの積載量を計測できる」を Collect タイプのゴールとして同定できる．同定された入力変数と Collect ゴール間の Concerns 関係に対しては，`：Monitor` ラベルを付与する．

続いて，同定した Collect タイプゴールを除くサブゴールであるゴール「ごみ箱に到達することができる」とゴール「ごみ箱にごみを捨てることができる」が Act タイプゴールであるかを判断する．これらのゴールはいずれも，アクションにより達成できる状態を表現しているため，Act タイプのゴールとして同定できる．ここで，後者のゴール「ごみ箱にごみを捨てることができる」については，エンティティ「ごみ積載量」を制御変数として同定することができるため，Concerns 関係に`：Control` ラベルを付与する．また，エンティティ「ロボットアーム」はごみを捨てるために操作可能なエンティティであることから，操作変数として同定することができ，該当する Concerns 関係に`：Manipulate` ラベルを付与する．

新たに追加された主要ゴール「ごみの積載量を管理することができる」に対する Control loop が構築されると，本主要ゴールを清掃ロボットの責務として新たに追加する．Control loop 構築後のゴールモデルを図 3.30 に示す．

競合の検出

最後に，Control loop 間の競合が存在するかどうかを，Entity-conflict パターンにより形式的に検査する．図 3.30 のゴールモデルでは，エンティティ「ロボットアーム」が，主要ゴール「個々のごみを清掃することができる」と「ごみの積載量を管理することができる」に対して，競合の可能性のあるエンティティとして抽出される．この競合に対しては，それぞれのゴールがごみの清掃と積載量管理を表現したものであり，今後の各機能個別の進化も考えると，2 つのゴールを集約するのではなく，独立した Control loop として構成した方が良いと判断できるため，解消法 1 の共通ゴールの導入により競合の解消を図る．本例では，新たな主要ゴール「アームを利用できる」を追加し，ロボットアームに関連するゴールには，新たな主要ゴールの利用を表現する `Uses` ラベルを付与する．解消法を適用し，新たな主要ゴールに対して Control loop を構築した後のゴールモデルを図 3.31 に示す．本ゴールモデルが，進化 1 に

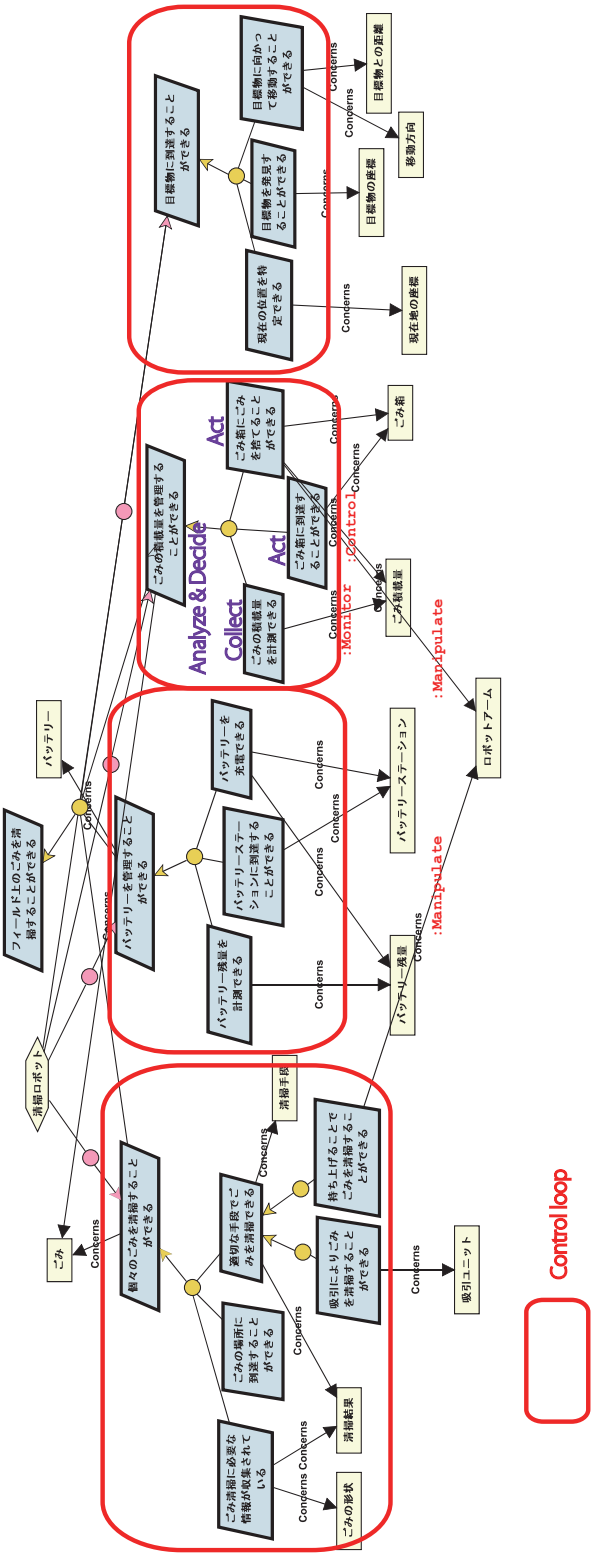

図 3.30. ごみ積載量管理機能に該当する Control loop 追加後のゴールモデル

対する整形プロセス適用後のゴールモデルとなる．

コンフィギュレーションへの変換

整形プロセス適用後のゴールモデル（図 3.31）から決定されるコンフィギュレーションを，図 3.32 に示す．図 3.25 のコンフィギュレーションと比較すると，ごみ積載量管理とロボットアーム管理に関する Control loop が新たに追加され，ごみ積載量管理のための Control loop からの目標物への到達に関する Control loop との接続と，ロボットアームに関する Control loop を利用するための接続が追加で必要となることが分かる（図中の赤線で示した箇所）．

3.6.2 進化 2：障害物回避機能の追加

続いて，既存主要ゴールの変更に該当する進化として，清掃ロボットに対する障害物回避機能の追加を例として説明する．障害物回避機能の追加に関する新たな要求は以下の通りである．

[清掃ロボットに対する要求変化 2] フィールド上にごみではない，通過不能であるオブジェクト（障害物）が設置される状況が生じることとなった．そこで，清掃ロボットには進行方向に障害物が存在する場合，迂回して目的の座標に到達する機能が求められることとなった．

進化要求に対してゴールモデルの構造に変更が少ない場合は，整形プロセスは簡略になることが多い．まず，この要求変化に対するゴールモデルの変更箇所を，図 3.33 に示す．本進化に対しては，目標物の到達に関するゴール群の変更により，進化要求に対する記述が可能である．図 3.33 の例では，進化前のゴールモデルに対して，ゴール「目標物を発見することができる」を「目標物・障害物を発見することができる」に，ゴール「目標物に向かって移動することができる」を「障害物を避けながら目標物に向かって移動することができる」に変更し，また，エンティティ「障害物の座標」を新たに追加している．

共通ゴールの集約に関しては，今回の記述変更は，既に共通ゴールとして集約されているゴール群内の変更に閉じており，新たに共通ゴールとして集約すべき箇所はないと判断する．また，主要ゴールの同定についても，前回開発時と同様のゴールを主要ゴールとして充足性が満足されるため，進化前の主要ゴール集合からの変更はない．

Control loop の構築については，変更したゴールは初期開発時同様，それぞれ Collect タイプ，Analyze & Decide タイプゴールとして同定する．また，今回追加したエンティティ「障害物の座標」は，入力変数として同定する．競合の検出についても，Entity-conflict パターンに

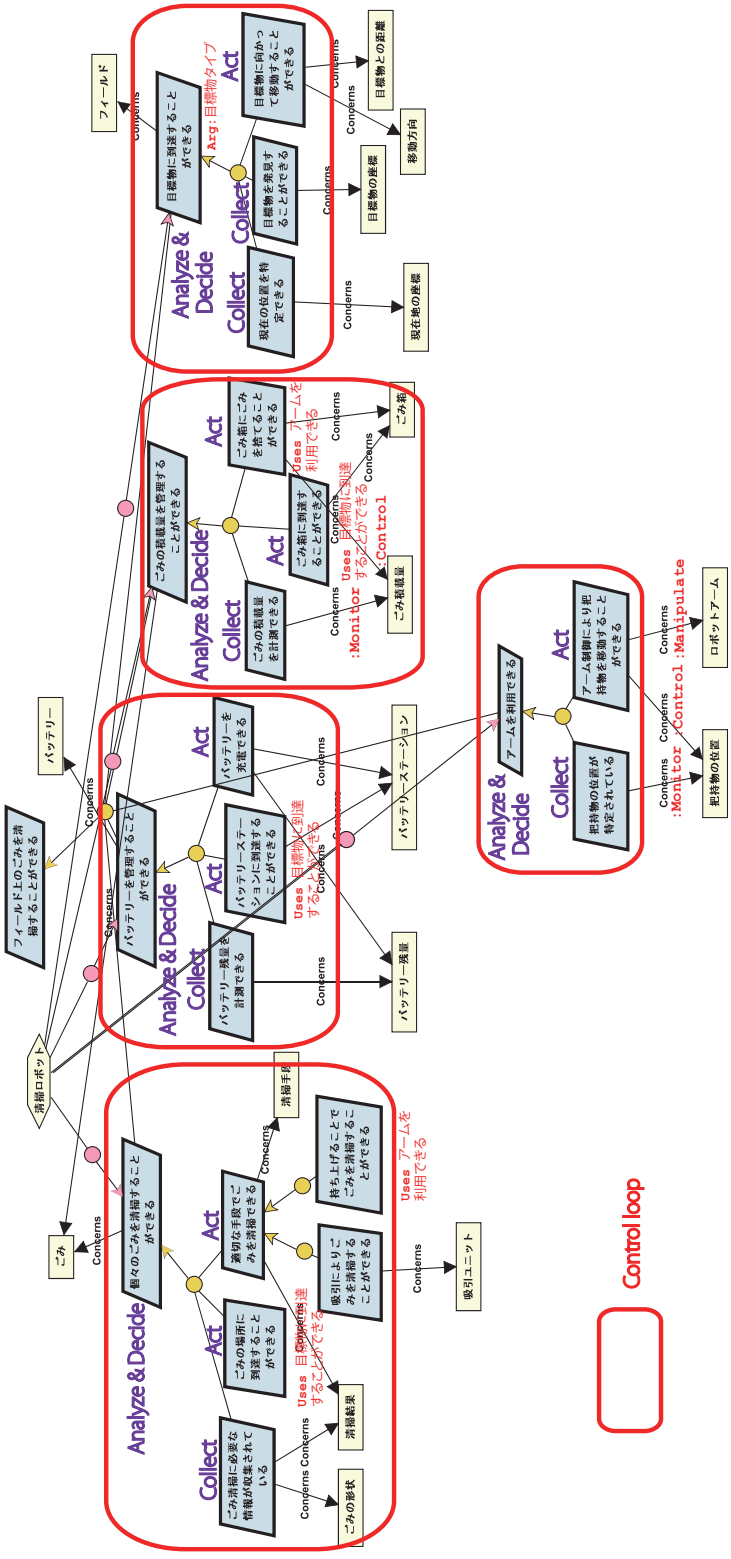

図 3.31. ごみ積載量管理機能の追加に対しての整形後ゴールモデル

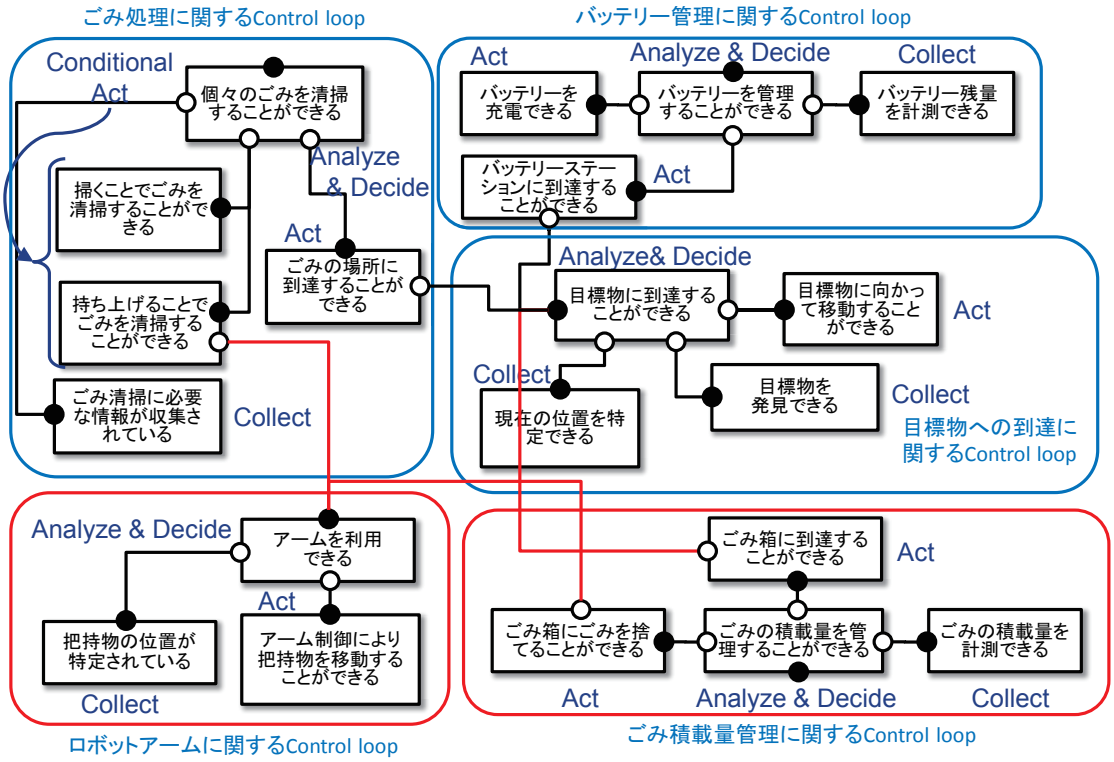

図 3.32. 積載量管理機能追加後のコンフィギュレーション

より検出される競合はない.

以上の変更に対して，整形後ゴールモデルから新たに決定されるコンフィギュレーションを図 3.34 に示す．図 3.32 の進化 1 後のコンフィギュレーションと比較すると，ゴールの変更に対応したコンポーネントが変更されていることが分かる.

3.7 まとめ

本章では，まず，本研究で要求記述として利用するゴールモデルから Control loop を同定するための整形プロセスについて述べた．整形プロセスの大きなアイデアは，ゴールモデル内において主要な役割を担う主要ゴールを同定する点と，同定した主要ゴールに対して，Control loop パターンを導入することによる Control loop のゴールモデル上での配備である．また，本章では，整形したゴールモデルの構造を利用した，コンフィギュレーションの決定法を示した．コンフィギュレーションとはシステムのアーキテクチャを示すものであり，本研究では，ゴールモデル上の一部のゴールとコンフィギュレーション上のコンポーネントとを対応付けたモデル変換により，コンフィギュレーションの決定を支援している.

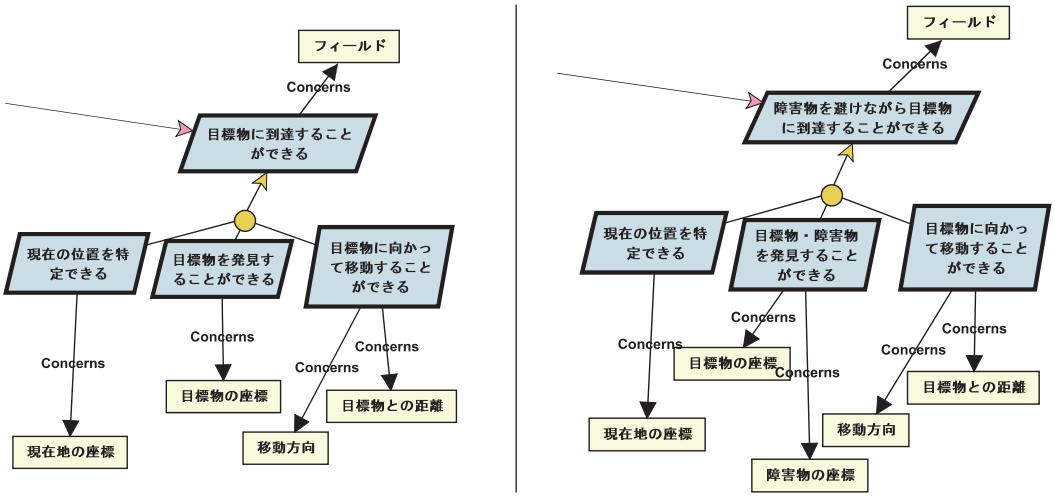

図 3.33. 障害物回避機能の追加に伴い変更するゴールツリー：(左図) 進化前，(右図) 進化後

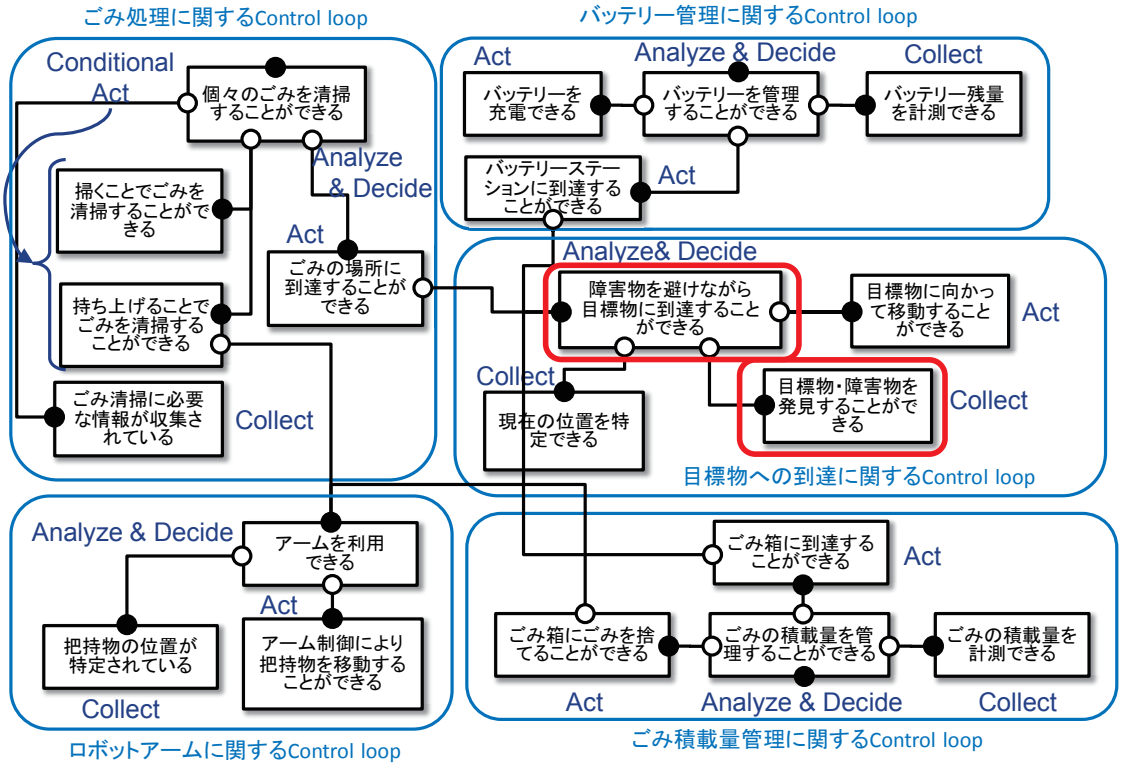

図 3.34. 障害物回避機能追加後のコンフィギュレーション

これらを利用することで，本研究では，進化の影響が分析可能であり，進化時の変更影響を限定化するソフトウェア開発プロセスを定義する．次章では，本研究で定義するソフトウェア開発プロセスを支援するツール群について論じる．

第 4 章

開発支援ツール

4.1 はじめに

前章では，Control loop を構成要素とするシステム構成，つまりコンフィギュレーションを決定するための開発プロセスについて定義し，主にゴールモデル整形プロセスを中心に説明した．Control loop を構成単位としてシステムをモデリングすることは，システムの各構成要素に独立した制御プロセスを持たせることを意味しており，この特性により，進化時に他の構成要素に与える影響を限定化させることが可能となる．

本章においては，進化を考慮したソフトウェアシステム開発の支援を目的とした開発支援ツール群について論じる．本研究で導入するツール，フレームワークと各モデルとの関係を図 4.1 に示す．本章では，まず，整形されたゴールモデルから，コンフィギュレーションやソフトウェア進化に必要な各種情報を生成するゴールモデルコンパイラ gocc の各機能について述べる．

続いて，Control loop を実装の単位とする，本研究で導入するプログラミングフレームワークについて述べる．本プログラミングフレームワークは，Control loop を 3 種類のコンポーネントにより構築し，複数の Control loop を並行動作させる点が特徴である．また，開発者が Control loop を制御，およびデプロイ可能なコマンドセットを提供する点も特徴として挙げられる．本章では，同プログラミングフレームワーク上で Control loop を実現するためのイディオムについても言及する．

4.2 gocc: ゴールモデルコンパイラ

3 章で導入したゴールモデルからのコンフィギュレーション生成法は，ゴールモデルの構造とゴール間の関係からコンフィギュレーションを決定するため，ゴールモデルに対してコン

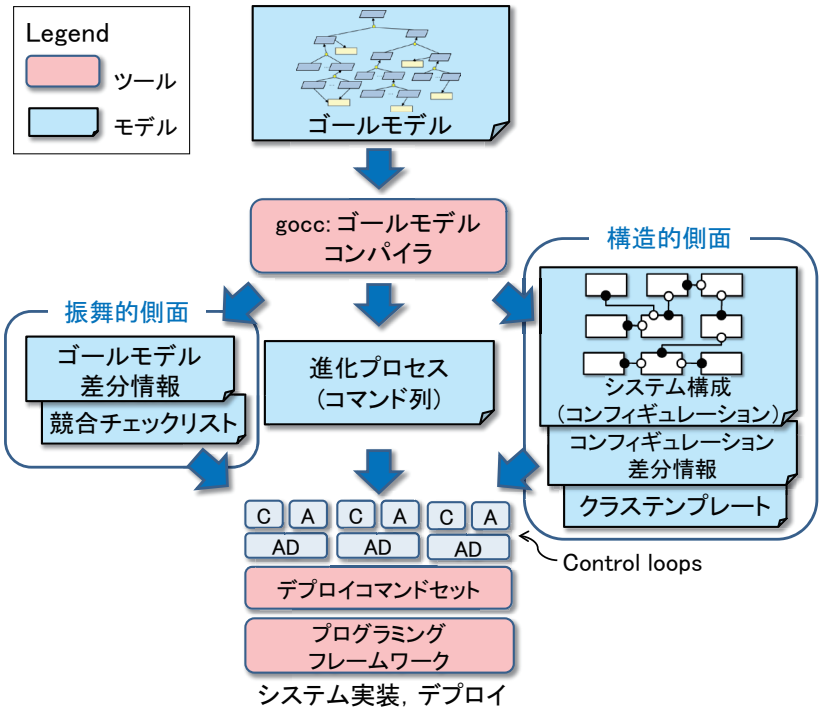

図 4.1. 進化を支援するコンパイラ，プログラミングフレームワーク

フィギュレーションを一意に決定できることができる．これは，変換の自動化が期待できることを意味する．また，ゴールモデルの解析過程やコンフィギュレーションの生成過程で得られる情報には，ソフトウェアの進化を実現する開発プロセスにあたり有益な情報も含まれていると考えられる．

そこで，本研究では，ゴールモデルを入力としたモデルコンパイラ gocc（Goal-oriented Configuration Compiler）を導入する．本研究では，gocc を用いて，コンフィギュレーション生成過程で得られた情報を各観点別に出力する．

gocc を利用した各モデル生成の流れを図 4.2 に示す．gocc は Eclipse Plugin [20] として実装されたゴールモデルコンパイラであり，まず，整形後のゴールモデルが記述された KAOS モデルを入力データとし，解析器において KAOS モデルの構造を解析する．gocc は，解析過程において，ゴールモデル内において競合が発生し得る箇所を，2 種類の Conflict パターンと照合することにより検出し，その結果を競合リストとして出力する．また，gocc は，進化前のバージョンの KAOS モデルも入力情報とすることで，進化前後のゴールモデルの差分を検出し，その結果を出力する．

ゴールモデルの解析が終わると，続いて変換器により，ゴールモデルをコンフィギュレー

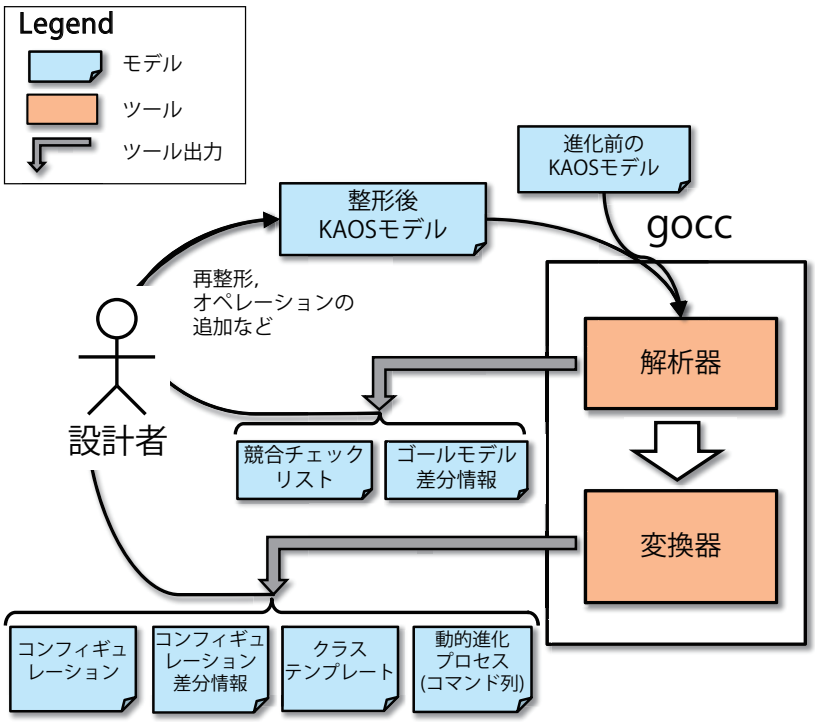

図 4.2. gocc を用いた進化型開発

ションへと変換する．gocc は，3 章で述べた変換法によりコンフィギュレーションを生成，出力するが，進化前後のコンフィギュレーションを比較することにより，進化時におけるシステム変更箇所を特定するための差分情報を出力する．また，コンフィギュレーション上の変更箇所から，Control loop の変更手順を表現したコマンド列を生成し，Control loop を実装するためのコードテンプレートも，コンフィギュレーションの情報をもとに生成する．

4.2.1　解析器

本研究では，ゴールモデルの記述に KAOS モデルの記述文法を利用する．KAOS のモデリングツールとしては，Objectiver [44] や国立情報学研究所開発の k-tool がある．オープンソースの汎用ドローツール Dia [45] も，KAOS による要求記述のためのプラグインを標準で装備している．本研究ではゴールモデルからの形式的な情報抽出を目的として，モデリング結果を XML ファイル形式で出力可能な Objectiver を KAOS モデリングツールとして利用する．gocc はまず，Objectivre が出力する XML 形式のファイルを入力データとし，与えられた KAOS モデルを解析する．

Objectiver が出力する XML ファイルの一部を図 4.3 に示す．KAOS モデルとして出力され

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <ERAModel>

  ...

  <C t="E" p="MM.xml#Method:NewKaos2003:MetaModel:Requirement"
     n="バッテリーステーションに到達することができる" id="E927">
    <AV n="Name" v="バッテリーステーションに到達することができる"/>
    <AV n="Pattern" v="Undefined"/>
    <AV n="Def" v="Uses 目標物に到達することができる"/>
    <AV n="Priority" v="Undefined"/>
    <AV n="Category" v="Undefined"/>
  </C>

  ...

  <C t="R" p="MM.xml#Method:NewKaos2003:MetaModel:Concerns"
     n="C7396" id="C7396">
    <R>
      <L r="Concerns" mx="1" mn="1" a="E927"/>
      <L r="ConcernedBy" mx="1" mn="1" a="E3374"/>
    </R>
  </C>

  ...
  <C t="E" p="MM.xml#Method:NewKaos2003:MetaModel:Entity"
     n="バッテリーステーション" id="E3374">
    <AV n="Name" v="バッテリーステーション"/>
  </C>

  ...
```

図 4.3. Objectiver の XML ファイル出力例

る XML ファイルでは，ゴールやエンティティなどのモデル要素だけでなく，Concerns 関係や，AND/OR-refinement リンク，責務割り当てなどの関係も出力情報に含まれる．さらに，あらかじめ定められた属性情報，例えば図 4.3 の `Requirement` 要素[*1]に含まれる “`Pattern`” や “`Def`” などの属性情報も出力情報に含まれる．

`gocc` は入力された KAOS ゴールモデルを解析し，もしゴールモデル上に Uses ラベルの定義ミスや，同一のゴールから AND-refinement と OR-refinement の両者が定義されているなどの構文解析エラーがあれば，コンパイルエラーを返す．構文解析エラーがなければ，Algorithm 6 に従って競合リストを返す．競合リストの生成については，4.2.4 節で述べる．

4.2.2 コンフィギュレーションの自動生成

解析器により KAOS モデルが解析されると，続いて `gocc` は解析結果をもとにコンフィギュレーションを生成する．変換の流れは，3.4 節で述べたとおりであり，`gocc` は，定義 3.3.5 に

[*1] 本研究では，既に述べたとおり，KAOS モデルにおけるゴールと要件（Requirement）とを統合してゴールとして扱っている．提案手法で追加する記述制約が Objectiver 上での要素間関係の制約を違反しないように，Objectiver 上では Requirement 要素を用いてゴールを記述している．

従ってシステムに割当てられたゴール集合を抽出し，それらのゴールに対してコンポーネントを割り当て，KAOS ゴールモデル上の AND/OR-refinement リンクに従ってコンポーネント間を接続する．このゴールモデルからシステムコンフィギュレーションへの変換は，3.4 節で定義した 2 つのアルゴリズムに従う．

Example 11　— 図 4.4 は，図 3.18 の清掃ロボットに対するゴールモデルを入力として gocc が出力するシステムコンフィギュレーションであり，図 3.25 で示したコンフィギュレーションと同一の情報が出力される．ここで，*addComponent(X, T)* はコンフィギュレーションへの *T* タイプのコンポーネント *X*（*X* はコンポーネント名）の定義（追加）を，*addPort(X, P)* はコンポーネント *X* 上へのポート *P* の追加を，*connect(P1, P2)* はポート *P1*，*P2* 間の接続の定義（追加）を示したものである．

4.2.3　階層構造情報の生成

提案手法においては，システム構成を表現したモデルとして，各 Control loop を構成するためのコンポーネント群とその接続関係が定義されたコンフィギュレーションを生成する．しかし，大規模システムを対象とする場合には，このようなコンフィギュレーションのような詳細な構造情報だけでなく，システム構成を俯瞰することのできる情報が求められる場合が多い．

そこで本研究では，コンフィギュレーションを補完する情報として，Control loop 間の階層構造情報も抽出する．階層構造情報を生成するためのアルゴリズムは Algorithm 4 に示すとおりであり，同アルゴリズムで利用するメソッド *checkHierarchy()* のアルゴリズムを Algorithm 5 に示す．この階層構造に関する情報は，ゴールモデル上に定義された Uses 関係に基づいて生成する．ただし，Uses 関係は通常，Act タイプのゴール上に定義されるが，階層構造情報は Control loop 間の関係を定義するため，Control loop を割り当てる主要ゴール，つまり Analyze & Decide タイプのゴール名を用いて表現する．図 4.5 は，図 3.31 の清掃ロボットに対するゴールモデルを入力として gocc が出力する階層構造情報と，得られた階層構造情報を可視化したものである．

4.2.4　競合チェックリストの生成

複数の Control loop が独立して動作する場合，Control loop 間で発生し得る**競合**（Conflict）を考慮する必要がある．本研究では，変数へのアクセスにより生じる競合については，唯一の Control loop に変数へのアクセスの責務を割当てることにより競合を回避する．この場合，Control loop 間での同 Control loop 利用に対する競合，つまり，複数の Control loop が共通の

```
%% #Components: 13 #Connections: 12 #Control loops: 3

%%% Components %%%
addComponent(バッテリーを管理することができる, ANALYZE_and_DECIDE)
addPort(バッテリーを管理することができる, p0)
addPort(バッテリーを管理することができる, r0)
addPort(バッテリーを管理することができる, r1)
addPort(バッテリーを管理することができる, r2)
addComponent(個々のごみを清掃することができる, ANALYZE_and_DECIDE)
addPort(個々のごみを清掃することができる, p0)
addPort(個々のごみを清掃することができる, r0)
addPort(個々のごみを清掃することができる, r1)
addPort(個々のごみを清掃することができる, r2)
addComponent(目標物に到達することができる, ANALYZE_and_DECIDE)
addPort(目標物に到達することができる, p0)
addPort(目標物に到達することができる, r0)
addPort(目標物に到達することができる, r1)
addPort(目標物に到達することができる, r2)
addComponent(バッテリーを充電できる, ACT)
addPort(バッテリーを充電できる, p0)
addComponent(バッテリーステーションに到達することができる, ACT)
addPort(バッテリーステーションに到達することができる, p0)
addPort(バッテリーステーションに到達することができる, r0)
addComponent(バッテリー残量を計測できる, COLLECT)
addPort(バッテリー残量を計測できる, p0)
addComponent(ごみの場所に到達することができる, ACT)
addPort(ごみの場所に到達することができる, p0)
addPort(ごみの場所に到達することができる, r0)
addComponent(ごみ清掃に必要な情報が収集されている, COLLECT)
addPort(ごみ清掃に必要な情報が収集されている, p0)
addComponent(現在の位置を特定できる, COLLECT)
addPort(現在の位置を特定できる, p0)
addComponent(目標物に向かって移動することができる, ACT)
addPort(目標物に向かって移動することができる, p0)
addComponent(目標物を発見することができる, COLLECT)
addPort(目標物を発見することができる, p0)
addComponent(持ち上げることでごみを清掃することができる, ACT)
addPort(持ち上げることでごみを清掃することができる, p0)
addComponent(吸引によりごみを清掃することができる, ACT)
addPort(吸引によりごみを清掃することができる, p0)

%%% Connections %%%
connect(バッテリーを管理することができる.r0, バッテリーを充電できる.p0)
connect(バッテリーを管理することができる.r1, バッテリーステーションに到達することができる.p0)
connect(バッテリーを管理することができる.r2, バッテリー残量を計測できる.p0)
connect(個々のごみを清掃することができる.r0, ごみの場所に到達することができる.p0)
connect(個々のごみを清掃することができる.r1, ごみ清掃に必要な情報が収集されている.p0)
connect(目標物に到達することができる.r0, 現在の位置を特定できる.p0)
connect(目標物に到達することができる.r1, 目標物に向かって移動することができる.p0)
connect(目標物に到達することができる.r2, 目標物を発見することができる.p0)
connect(個々のごみを清掃することができる.r2, 持ち上げることでごみを清掃することができる.p0)
connect(個々のごみを清掃することができる.r2, 吸引によりごみを清掃することができる.p0)
connect(バッテリーステーションに到達することができる.r0, 目標物に到達することができる.p0)
connect(ごみの場所に到達することができる.r0, 目標物に到達することができる.p0)
```

図 4.4. gocc により生成されるコンフィギュレーション

Algorithm 4 階層構造情報の生成

[入力] *gModel*: KAOS ゴールモデル

[出力] *hList*：階層構造が記述されたリスト

```
1: assignedGoals ← システムに責務割当てされているゴール群;
2: goalQueue.enqueue(agent); //ゴールを格納するキュー
3: while goalQueue.size() ≠ 0 do
4:     goal ← goalQueue.dequeue();
5:     hList ← checkHierarchy(goal, hList);
6: end while
7: return hList;
```

Algorithm 5 *checkHierarchy(goal, hList)*: Uses 関係からの階層構造情報抽出メソッド

[入力] *goal*: 検査対象のゴール，*hList*：階層構造が記述されたリスト

[出力] *hList*：階層構造が記述されたリスト

```
1: if goal.hasLabel("Uses ⟨usedGoal⟩") then
2:     adTypeGoal ← goal.getADTypeGoal();
3:     entry ← adTypeGoal.getName()+ " Uses " + ⟨usedGoal⟩";
4:     if ¬ hList.contains(entry) then
5:         hList.add(entry);
6:     end if
7: end if
8: for all child in goal.childrenGoals do
9:     checkHierarchy(child, hList);
10: end for
11: return hList;
```

Control loop を利用する可能性のある状況に着目する必要がある．本研究では，開発の初期段階で競合が発生し得る箇所を特定することを目的とし，ゴールモデル上で競合が発生し得る箇所が存在するかどうかを検査する．

gocc は，入力されたゴールモデルから，本研究で導入する 2 種類の Conflict パターンに合致する箇所を競合発生の可能性がある箇所として検出する．Conflict パターンの 1 つは，3.3.6 節で定義したエンティティに対する競合に着目した Entity-conflict パターンであり，整形プロセス時同様に，変数に対する競合の検出に利用する．この検査は，整形プロセスに対するチェック機能としての意味を持つ．もう一方はゴールに対する競合に着目した Goal-conflict パターンであり，Control loop の利用に関する競合の検出に利用する．Goal-conflict パターンの定義を以下に示す．

定義 4.2.1　**Goal-conflict パターン**

```
個々のごみを清掃することができる Uses 目標物に到達することができる
個々のごみを清掃することができる Uses アームを利用できる
バッテリーを管理することができる Uses 目標物に到達することができる
ごみの積載量を管理することができる Uses 目標物に到達することができる
ごみの積載量を管理することができる Uses アームを利用できる
```

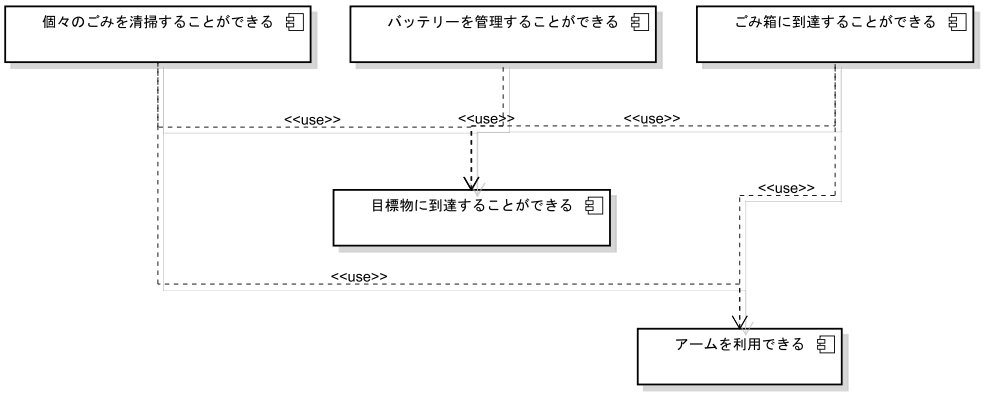

図 4.5. 階層構造情報の出力例（上段： gocc により生成される階層構造情報，下段：コンポーネント図による可視化表記）

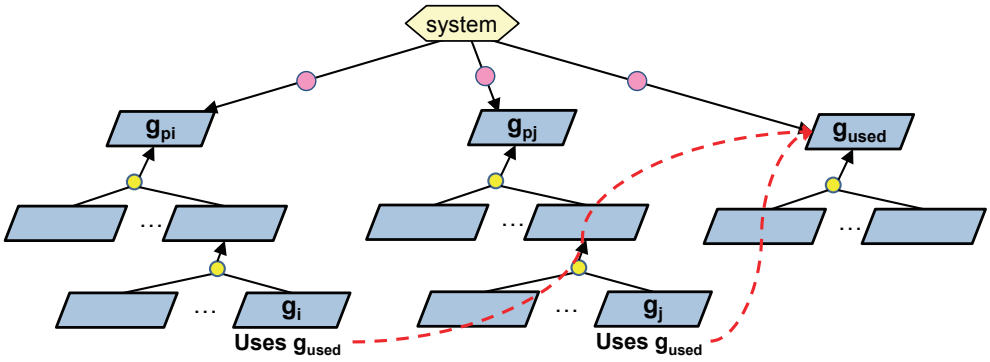

図 4.6. Goal-conflict パターン

以下の条件を満たす g_{used} を，競合の可能性のあるゴールとする．ここで，*PGoal* は主要ゴールを要素とする集合を，述語 *isAncestor*(x, y) はゴール x がゴール y の祖先である場合に真となる述語を示す．

$$\exists g_{used}(g_{used} \in PGoal \ \wedge \exists g_i \exists g_j(g_i.hasLabel(\text{“Uses”}, g_{used}) \wedge g_j.hasLabel(\text{“Uses”}, g_{used}) \wedge \exists g_{p_i} \exists g_{p_j}(g_{p_i} \in PGoal \wedge g_{p_j} \in PGoal \wedge isAncestor(g_{p_i}, g_i) \wedge isAncestor(g_{p_j}, g_j) \wedge g_{p_i} \neq g_{p_j}))) \quad \square$$

Goal-conflict パターンのイメージを図 4.6 に示す．Goal-conflict パターンは，複数の主要ゴールを根とするツリー，すなわち複数の Control loop から，`Uses` ラベルにより依存関係を定義されている主要ゴール，つまり Control loop を競合対象として同定するためのものである．

2 種類の Conflict パターンは，ゴールモデル上のゴールに対して，エンティティとの `Concerns` 関係や `Uses` ラベルを利用することで Control loop 間で発生する可能性のある競合を検出するものであり，いずれもゴールモデルの構造に基づいた検出が可能である．従って，ツール，つまり `gocc` による形式的な検出が可能である．`gocc` は，入力として与えられたゴールモデルに対して，Entity-conflict パターンと Goal-conflict パターンの 2 種類の Conflict パターンに照合する箇所を，Control loop に対する競合発生箇所として検出する．`gocc` に実装する競合検出アルゴリズムを Algorithm 6 に示す．Algorithm 6 では最初に，システムに割り当てられた主要ゴール単位で各ツリーに属するゴール群を集約する．続いて，集約された各ゴールが，`Concerns` 関係あるいは `Uses` ラベルを持つかどうかをすべてチェックし，関連を持つエンティティと利用するゴールとを主要ゴール単位で集約する．その後，複数の主要ゴール間で `Concerns` 関係を定義されているエンティティや `Uses` ラベルで利用関係を定義されているゴールが存在すれば，該当エンティティあるいは該当ゴールを競合対象とし，関与している Control loop（主要ゴール）の情報とともに競合リストのエントリとして追加する．最後に，すべての Control loop 間での検査が終わったら，競合リストを出力する．

競合リストを利用した競合への対応法は，以下のとおりである．

Procedure 4.2.1　競合の検出と対応法

1. **競合の検出:** `gocc` を実行して，競合リストを生成する．
2. **競合の判定:** `gocc` が出力する競合リストは，ゴールモデルの構造から 2 種類の Conflict パターンに合致する部分を抽出しているに過ぎないため，実際には競合が発生しない状況も抽出している可能性がある．従って，得られた競合リストをもとに，これらの競合が実際に発生し得るかを判断する．

   競合対象としてエンティティが検出された場合は，3.3 節で示した整形プロセスに戻り，ゴールモデルを修正する．もし，競合対象として共通ゴールが検出された場合は，次に述べる競合回避策について検討する．
3. **競合回避策の設計:** Goal-conflict パターンで検出された競合については，本研究では被利用側 Control loop に競合回避の責務を持たせる．具体的には，被利用側 Control loop における Analyze & Decide タイプのコンポーネントが競合回避の責務を持つ．Control loop の利用に関する競合を避けるために，優先度によるサービス提供や依頼順序に従っ

Algorithm 6 競合検出アルゴリズム

[入力] *AssignedGoals:* 責務割り当てされたゴール（*AD* タイプゴール）の集合

[出力] *ConfList:* 競合リスト

```
1: // サブツリー単位で関連するエンティティ，利用するゴールを集計
2: for all root in AssignedGoals do
3:     Resources(root) ← { };
4:     GoalSet(root) ← {root} ∪ {goal | isAncestor(root, goal)};
5:     for all g in GoalSet(root) do
6:         Resources(root) ← Resources(root) ∪ {ent | ent ∈ Entities ∧ concerns(g, ent) ∈ Concerns ∧
           ¬concerns(g, ent).hasLabel("Monitor")};
7:         // "Uses" ラベルで参照しているゴールも含める
8:         if g.hasLabel("Uses <usedGoal>") then
9:             Resources(root) ← Resources(root) ∪ {<usedGoal>};
10:        end if
11:    end for
12: end for
13: // 複数の Control loop と関連するエンティティ，利用されているゴールを検出
14: for all root in AssignedGoals do
15:    for all other in AssignedGoals - {root} do
16:        if (∃ent (ent ∈ Resources(root) ∧ ent ∈ Resources(other))) then
17:            ConfList.append(<ent, root, other>);
18:        end if
19:    end for
20: end for
```

たサービス提供など，利用側 Control loop を決定するための手段やインタフェースを決定させる． □

Example 12 — 例えば，ごみの積載量管理機能を追加後の整形後ゴールモデル（図 3.31）に対して，gocc は表 4.1 の競合リストを出力する[*2]．表 4.1 の競合リストの最初の 3 つのエントリは，ごみ清掃，バッテリー管理，ごみ積載量管理の 3 つの Control loop が目標物への移動を実現する Control loop を共通利用することにより発生する競合を示している．実際に，ごみ清掃，バッテリー管理，ごみ積載量管理に際しては，それぞれの目標物が，ごみ，バッテリーステーション，ごみ箱と異なるため，3 つの Control loop が同時に目標物への移動を実現する Control loop を利用しようとする競合が発生すると考えられる．この競合に対しては，被利用側，つまり目標物への移動を実現する Control loop が競合回避の責務を持つ．競合回避法は，

[*2] 実際には，gocc は CSV 形式で競合リストを出力する．

表 4.1. 図 3.31 のゴールモデルから生成される競合リスト

<Conflict object>	**<Control loop 1>**	**<Control loop 2>**
目標物に到達することができる	個々のごみを清掃することができる	バッテリーを管理することができる
目標物に到達することができる	個々のごみを清掃することができる	ごみの積載量を管理することができる
目標物に到達することができる	バッテリーを管理することができる	ごみの積載量を管理することができる
アームを利用できる	個々のごみを清掃することができる	ごみの積載量を管理することができる

競合が発生し得る状況を検討し，その状況を回避する制御を同 Control loop に設計・実装することであるが，本例では，この 3 つの機能を実現する Control loop は，バッテリが切れると清掃ロボットが動作しなくなること，積載量を超えるとごみを積載することができなくなることから，バッテリー管理機能，積載量管理機能，ごみ清掃機能の順で優先順位を決定し，優先順位に従って移動のサービスを提供するという手段が妥当であると考えられる．

一方，表 4.1 の競合リストの最後のエントリは，ロボットアームの利用に関する競合を表している．この検出結果も，共通する Control loop の利用に関するものであり，積載量を超えるとごみを積載することができなくなることから，積載量管理に関する Control loop の利用優先度を高く設定することで，競合を防ぐこととする．

このように，競合に関する分析結果は，Control loop の設計，実装時に，競合が発生し得る状況においてどの Control loop の動作を優先させ，また，どの Control loop においてその制御を実現するかを決定するために用いる．

4.2.5　差分情報の抽出

通常のソフトウェア開発においては，Charrada ら [46] が指摘するように，ソフトウェア進化時には多くの場合，直接ソースコードに変更が加えられ，要求記述が変更されることは少ない．一方，本研究においては，進化時にはゴールモデルを変更し，その後コンフィギュレーションを再生成するため，進化時にもソフトウェア開発プロセス上の各モデルが変更されるという特徴を持つ．そこで本研究では，これらのモデルに対して，進化前後の差分を取ることで，進化に対する要求の変化や，進化に伴うシステム構成の変更箇所を明確化させる．

本研究では，要求進化後のゴールモデルから進化に必要な設計情報を抽出する際に，進化前

のゴールモデルも参照することにより，ゴールモデル上での差分と，生成されるコンフィギュレーション上での差分を検出し，変更情報として生成させる．これらの検出結果は，進化に対して生じた要求・設計モデル上の変更箇所を示したものとなる．

具体的には，進化前後のゴールモデル上の各要素・関係（ゴール，エンティティ，Concerns 関係，責務割り当て）に対して差分が生じているかどうかを検査を検査する．ゴールモデル上の差分を把握することにより，進化に対する設計モデル上での変更内容が要求の変化に合致しているかどうかを確認することができるようになる．ゴールの追加・変更は達成すべき状態の追加・変更に該当し，エンティティや Concerns 関係の追加・変更は，考慮すべきオブジェクトの追加・変更を意味する．また，責務割り当ての変更は，システムに構築すべき Control loop の変更を意味する．

一方，コンフィギュレーションについては図 4.4 のようなテキスト形式で生成されるため，`diff` コマンドによる比較により，変更箇所を検出することが可能である．コンフィギュレーション上の差分情報からは主に，新たに追加，あるいは変更すべきコンポーネントと Control loop 間での新たな接続関係を把握することができる．

Example 13 — ごみ積載量管理機能追加後のゴールモデル（図 3.31）と，進化前のバージョンにあたる初期開発時の整形後ゴールモデル（図 3.18）を入力として与えた場合の差分抽出結果を，図 4.7，図 4.8 に示す．まず，図 4.7 は，ごみ積載量管理機能の追加に対して生じたゴールモデル上での差分である．図 4.7 では，ゴール（3〜9 行目）やエンティティ（10〜12 行目）の追加，Concerns 関係の追加・変更（13〜21 行目）や責務割り当ての追加（1,2 行目）が差分として検出されている．この差分情報からは特に，2 つの Control loop が追加されたことが，1,2 行目の責務割り当ての追加から確認でき，また，ロボットアームを制御する Control loop を新たに切り出したことにより，ロボットアームに関与する Control loop が進化前後で変更されていることが，Concerns 関係の変化（13 行目と 20 行目）と進化後のゴールモデルから確認できる．

一方で図 4.8 は，ごみ積載量管理機能の追加に対して生じた，コンフィギュレーション上での差分を示したものである．まず，最初の差分（ラベル “1c1”）からは，コンフィギュレーションの構成要素数，つまりコンポーネント数とコンポーネント間接続数の変化が把握できる．ラベル “3a4,12”，“18a28,39”，“40a62” はコンポーネントとポートの追加を示した箇所になる．ここでは，ごみ積載量管理機能の追加により，本機能を実現する Control loop とロボットアーム制御を実現する Control loop を構成する合計 7 つのコンポーネントを追加する必要があることが分かる．一方，後半の記述（ラベル “42a65,69”，“52a80,81”，“54a84”）は，進化により変更するコンポーネント間接続を示したものである．接続に関しては，追加する 2 つ

```
 1: > Assignment(清掃ロボット, ごみの積載量を管理することができる)
 2: > Assignment(清掃ロボット, アームを利用できる)
 3: > Goal(ごみ箱にごみを捨てることができる)
 4: > Goal(アームを利用できる)
 5: > Goal(ごみ箱に到達することができる)
 6: > Goal(ごみの積載量を計測できる)
 7: > Goal(アーム制御により把持物を移動することができる)
 8: > Goal(ごみの積載量を管理することができる)
 9: > Goal(把持物の位置が特定されている)
10: > Entity(ごみ箱)
11: > Entity(把持物の位置)
12: > Entity(ごみ積載量)
13: < Concerns(持ち上げることでごみを清掃することができる, ロボットアーム)
14: > Concerns(ごみ箱に到達することができる, ごみ箱)
15: > Concerns(ごみ箱にごみを捨てることができる, ごみ積載量)
16: > Concerns(ごみ箱にごみを捨てることができる, ごみ箱)
17: > Concerns(ごみの積載量を管理することができる, ごみ)
18: > Concerns(ごみの積載量を計測できる, ごみ積載量)
19: > Concerns(把持物の位置が特定されている, 把持物の位置)
20: > Concerns(アーム制御により把持物を移動することができる, ロボットアーム)
21: > Concerns(アーム制御により把持物を移動することができる, 把持物の位置)
```

図 4.7. ごみ積載量管理機能追加に対するゴールモデル上での差分検出

の Control loop 内のコンポーネント間接続の他に，ごみ積載量管理に関する Control loop が目標物への到達を実現する Control loop を利用するための接続と，ロボットアームを制御する Control loop に対してサービスを利用するための 2 つの他 Control loop からの接続が新たに必要となることが確認できる．

4.2.6　その他の生成モデル

これまでに述べた各種モデルの他に，gocc は与えられたゴールモデルから，次節で述べるプログラミングフレームワークの利用を前提とした 2 種類の情報を生成する．一つは，システム実装に必要なクラスファイルのテンプレートであり，もう一方は，プログラミングフレームワークが提供する動的進化を実現するための Control loop デプロイコマンドにより構成される動的進化プロセス系列である．後者は，gocc に実装した差分情報生成機能に基づいた決定法を用いるが，詳細は次節のプログラミングフレームワークの議論で述べる．

4.3　Control loop 実装・デプロイのためのプログラミングフレームワーク

提案手法のソフトウェア進化を支援する一つのフレームワークとして，本研究では，同定した Control loop の実装と実行をサポートするプログラミングフレームワークを実装した．本節

```
1c1
< %% #Components: 13 #Connections: 12 #Control loops: 3
---
> %% #Components: 20 #Connections: 20 #Control loops: 5
3a4,12
> addComponent(ごみの積載量を管理することができる, ANALYZE_and_DECIDE)
> addPort(ごみの積載量を管理することができる, p0)
> addPort(ごみの積載量を管理することができる, r0)
> addPort(ごみの積載量を管理することができる, r1)
> addPort(ごみの積載量を管理することができる, r2)
> addComponent(アームを利用できる, ANALYZE_and_DECIDE)
> addPort(アームを利用できる, p0)
> addPort(アームを利用できる, r0)
> addPort(アームを利用できる, r1)
18a28,39
> addComponent(ごみの積載量を計測できる, COLLECT)
> addPort(ごみの積載量を計測できる, p0)
> addComponent(ごみ箱にごみを捨てることができる, ACT)
> addPort(ごみ箱にごみを捨てることができる, p0)
> addPort(ごみ箱にごみを捨てることができる, r0)
> addComponent(ごみ箱に到達することができる, ACT)
> addPort(ごみ箱に到達することができる, p0)
> addPort(ごみ箱に到達することができる, r0)
> addComponent(アーム制御により把持物を移動することができる, ACT)
> addPort(アーム制御により把持物を移動することができる, p0)
> addComponent(把持物の位置が特定されている, COLLECT)
> addPort(把持物の位置が特定されている, p0)
40a62
> addPort(持ち上げることでごみを清掃することができる, r0)
42a65,69
> connect(ごみの積載量を管理することができる.r0, ごみの積載量を計測できる.p0)
> connect(ごみの積載量を管理することができる.r1, ごみ箱にごみを捨てることができる.p0)
> connect(ごみの積載量を管理することができる.r2, ごみ箱に到達することができる.p0)
> connect(アームを利用できる.r0, アーム制御により把持物を移動することができる.p0)
> connect(アームを利用できる.r1, 把持物の位置が特定されている.p0)
52a80,81
> connect(ごみ箱にごみを捨てることができる.r0, アームを利用できる.p0)
> connect(ごみ箱に到達することができる.r0, 目標物に到達することができる.p0)
54a84
> connect(持ち上げることでごみを清掃することができる.r0, アームを利用できる.p0)
```

図 4.8. ごみ積載量管理機能追加に対するコンフィギュレーション上での差分検出

では，このプログラミングフレームワークについて説明する．

4.3.1 フレームワークに求められる要件

提案手法によるシステムモデリングでは，システムの構成要素として Control loop が複数抽出されることとなる．Control loop は，環境やシステム状態などの情報の収集（Collect），得られた情報を基にした分析（Analyze），分析結果による振舞いや構成変更のための意思決定（Decide），変更後の実行（Act）の 4 つのプロセスが繰り返される制御構造を指す．本研究で提案する開発プロセスにおいては，このような Control loop をコンポーネントの結合により実現する必要がある．また，Control loop を構成要素とするシステムを実現するためには，複数

の Control loop を並行に動作させる必要がある．

これらの背景から，本研究では Control loop を構成要素とするシステムを実現するためのプログラミングフレームワークに求められる要件として以下を定義する．

定義 4.3.1　Control loop を構成要素とするシステムを実現するためにプログラミングフレームワークに求められる要件

- **要件 1（Control loop の実現容易性）**：Control loop の各アクティビティ（Collect, Analyze, Decide, Act）の実装を支援する環境が提供されている．
- **要件 2（Control loop 群の制御）**：Control loop 群の制御が可能であり，複数の Control loop が並行に動作可能な実行環境が提供されている．

4.3.2　Control loop 実装のためのプログラミングフレームワーク

本研究では，定義 4.3.1 の要件を満たすフレームワークとして，複数の並行タスクを実装可能であるエージェントプラットフォーム JADE [47] を拡張したプログラミングフレームワークを実装した．本節ではまず，JADE の概要について説明し，その後，Control loop 実装を考えた際に拡張が必要な部分について論じる．

JADE

JADE は Java 言語によるマルチエージェントシステムの構築が可能なプログラミングフレームワークであり，JADE の大きな特徴として，複数のビヘイビアクラスを用意することで，エージェント，つまり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の動作が定義可能である点が挙げられる．これは，システムの動作に必要となる複数の並行タスクが記述可能であるということを意味する．

JADE では振舞いを定義，つまり実装するために，Behaviour クラスとその派生クラスが提供されている．例えば，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．また，JADE はマルチエージェントシステムの実装を目的として設計されていることから，エージェント間で送受信されるメッセージの処理に関する実装を支援するビヘイビアも数多く提供されている．

開発者はビヘイビアの利用形態に応じて適切な派生クラスを選択し，その派生クラスを継承するビヘイビアを実装することで，エージェントの多様な振舞いを実装することができる．4.9

```
 1: public class SampleAgent extends Agent{
 2:   protected void setup(){
 3:     addBehaviour(new Printer(this,"p1"));
 4:     addBehaviour(new Printer(this,"p2"));
 5:   }
 6: }
 7:
 8: class Printer extends SimpleBehaviour{
 9:   int n=0;
10:   String name;
11:
12:   public Printer(Agent a, String st){
13:      super(a);
14:      name = st;
15:   }
16:   public void action(){
17:     System.out.println("I am "+name+".");
18:     n++;
19:   }
20:   public boolean done(){return n>=5;}
21: }
```

図 4.9. JADE におけるビヘイビア記述例

に JADE 上でのビヘイビア記述例を示す．ビヘイビアの記述には通常，繰り返し実行する処理を記述する *action* メソッドと，終了条件を記述する *done* メソッドを利用する．この例では，SimpleBehaviour クラスを継承し（8 行目），繰り返し実行する処理として自身の名前の出力とカウント変数のインクリメントを（16〜19 行目），終了条件としてカウント変数値のチェック（20 行目）を記述している．ビヘイビアは *addBehaviour* メソッドにより，エージェントの実行プロセスとして登録・起動され（3, 4 行目），この例では 2 つのビヘイビアインスタンスが終了条件が満たされるまでそれぞれ並行に実行される．

拡張すべき点

このように，JADE では複数ビヘイビアの記述により，システムに割り当てられるべき各プロセスの並行実行を実現可能であることから，コンポーネントをビヘイビアにより実装することで，定義 4.3.1 における，実装フレームワークに求められる要件の 1 つ「Control loop 群の制御」を満たす可能性があると考えられる．しかしながら，現状の JADE を Control loop の実装に利用しようとすると，以下のような難しさに直面する．

Conrol loop におけるコンポーネント制御：本研究では，Control loop を構成する要素として，Collect，Analyze & Decide，Act タイプの 3 種類のコンポーネントを定義している．従っ

て，Control loop を実現するためには，これらの複数のコンポーネントが相互に連携する必要がある．これに対し，JADE では複数ビヘイビアによりタスクの並行実行を実現することができるが，他ビヘイビアのタスク実行状態を能動的に確認する手段は提供されていない．また，他ビヘイビアの動作を制御する手段として，ビヘイビアを強制的に中断させる *block* メソッドが提供されているが，処理を中断する際の適切な退避処理などを記述することができない．その結果，単に JADE 上でビヘイビアを用いてコンポーネントを実装すると，適切な退避処理が記述できないことから誤動作を引き起こすといった，コンポーネント連携による問題が発生することになる．

Conrol loop 間の連携：Control loop を複数動作させる場合，競合回避メカニズムを実現するという観点からも，Control loop に対する制御手段が必要である．しかし，先の Control loop 内でのコンポーネント制御の議論と同様に，Control loop を制御するためのメカニズムの実装方法は明らかではない．

複数の Control loop により構成されるシステムの実装に JADE を利用するためには，これらの問題を解消するような拡張が必要となる．

4.3.3　本研究で導入するプログラミングフレームワーク

本研究では，JADE を拡張することで，コンフィギュレーションに基づいて Control loop を構成することが可能なプログラミングフレームワークを実現する．図 4.10 は，本研究で導入するプログラミングフレームワークの構造をクラス図により示したものである．

本拡張では，Control loop を構成するコンポーネントを実現するために，まず，SimpleBehaviour クラスを継承する ComponentBehaviour クラスを導入する．この ComponentBehaviour クラスは，4.3.1 節で定義した要件を満たす Control loop の実現するための基礎となる機能を提供するクラスである．以下，ComponentBehaviour クラスについて述べる．

まず，ComponentBehaviour クラスは，拡張 Darwin モデル [4] を実装すべきコンポーネントの基本モデルとする．拡張 Darwin モデルは，3.4 節でコンフィギュレーションの記述に用いた Darwin モデルに対して，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* と呼ばれるインタフェースを追加した拡張モデルである．

実装するコンポーネントにおいては，拡張 Darwin モデルの特徴の 1 つである mode を変数として持ち，各コンポーネントの状態を mode 変数を通じて外部に公開する．しかしながら，単に拡張 Darwin モデルに従って mode 変数を導入するだけでは，コンポーネント間で制御ループを実現するための制御インタフェースとしては十分ではない．そこで本研究では，コンポーネントの制御を容易にするために，図 4.12 に示す状態遷移をコンポーネントの汎用的な

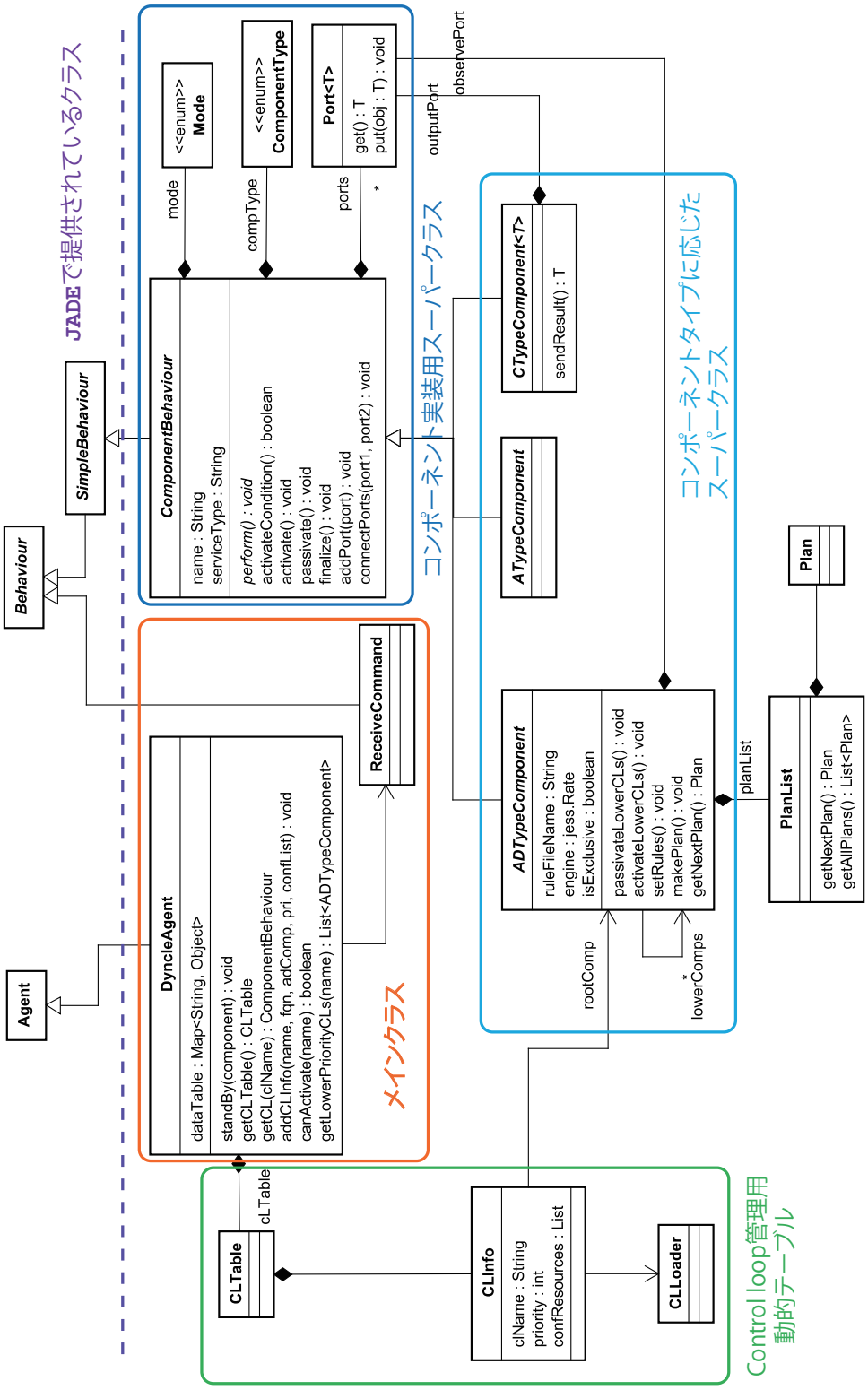

図 4.10. 本研究で導入するプログラミングフレームワークの構造

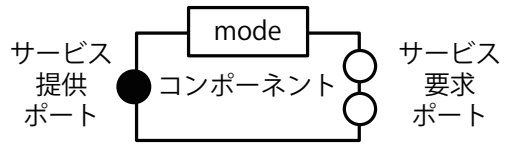

図 4.11. 拡張 Darwin コンポーネントモデル [4]

ライフサイクルとして定義し，状態遷移を実現するメソッドを導入する．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サービス提供のために稼働中であることを表す．一方，ACHIEVED と NOT_ACHIEVED は，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．

ComponentBehaviour クラスを拡張したコンポーネントは，図 4.12 のライフサイクルに従い，活性状態（ACTIVE），待機状態（WAITING），達成状態（ACHIEVED），達成不能状態（NOT_ACHIEVED）の 4 つの状態のうちのいずれかの値をを mode 変数値として取る．また，図 4.12 中の *activate*，*passivate* メソッドはそれぞれコンポーネントの状態を活性状態あるいは待機状態へと遷移させるメソッドであり，この 2 つのメソッドによりコンポーネントの動作を制御する．さらに，*standBy*，*finalize* メソッドはそれぞれ，コンポーネントを生成して待機状態にするメソッドと，コンポーネントを終了させるメソッドである．各メソッドに関しては，活性時の初期化やデータ退避処理など各コンポーネント固有の処理が必要な場合に，必要に応じてオーバーライドすることで実装する．これらの拡張により，状態遷移制御がメソッドとして分離され，振舞い変更時のコンポーネント状態の参照・制御が簡易化される．また，状況の変化に即応するコンポーネント制御手段が外部のコンポーネントに提供されることとなる．

本プログラミングフレームワークでは，このような ComponentBehaviour クラスを継承する，Collect，Analyze & Decide，Act タイプに応じたスーパクラスを提供する．開発者はこれらのスーパークラスを継承したコンポーネントを次節で説明するイディオムに従って実装することで，4.3.1 で定義した要件を満たす Control loop の実現が可能となる．なお，gocc ではゴールモデルから各コンポーネントのタイプを同定可能であることから，生成されるコンポーネント実装用のクラステンプレートでは，図 4.15 の 5 行目に示すように，該当するタイプのスーパークラスが継承元として定義される．

最後に，JADE では，ビヘイビアを実行するアクタを実装するために Agent クラスが存在する．本研究では，この Agent クラスを Control loop を制御，管理するために拡張する．図 4.10 の DyncleAgent クラスは，本研究での Agent クラスの拡張である．本クラスの主な特徴は，Control loop を管理するためのテーブルを保有・管理することと，Control loop をロード・ア

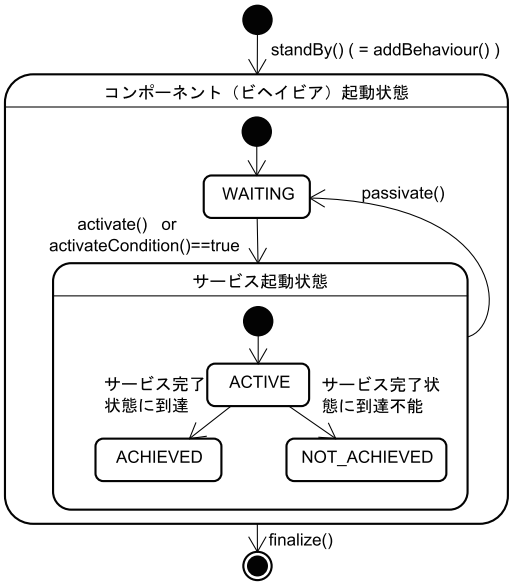

図 4.12. ComponentBehaviour のライフサイクル

ンロードするためのコマンドを受理するビヘイビアをあらかじめ所有しているところにある．本拡張により，Control loop の管理・制御だけでなく動的なデプロイも実現可能となる．

4.3.4　Control loop の制御，デプロイ

本研究で導入するプログラミングフレームワークは，Control loop を構成するコンポーネントに対する制御メソッドを備えている．このメソッド群を利用することで，Control loop を動的に制御することができる．従って，本フレームワークには Control loop を動的に制御および配備（デプロイ）可能なコマンドセットも実装する．表 4.2 は，本プログラミングフレームワークが提供する Control loop の制御・デプロイ関連のコマンドセットである．

表 4.2 のコマンドセットのうち，`activateCL`，`passivateCL` などの Control loop の制御に関しては，本研究で導入するコンポーネントのライフサイクルを制御するメソッド，つまり activate，passivate メソッドなどの組合せにより，実現可能である．開発者によりこれらのコマンドが投入されると，該当する Control loop の Analyze & Decide タイプコンポーネントの activate あるいは passivate メソッドを呼び出し，該当する Control loop を活性化，あるいは待機化させる．

一方の Control loop のデプロイには，Java のクラスローダ（Classloader）[48] を利用する．Java のクラスローダを利用することで，システムが動作している状態においてもコマンドを

表 4.2. Control loop 制御・デプロイコマンドセット

コマンド	オプション	役割
addCL	-n <CLName> -path <classpath> -fqn <AD タイプクラスの FQN> -p <priority> -path <classpath> -conf <conflict objects>	該当 CL の追加
rmCL	-n <CLName>	該当 CL の削除（アンロード）
activateCL	-n <CLName>	CL の活性化
passivateCL	-n <CLName>	CL の待機化
setConf	-n <CLName> -conf <conflict objects> -p <priority>	競合対象の追加，優先順位のセット，変更
rmConf	-n <CLName> -conf <conflict objects>	競合対象の削除

受け付けたタイミングで，新たな Control loop，つまり新たなコンポーネントクラス群の実行ファイル（.class ファイル）をロードすることが可能である．

しかし，Java のクラスローダにおいては，クラスの更新（再読み込み）は容易ではない．クラスを再読み込みするには，一度アンロードした後にリロードする必要があるが，アンロードするには，ロードしたクラスローダを破棄しなければならない．しかしながら，通常のロードでは，システムクラスローダと呼ばれる親ローダがクラスをロードし，システムクラスローダは破棄をすることができないため，通常のロードにより読み込んだクラスはアンロードをすることができず，従って，更新できない．

よって，本プログラミングフレームワークでは，ロード時に Control loop 単位で個別のクラスローダを用い，クラスローダの参照を Control loop 管理用テーブルのエントリである CLInfo クラスに格納しておくことで，アンロード時に備える．rmCL コマンドによるアンロードの指示を受けた場合は，該当する Control loop 固有のクラスローダにより Control loop を構成するコンポーネント群のクラスをアンロードする．これにより，Control loop のデプロイ，アンロードが可能となる[*3]．

[*3] 実際にアンロードを実現するには，Control loop がシステムクラスローダによりロードされないようにするために，Control loop の構成要素となる Class ファイルを Java の ClassPath で定義されていないパス上に配置する必要もある．

4.3.5 進化プロセスの提示

提案するプログラミングフレームワークは，提供コマンドにより，Control loop を配備することが可能である．従って，適切な Control loop の追加，置換手順により，Control loop を動的に進化させることも可能である．

4.2.5 節で述べたように，gocc はゴールモデル，コンフィギュレーションの差分に着目することで，進化に対する変更箇所を同定する．gocc に実装したこの変更箇所同定手法を利用することにより，Control loop 単位での進化の手順を決定することができる．

gocc はゴールモデルの変更内容から，Algorithm 7 のアルゴリズムに従って，進化手順（進化プロセス）を出力する．ここで，進化プロセスとは，プログラミングフレームワークのデプロイコマンドにより構成されるリストである．例えば，進化により新たに Control loop CL1 が追加される場合は，リストの要素は，[`addCL -n CL1; activateCL -n CL1`] となり，Control loop CL2 が変更される場合は，[`rmCL -n CL2; addCL -n CL2; activateCL -n CL2`] となる．ここで，`addCL`，`rmCL` はそれぞれ，該当 Control loop の追加（ロード），削除（アンロード）を，`activateCL` は該当 Control loop の活性化を実行するコマンドである．進化により複数の Control loop が追加，変更される場合は，Algorithm 7 のアルゴリズムに従って，`Uses` ラベルによって定義された利用関係も考慮しながら，コマンドの順序を決定する．

4.4 プログラミングフレームワークによる Control loop の実装

本研究では，定義 4.3.1 の要件を満たすプログラミングフレームワーク上での実装を想定した実装ガイドラインも導入する．提案する開発プロセスでは，プログラミングフレームワークが提供するクラスと Control loop 実現のためのイディオムに基づいて，Control loop の構成要素となるコンポーネントを実装する．提案手法におけるプログラミングモデルは図 4.13 に示すとおりであり，本章では以降，清掃ロボットシミュレータの構築を例に挙げ，提案手法における実装プロセスを説明する．

4.4.1 クラスファイルテンプレートの自動生成

本プログラミングフレームワークは，4.2 節で導入したゴールモデルコンパイラ gocc が生成するコンフィギュレーションに記述されるコンポーネントの実装を支援するものである．gocc が生成するコンフィギュレーションは，コンポーネントとコンポーネント間の接続関係

Algorithm 7 進化プロセス決定アルゴリズム

[入力] *preModel, postModel:* 進化前後の *KAOS* モデル

[出力] *commandList:* 進化プロセスを表現するコマンドリスト

```
1: /* preModel と postModel を比較し，変化のあった Control Loop を検出 */
2: changeCLList ← detectChangeCLs(preModel, postModel);
3: /* Uses の依存関係に従ってソート */
4: index ← 0;
5: while index < changeCLList.size() do
6:     isSwapped ← false;
7:     cl ← changeCLList.elementAt(i);
8:     if (usedCLs ← cl.getUsedCLs()) ≠ null then
9:         for all usedCL in usedCLs do
10:            if changeCLList.indexOf(usedCL) < i then
11:                /* cl の直後に usedCL を挿入 */
12:                changeCLList.remove(usedCL);
13:                changeCLList.insert(usedCL,i);
14:                isSwapped ← true;
15:            end if
16:        end for
17:    end if
18:    if ¬isSwapped then
19:        index++;
20:    end if
21: end while
22: commandList ← [];
23: /* コマンドの発行 */
24: for (i = 0; i < changeCLList.size(); i++) do
25:    cl ← changeCLList.elementAt(i);
26:    if (cl.changeType = "updated" ∨ cl.changeType = "removed") then
27:        commandList.add("rmCL -n <cl.name>");
28:    end if
29: end for
30: for ( i = changeCLList.size()-1;0 < i; i−−) do
31:    cl ← changeCLList.elementAt(i);
32:    if (cl.changeType = "updated" ∨ cl.changeType = "added") then
33:        commandList.add("addCL -n <cl.name>");
34:        commandList.add("activateCL -n <cl.name>");
35:    end if
36: end for
```

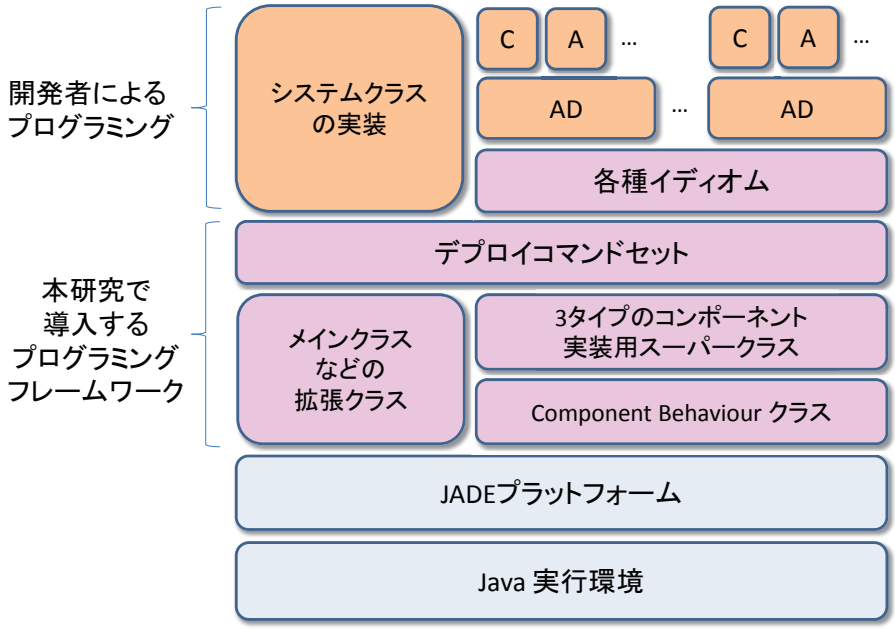

図 4.13. 本研究におけるプログラミングモデル

によりシステム構成を定義するものであり，コンポーネントモデルには，責務を明確化するために Darwin モデルを利用している．gocc の出力コンフィギュレーションでは，各コンポーネントに，Control loop の各アクティビティに該当したラベルが付与されているが，本研究で提案する実装ガイドラインにおいても，付与されたラベル，つまり Control loop において責務を持つアクティビティを考慮して各コンポーネントを実装する．

各コンポーネントの実装にあたっては，コンポーネントクラスのテンプレートを利用する．このテンプレートは，gocc を拡張することで，ゴールモデルからの形式的な取得，つまり自動生成を試みる．gocc は与えられたゴールモデルから，システムクラス，コンポーネントクラスの 2 種類のコードの断片をテンプレートとして生成する．このうち，システムクラスはシステム本体に該当するクラスであり，KAOS ゴールモデル中のエージェントに関する情報をもとに生成する．一方のコンポーネントクラスは，ゴールモデル内の各ゴールに対応付けられたオペレーションに対して一対一の関係で生成される．このオペレーションは，gocc により生成されたコンフィギュレーション上のコンポーネントに対応するゴールに対して，ゴールを達成するための操作として設計者がゴールモデル上に新たに定義するものである．設計者により定義されたオペレーションは，gocc の再実行により，Control loop 内の該当するタイプのコンポーネントクラスとして，実装コードのテンプレートの形式で自動生成される．生成されるコンポーネントクラスのテンプレートは，各コンポーネントタイプのスーパークラスを継承す

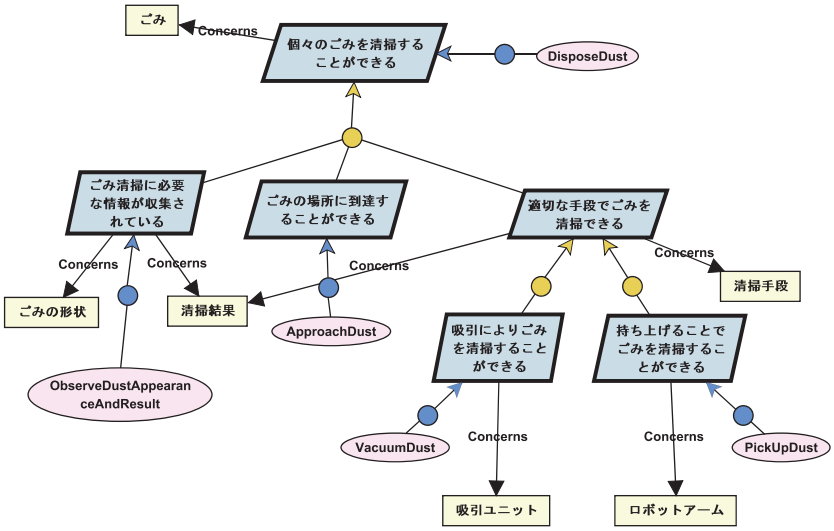

図 4.14. オペレーションの定義例

ることで，あらかじめ Control loop 構成コンポーネント用の基本機能が備わっていることが特徴である．

Example 14　— 図 4.14 は主要ゴール「個々のごみを清掃することができる」に関する Control loop に対してオペレーション群を定義した例である．また，ゴール「個々のごみを清掃することができる」を達成するオペレーション「DisposeDust」に対して生成されるクラステンプレートを図 4.15 に示す．この例では，生成される DisposeDust クラスは，まず，対応するゴールが Analyze & Decide タイプゴールであることから，同タイプのコンポーネント実装を支援するスーパークラスである ADTypeComponent クラスを拡張する（5 行目）[*4]．また，各メソッドの詳細は次節で述べるが，同クラスの動作や活性化時，待機化時の処理を記述するための，perform，activate，passivate メソッドのテンプレートが生成される．これらのメソッド内に各コンポーネント固有の振る舞いを記述することで，同クラスの実装コードを完成させる．

なお，ゴールモデル上にオペレーションを定義した後は，オペレーション名をコンポーネント名とするコンフィギュレーションや階層構造情報，差分検出結果を生成することが可能なように gocc を実装している．これは，実装段階においてはオペレーション名のコンポーネントを扱うため，オペレーション名によるコンフィギュレーションや階層構造情報等の提示が有益であると考えるためである．図 3.25 で示したコンフィギュレーションに対して，オペレーション定義後に生成されるコンフィギュレーションの例を図 4.16 に示す．

[*4] ADTypeComponent クラスの概要については，4.3.3 節で述べる．

```java
package output.firstDev.src;

import org.dyncle.ADTypeComponent;

public class DisposeDust extends ADTypeComponent {

    /** Constructor */
    public DisposeDust(CleaningRobot agent){
        super(agent);
        // TODO Auto-generated constructor block
    }

    /** Behavior statement */
    public void perform(){
        // TODO Auto-generated perform-method block
    }

    /** This method is invoked before activation */
    public void activate(){
        super.activate();
        // TODO Auto-generated activate-method block
    }

    /** This method is invoked before passivation */
    public void passivate(){
        super.passivate();
        // TODO Auto-generated passivate-method block
    }

}
```

図 4.15. 生成されるコンポーネントクラスの例

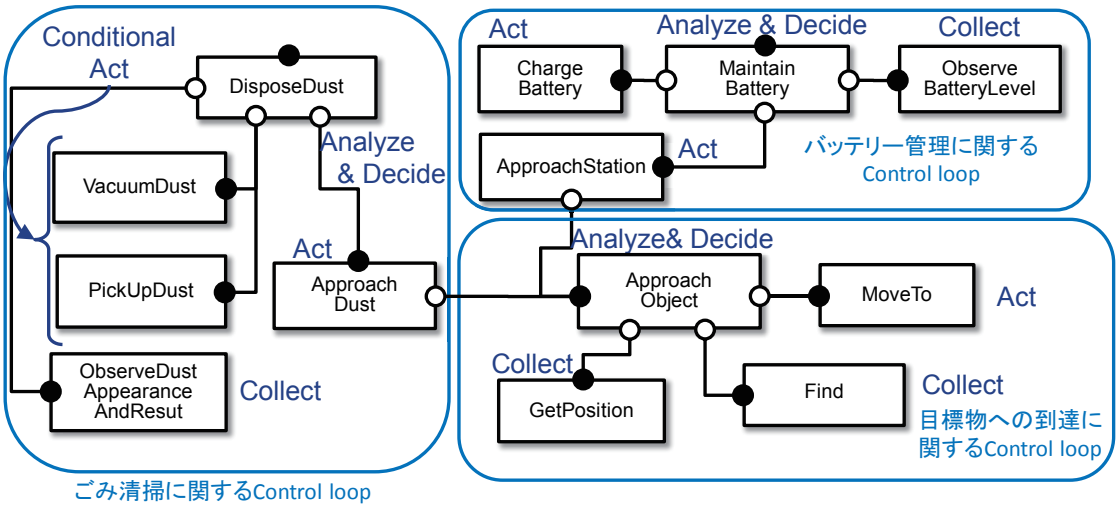

図 4.16. オペレーション名表記によるコンフィギュレーション

4.4.2　イディオムに従ったコンポーネントの実装

本研究では Control loop の機能をコンポーネント上に実装するために，Control loop の各責務に対応する 4 種類のイディオム [49] を導入する．イディオムとは，コードレベルのプログラミング言語に特化した，抽象度の低いパターンである．以下，Control loop の各アクティビティに対応するイディオムを示す．

共通イディオム

Collect, Analyze & Decide，Act タイプの全てのコンポーネントに共通するイディオムとして，状態の可視化と，制御処理の分離，ラッピングによる実装が挙げられる．

状態の可視化：まず，本研究におけるコンポーネントの責務割り当てでは，Analyze & Decide タイプのコンポーネントが他のタイプのコンポーネントの状態を把握する必要がある．従って本研究では，各コンポーネントに対して，ライフサイクルに従った状態を外部に可視化する責務を持たせる．この実装には，拡張 Darwin モデルで導入された mode の概念を利用し，各コンポーネントで状態遷移の時点で mode 変数に変更後の状態値をセットする．例えば，コンポーネントのプロセス開始時には mode の値を `ACTIVE` にセットし，サービス完了状態に到達した場合は mode の値を `ACHIEVED` に，到達できない状態になった場合は `NOT_ACHIEVED` にセットする．これにより，参照側のコンポーネントにおいては，

＜コンポーネント名＞.`mode`

という記述で現在の状態を参照できるようになる．

制御処理の分離：コンポーネントのサービスを実現する記述と，コンポーネントの制御，つまり状態遷移時に必要となる処理の記述を分離する．コンポーネントが提供すべきサービスを実現する処理は，perform メソッドに記述し，活性時の初期設定や退避処理は，それぞれ activate, passivate メソッドに記述する．これらの処理は，スーパークラスの各メソッドのオーバーライドにより記述する．特に，Analyze & Decide タイプのコンポーネントにおいては，Control loop の活性化および待機化の責務を持つため，同メソッド内において，Collect タイプや Act タイプコンポーネントの活性化，非活性化の処理を，*activate*，*passivate* メソッドを呼び出す形で記述する．

以上の「状態の可視化」と「制御処理の分離」については，3 タイプのコンポーネントのスーパークラスに該当する ComponentBehaviour クラスが提供する機能を利用した記述である．

ラッピングによる実装：Collect タイプや Act タイプにおいては，情報収集や処理の実現に，センサやアクチュエータに相当する外部コンポーネントを利用する場合も考えられる．このよ

うな場合には，外部コンポーネントをラッピング（wrapping）する形態で，Collect タイプや Act タイプコンポーネントを実装する．例えば，Act タイプコンポーネントにおいて環境に対して影響を与える外部コンポーネントを利用する場合は，activate メソッド内に該当コンポーネントを駆動する記述を加え，外部コンポーネントのサービスが終了し，Act タイプのゴールが達成されたと判断した時点で mode の値を `ACHIEVED` にセットする．また，passivate メソッド内には外部コンポーネントを停止させる動作を記述する．

Collect タイプイディオム

Collect タイプコンポーネントにおいては，外部環境やシステムの内部状態に関する情報を収集し，収集結果を Analyze & Decide タイプのコンポーネントに伝える責務を持つ．従って，Collect タイプイディオムは以下の構造を持つ．

```
＜情報収集＞
try{
    sendResult(＜収集結果＞);
}catch (PortException e){
    ＜例外時処理＞
}
```

Collect タイプコンポーネントでは，まず，入力変数として定義されている情報を収集する．収集結果の Analyze & Decide コンポーネントへの伝達には，Darwin モデルにおいてコンポーネント間を接続するポートを利用する．本研究で導入するプログラミングフレームワークでもポートは実装されており，特に CTypeComponent を継承することで，ポートを利用した情報送信用の `sendResult` メソッドが利用できる．情報の収集には，センサモジュールなどを利用する場合もあれば，デザインパターンにおける Observer パターン [50] を利用した，他コンポーネントの状態変化検知による情報収集も考えられる．

Analyze & Decide タイプイディオム

Analyze & Decide タイプコンポーネントは，Collect タイプのコンポーネントからの収集結果をもとに状況を分析（Analyze）し，処理部，つまり Act タイプのコンポーネントの制御内容を決定（Decide）し，決定内容に従って Act タイプコンポーネントを制御する責務を持つ．従って，以下の構造を持つ if-文を用いる．

```
if (＜分析結果＞){
    ＜制御内容＞
}
```

ここで < 分析結果 > には，現在の状況に対する分析結果を記述する．現在の状況としては，Collect タイプのコンポーネントにより得られた情報と，Control loop 内で把握可能な情報の 2 種類がある．後者としては，特に，Act タイプコンポーネントの状態が挙げられるが，本研究では，mode 変数によりコンポーネントのライフサイクル上での状態を可視化しているため，この mode の値を参照して Control loop の内部状態を把握することができる．例えば，清掃ロボットシミュレータの例において，ごみへ近づいた状態であるかを分析するには，ごみへ近づくという Act タイプのコンポーネント approachDust のサービスが完了している，つまり完了状態に到達したかどうかを確認する

```
approachDust.mode==Mode.ACHIEVED
```

といった条件式を用いることで，現在の状況を分析することができる．

一方，< 制御内容 > に対しては，Collect，Act タイプコンポーネントに対しての activate，passivate メソッドを用いた動作制御や，ポートを通じた Act タイプコンポーネントへの操作指示の送信処理を記述する．

Act タイプイディオム

Control loop における処理部は Act タイプコンポーネントに該当し，従って，Control loop が提供するサービスの実現に関する責務を持つ．Act タイプイディオムは，共通イディオムに従ったものであり，perform メソッドに処理部の動作を記述し，実行状況により，サービスの提供状態を示す mode の値を随時変更する．また，退避処理を passivate メソッドに記述する．

提案手法では，gocc により生成されたテンプレート上に，これらのイディオムを用いて固有の処理を記述することで，Control loop を構成する各コンポーネントを実装する．

4.5　評価

提案手法の有用性を検証するために，提案手法に従ってシミュレータ上における清掃ロボットの進化実験を実施した．本節では実験内容とその結果を示し，本章で導入した gocc とプログラミングフレームワークの有効性を評価する．

4.5.1　評価実験の概要

本実験は大きく 2 種類の実験に分類される．まず，本研究で導入した gocc，プログラミングフレームワークを利用したシステム実装が可能であることを確認するために，ゴールモデル

上で同定された Control loop を構成するコンポーネントを実装イディオムに従って実際に構築し，その挙動とコンポーネントの記述方法を分析した（実験 1）．続いて，gocc とプログラミングフレームワークのソフトウェア進化に対する有効性を評価するために，進化シナリオに沿った Control loop の追加，変更実験を実施した（実験 2）．

実験において対象とするシステムは，本論文の例題として用いたシミュレータ上の清掃ロボットである．本実験では，実験 1 において清掃ロボットのコンフィギュレーションとして図 4.19 の構成を用いた．このコンフィギュレーションは，2.5 節で説明した清掃ロボットの初期要求（図 4.17）に対応するコンフィギュレーションであり，ゴールモデルにオペレーションを割当てた後に，gocc により生成されたものである．実験 2 においては，進化後のコンフィギュレーションとして，図 4.24 のコンフィギュレーションを用いた．このコンフィギュレーションは，図 4.19 のコンフィギュレーションに対してごみ積載量管理機能が追加されたものである．

実験では，ごみやバッテリーステーションなどが配置されたいくつかのフィールドを用意し，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．図 4.18 は実験で用いたフィールドの一例である．

4.5.2 実験 1：実現可能性

まず，提案手法に基づいて構築した Control loop の挙動を確認するために，清掃ロボットの振る舞いを観測した．図 4.18 の配置パターンに対する実験結果として，清掃ロボットが出力したログの一部を図 4.20 に示す．図 4.20 からは，清掃ロボットがごみのある座標 (6, 3) に向かって移動中（173〜176 行目）にバッテリーが低下したため（177 行目），座標 (4, 3) に存在するバッテリーステーションへの移動へと動作を切り替えたことが分かる（178〜195 行目）．その後，バッテリーステーションに到着して充電した後に（196〜199 行目），ごみ収集を再開し（200 行目），バッテリーステーションから最も近いごみを見つけて移動（202 行目〜）していることも確認できる．このように，実装した清掃ロボットでは，ごみ清掃に関する Control loop とバッテリー管理に関する Control loop が動作し，バッテリーの残量がある間はごみの清掃を続け，残量が少なくなるとバッテリーステーションへ移動し充電するという，各 Control loop の主要ゴール達成のための動作を実現していることが確認できた．

続いて，提案手法に従って実装したコンポーネントの記述方法を分析した．まず実験結果から，清掃ロボットはごみ，あるいはバッテリーステーションを見つけ，その後目標オブジェクトへの移動を開始していることが確認できる．この目標物への到達に関する動作を実現するのは，図 4.19 のコンフィギュレーションにおける ApproachObject コンポーネントである．図

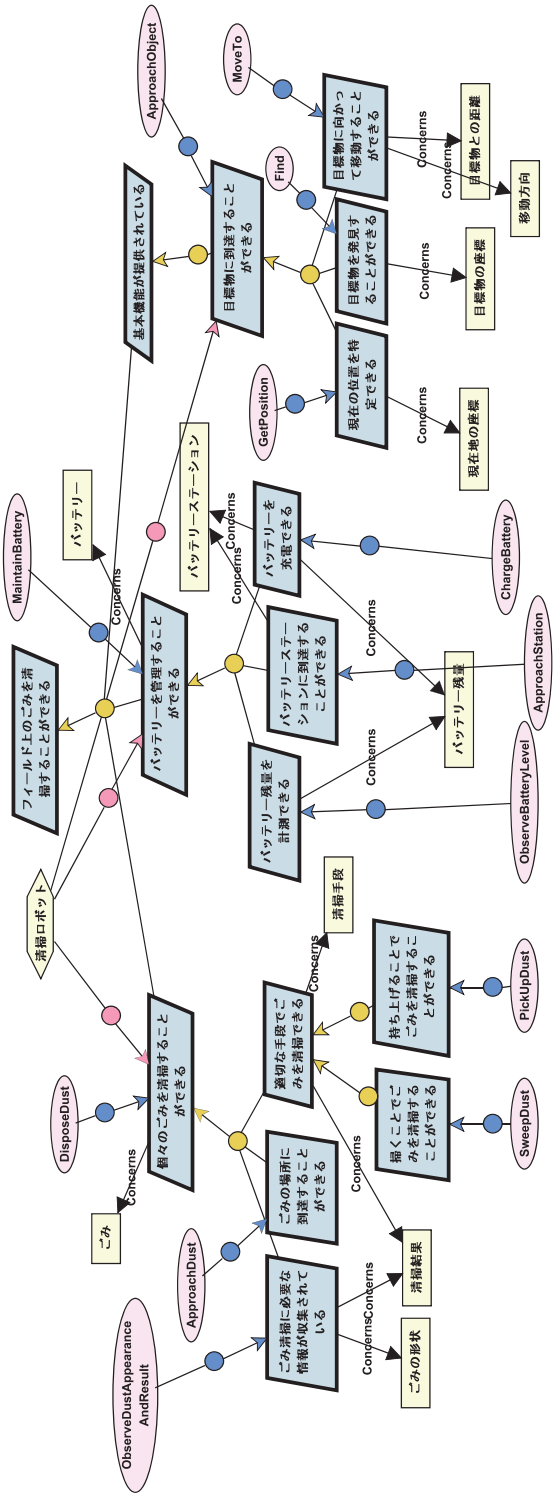

図 4.17. 清掃ロボットに対する初期要求モデル（図 3.18 のゴールモデルに対してオペレーションを定義したもの）

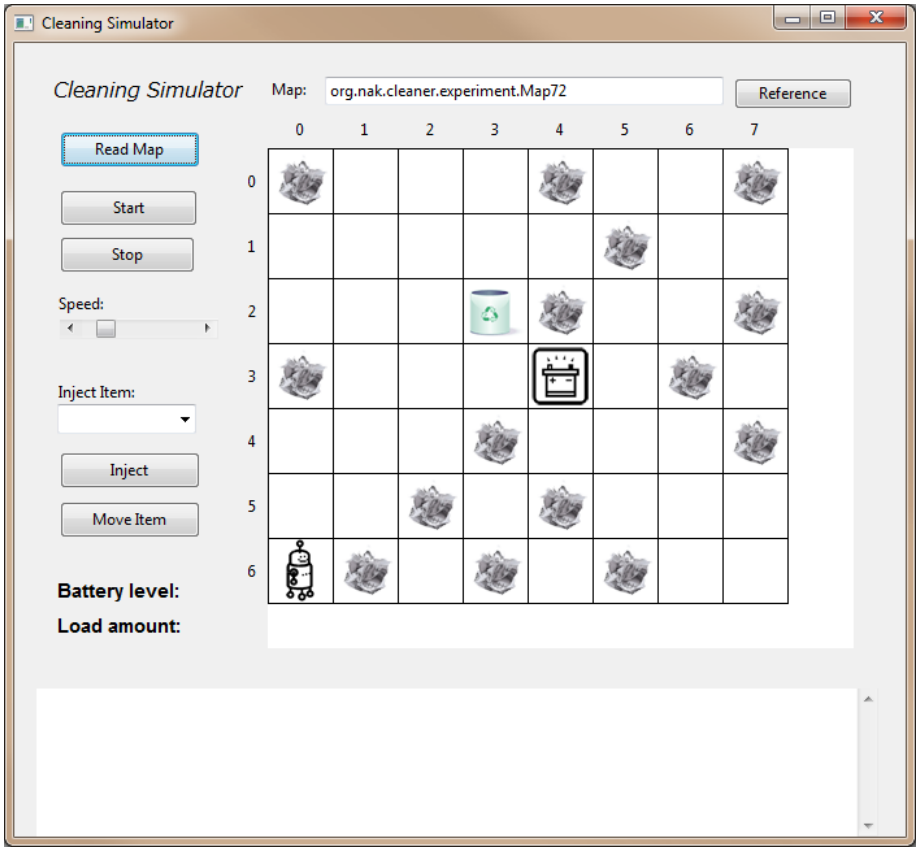

図 4.18. 実験で用いたフィールドの例

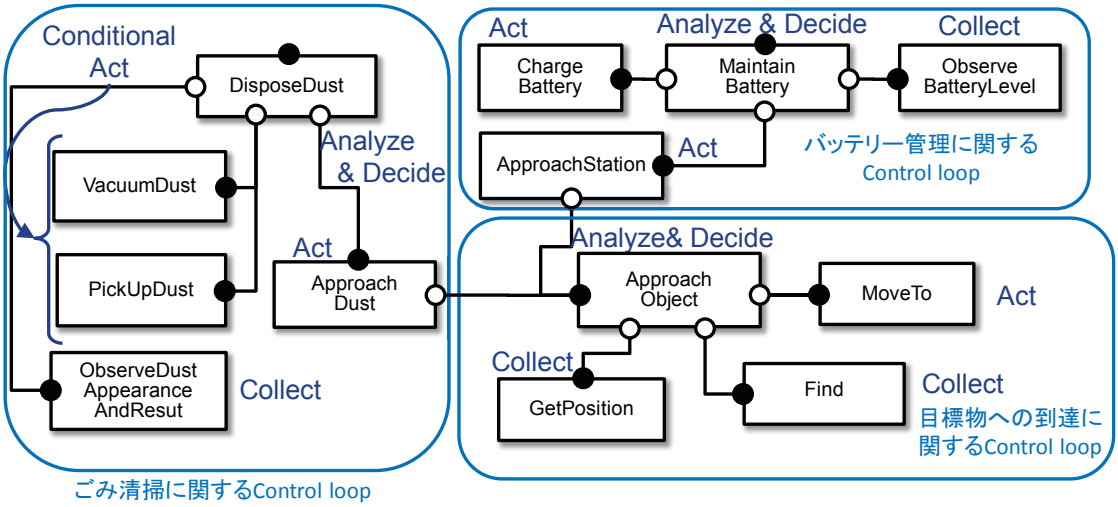

図 4.19. 実験 1 で用いたコンフィギュレーション

```
...
173:     [java]           Dust is found at (6,3).
174:     [java] Component: Move
175:     [java]            Moved to (6,6).
176:     [java]                            Rest battery: 28
177:     [java] *** Robot noticed the battery level is low.
178:     [java] Component: ApproachStation
179:     [java] Component: ApproachObject
180:     [java] Component: Find
181:     [java]           targetType: ChargeStation
182:     [java]                            Rest battery: 28
183:     [java]           ChargeStation is found at (4,3).
184:     [java] Component: Move
185:     [java]            Moved to (5,6).
          ...
193:     [java]            Moved to (4,3).
194:     [java]                            Rest battery: 8
195:     [java] Arrive at (4,3).
196:     [java] Component: ChargeBattery
197:     [java]           Fill battery ...
198:     [java]                            Rest battery: 100
199:     [java]           Battery is full
200:     [java] Component: ApproachDust
201:     [java] Component: ApproachObject
202:     [java] Component: Find
203:     [java]           targetType: Dust
204:     [java]                            Rest battery: 100
205:     [java]           Dust is found at (4,2).
206:     [java] Component: Move
207:     [java]            Moved to (4,2).
          ...
274:     [java]           Dust is found at (6,3).
275:     [java] Component: Move
276:     [java]            Moved to (6,1).
277:     [java]                            Rest battery: 64
278:     [java]            Moved to (6,2).
279:     [java]                            Rest battery: 60
280:     [java]            Moved to (6,3).
281:     [java]                            Rest battery: 56
282:     [java] Arrive at (6,3).
283:     [java] Change to cleaning method.
284:     [java] Got appearance: LITTER
285:     [java]           Identified object type :LITTER
286:     [java] Determined cleaning method: SweepDust
287:     [java] Component: SweepDust
288:     [java]           Sweep dust at (6,3).
289:     [java]                            Rest battery: 52
290:     [java]           Dust at (6,3) was swept.
291:     [java]                             load amount: 200
292:     [java] VirtualRobot.java: OverloadItemsError
```

ごみに
向かって移動

←バッテリー低下を検知

ステーションに
向かって移動

充電

ごみ清掃を再開

←積載量オーバー

図 4.20. 清掃ロボットの実行ログ（抜粋）

```
 1: public void perform() {
       ...
23:    if(find.mode == Mode.WAITING){
24:       connectPorts(findPort, find.getProvidedPorts().get(0));
25:       try {
26:          findPort.put((Object)targetType);
27:       } catch (PortException e) {
28:          e.printStackTrace();
29:       }
30:       find.activate();
31:    }
32:    if(find.mode==Mode.ACHIEVED){
33:       if(moveTo.mode == Mode.WAITING){
34:          targetPoint = null;
35:          if(findPort.isArrive()){
36:             try {
37:                targetPoint = (Point)findPort.get();
38:             } catch (PortException e) {
39:                e.printStackTrace();
40:             }
41:          }
42:       connectPorts(moveToPort, moveTo.getProvidedPorts().get(0));
43:       try {
44:          moveToPort.put(targetPoint);
45:       } catch (PortException e) {
46:          e.printStackTrace();
47:       }
48:       moveTo.activate();
49:    }
       ...
```

図 4.21. イディオムの適用例（ApproachDust コンポーネントの perform メソッド）

4.21 は，ApproachObject コンポーネントの perform メソッドの一部を示したものである．ここでは周囲の物体を探索する Act タイプコンポーネントである Find コンポーネントの mode 値が “WAITING”，つまり待機中である場合には（23 行目），Find コンポーネントに，探索対象となる物体（ごみ，あるいはステーション）を指定し，Find コンポーネントを activate メソッドを用いて活性化させている（30 行目）．また，32 行目以降は，Find コンポーネントが “ACHIEVED” となったことを判定して，移動を実現する Act タイプコンポーネントである MoveTo コンポーネントを activate メソッドを用いて活性化させている（48 行目）．これは，本研究で導入する Analysis& Decide タイプイディオムを適用した記述である．

一方，図 4.22 は，バッテリー管理に関する Control loop を構成する ObserveBatteryLevel コ

```
1: public void perform() {
2:    try {
3:       sendResult(new Integer(((CleaningRobot)myAgent).simulator.getBattery()));
4:    } catch (PortException e) {
5:       e.printStackTrace();
6:    }
7: }
```

図 4.22. ObserveBatteryLevel コンポーネントの perform メソッド

ンポーネントの perform メソッドの記述を示したものであり，同コンポーネントは Collect タイプに該当するが，バッテリー残量の情報取得に Collect タイプイディオムを適用した例に該当する．

さらに，図 4.20 のログからはバッテリーステーションへの移動の前後で，目標とするごみが座標 (6, 3) の紙くずから座標 (4, 2) の紙くずへと変わっていることも確認できる．これは，目標物への到達に関する Control loop を利用する Control loop が切り替わる際に，DisposeDust の passivate メソッドが呼び出されることで適切な退避処理が実行されたことによるものである．図 4.23 は DisposeDust コンポーネントの passivate メソッドの実装を示したものであるが，本メソッドでは，現在動作している Act タイプコンポーネントを passivate メソッドにより待機状態（waiting）に遷移させ（5 行目），目標地点であるごみの座標をリセットしている（9 行目）ことがわかる．このように，ComponentBehaviour クラスの passivate メソッドをオーバーライドすることで，Control loop のゴールを達成途中であっても適切な退避処理が実現可能であることが確認できた．

以上のように，本研究で導入する各イディオムに従った実装により，各 Control loop の実装と，複数 Control loop の並行動作が実現可能であることが確認できた．

4.5.3　実験 2：進化への対応

続いて，本章で導入したツール群のソフトウェア進化時に対する有効性を評価するために，実験 1 で構築した清掃ロボットに対して Control loop の追加，変更による進化実験を実施した．本実験では，2 章で示した 2 種類の進化シナリオに基づいて実際に Control loop を追加，更新した．以降，進化実験の結果について述べる．

進化 1（ごみ積載量管理機能の追加）

実験 1 においては，図 4.20 の実行ログの最終行からも分かるように，清掃ロボットが保有できるごみ保有量を越えたために，フィールド上の全てのごみを清掃する前に，ロボットが異

```
 1: public void passivate(){
 2:    super.passivate();
 3:
 4:    if(currentComponent!=null){
 5:       currentComponent.passivate();
 6:       currentComponent = null;
 7:    }
 8:
 9:    nextDust = null;
10:    isFirstTime = true;
11:
12:    observe.passivate();
13:    if(observePort.isArrive()){//clean remained collected data
14:       try {
15:          targetAppearance = (Appearance)observePort.get();
16:       } catch (PortException e) {
17:          e.printStackTrace();
18:       }
19:    }
20: }
```

図 4.23. DisposeDust コンポーネントの passivate メソッド

常終了した．そこで，2.5 節で示した進化シナリオに従って，gocc の出力情報を利用しながら，プログラミングフレームワーク上に構築した清掃ロボットに対して，ごみ積載量管理機能を追加した．

まず，ごみ積載量管理機能の追加に伴う，進化後のコンフィギュレーションを図 4.24 に示す．進化前後のコンフィギュレーションを比較すると，進化後はごみ積載量管理に関する Control loop とロボットアーム制御に関する Control loop が新たに追加されることとなり，これは gocc が出力する差分検出結果（図 4.25 中の青字で示された行）からも確認できた．また，gocc のクラステンプレート生成機能により，以下の 7 つのコンポーネントテンプレートが生成された．

- ごみ積載量管理に関する Control loop を構成するもの
  - MaintainLoadAmount（Analyze & Decide タイプ）
  - ObserveLoadAmount（Collect タイプ）
  - ApproachDustBin（Act タイプ）
  - UnloadDust（Act タイプ）
- ロボットアームに関する Control loop を構成するもの
  - ControlArm（Analyze & Decide タイプ）

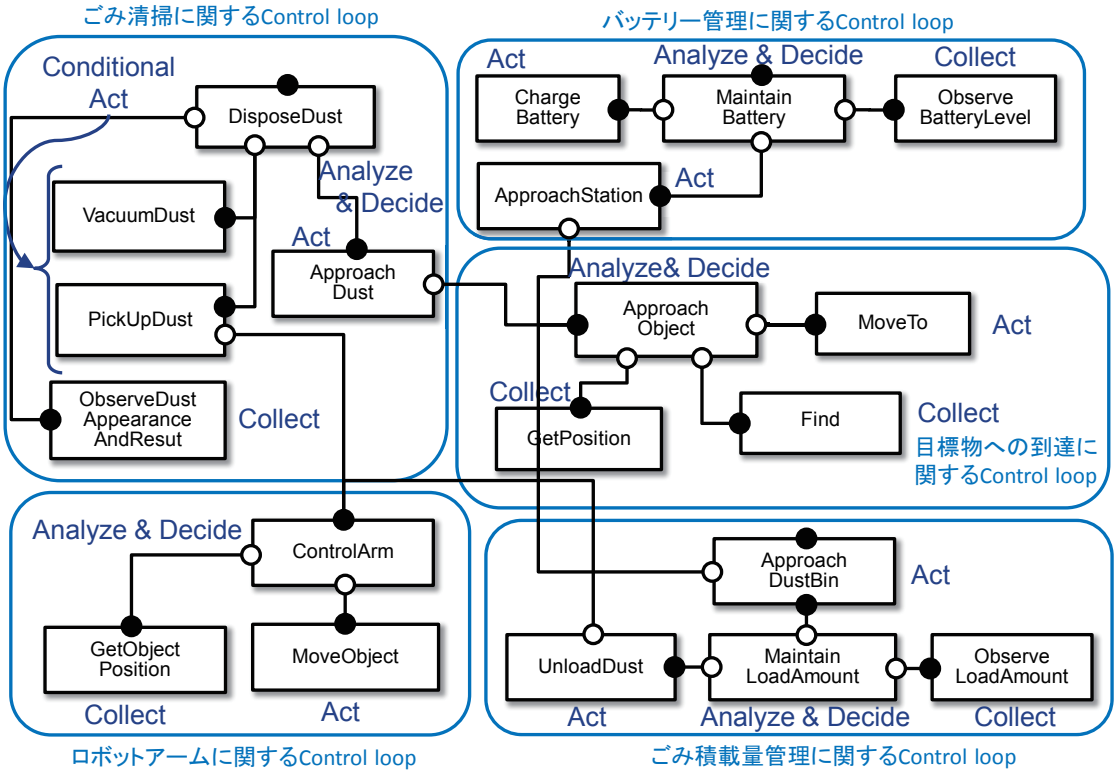

図 4.24. 実験 2 で用いたコンフィギュレーション

– GetObjectPosition（Collect タイプ）
– MoveObject（Act タイプ）

進化に際して，gocc は競合チェックリストとして表 4.1 のリストを出力した．この競合チェックリストに関しては，4.2.4 節での議論の通り，競合利用する Control loop に対して 3 つの Control loop の優先順位を決定し，バッテリー管理機能，積載量管理機能，ごみ清掃機能の順で動作の優先順位を決定した．

また，本進化に対して gocc は，進化プロセスとして，次のコマンド系列を出力した．

```
[rmCL -n DisposeDust; addCL -n ControlArm; activateCL -n ControlArm;
    addCL -n DisposeDust; activateCL -n DisposeDust;
    addCL -n MaintainLoadAmount; activateCL -n MaintainLoadAmount]
```

新たな Control loop を構成する上記の 7 つのコンポーネントをイディオムに従って実装し，また，ロボットアームを利用する PickUpDust コンポーネントを修正した後，進化プロセスに従って下記のコマンドを順次投入した．

```
rmCL -n DisposeDust
```

```
 1: 1c1
 2: < %% #Components: 13         #Connections: 12      #Control loops: 3
 3: ---
 4: > %% #Components: 20         #Connections: 20      #Control loops: 5
 5: 3a4,12
 6: > addComponent(MaintainLoadAmount, ANALYZE_and_DECIDE)
 7: > addPort(MaintainLoadAmount, p0)
 8: > addPort(MaintainLoadAmount, r0)
 9: > addPort(MaintainLoadAmount, r1)
10: > addPort(MaintainLoadAmount, r2)
11: > addComponent(ControlArm, ANALYZE_and_DECIDE)
12: > addPort(ControlArm, p0)
13: > addPort(ControlArm, r0)
14: > addPort(ControlArm, r1)
15: 18a28,39
16: > addComponent(ObserveLoadAmount, COLLECT)
17: > addPort(ObserveLoadAmount, p0)
18: > addComponent(UnloadDust, COLLECT)
19: > addPort(UnloadDust, p0)
20: > addPort(UnloadDust, r0)
21: > addComponent(ApproachDustBin, ACT)
22: > addPort(ApproachDustBin, p0)
23: > addPort(ApproachDustBin, r0)
24: > addComponent(MoveObject, ACT)
25: > addPort(MoveObject, p0)
26: > addComponent(GetObjectPosition, COLLECT)
27: > addPort(GetObjectPosition, p0)
28: 40a62
29: > addPort(PickUpDust, r0)
30: 42a65,69
31: > connect(MaintainLoadAmount.r0, ObserveLoadAmount.p0)
32: > connect(MaintainLoadAmount.r1, UnloadDust.p0)
33: > connect(MaintainLoadAmount.r2, ApproachDustBin.p0)
34: > connect(ControlArm.r0, MoveObject.p0)
35: > connect(ControlArm.r1, GetObjectPosition.p0)
36: 52a80,81
37: > connect(UnloadDust.r0, ControlArm.p0)
38: > connect(ApproachDustBin.r0, ApproachObject.p0)
39: 54a84
40: > connect(PickUpDust.r0, ControlArm.p0)
```

図 4.25. 進化 1 におけるコンフィギュレーション上での差分検出

```
addCL -n ControlArm
       -fqn robot.de.comp.additional.ControlArm
activateCL -n ControlArm
addCL -n DisposeDust
       -fqn robot.de.comp.additional.DisposeDust
       -p 5 -conf ApproachObject, ControlArm
activateCL -n DisposeDust
addCL -n MaintainLoadAmount
       -fqn robot.de.comp.additional.MaintainLoadAmount
       -p 3 -conf ApproachObject, ControlArm
```

```
 1: < Assignment(清掃ロボット, 目標物に到達することができる)
 2: > Assignment(清掃ロボット, 障害物を避けながら目標物に到達することができる)
 3: < Goal(目標物を発見することができる)
 4: < Goal(目標物に到達することができる)
 5: > Goal(目標物・障害物を発見することができる)
 6: > Goal(障害物を避けながら目標物に到達することができる)
 7: > Entity(障害物の座標)
 8: < Concerns(目標物に到達することができる, フィールド)
 9: < Concerns(目標物を発見することができる, 目標物の座標)
10: > Concerns(目標物・障害物を発見することができる, 目標物の座標)
11: > Concerns(障害物を避けながら目標物に到達することができる, フィールド)
12: > Concerns(目標物・障害物を発見することができる, 障害物の座標)
```

図 4.26. 障害物回避機能追加に対するゴールモデル上での差分検出

```
activateCL -n MaintainLoadAmount
```

ここで, addCL のオプションである “-p” に指定されている優先度値は, 競合する他の Control loop との間での優先度である．今回は，競合利用する Control loop である “ApproachObject” と “ControlArm” に対して，ごみ積載量管理の方がごみ清掃より優先すべきであるため，ごみ積載量管理に関する Control loop の方に高い優先度（小さい優先度値）を割り当てる．

上記コマンド群を投入することにより，動作中のロボットに対してごみ積載量管理機能が追加され，積載量が一定値を越えると，ごみを捨てるためにごみ箱へと向かう動作が実現されることが確認できた．

進化 2（障害物回避機能の追加）

続いて，障害物回避機能の追加に関する進化実験を実施した．まず，進化後のゴールモデルを gocc に入力した際のコンフィギュレーションは図 3.34 に示す通りであり，ゴールモデルに対する差分検出結果として図 4.26 が出力された．

オペレーションに対するコンフィギュレーションは，前回の進化時と同様，図 4.24 の通りであり，オペレーションに対するコンフィギュレーション上での差分は出力されなかった．

続いて，ゴールモデル上での差分検出結果をもとに，目標物への到達に関する Control loop を構成するコンポーネントの実装を変更した．具体的には，障害物を検知するために Collect タイプである Find コンポーネントを変更し，また，障害物の座標を管理し，障害物を避けた移動を実現するよう Analyze & Decide タイプである ApproachObject コンポーネントの実装を変更した．

本進化に対して gocc は，進化プロセスとして以下のコマンド系列を出力した．

```
[rmCL -n ApproachObject; addCL -n ApproachObject;]
```

競合チェックリストは前回の進化時と同様のリストであったが，既に優先順位の設定により競

合への対応は実装されているため，今回の進化では，新たな対応は不要と判断した．

コンポーネントの変更が完了した後，進化プロセスに従って下記のコマンドを順次投入した．

```
rmCL -n ApproachObject
addCL -n ApproachObject
       -path ../EvolExperiment/bin
       -fqn robot.de.comp.ApproachObject
activateCL -n ApproachObject
```

これらのコマンドを投入することにより，動作中のロボットに対して，障害物を回避する機能が追加され，障害物を新たにフィールド上に設置しても，回避した移動が実現されることが確認できた．なお，移動中のロボットに対してコマンドを投入した場合は，`rmCL` コマンドを投入してから，`activateCL` コマンドを投入するまでの間は，目標物への到達，つまり移動に関する Control loop が停止するため，ロボットが移動を停止することが確認された．

4.5.4 要件に対する考察

清掃ロボット構築実験の実験結果をもとに，4.3.1 節で定義した要件に対して，本章で導入した開発支援ツール，プログラミングフレームワークを評価する．

Control loop の実現容易性

実験 1 の結果から，本研究で導入するプログラミングフレームワークは，Control loop を構成する各コンポーネントのスーパークラスとコンポーネントの制御メカニズムを提供することで，Control loop の実装を支援していることが確認できる．また，実験 1，実験 2 の結果から，コンフィギュレーションをはじめ，`gocc` により出力された各種情報と，プログラミングフレームワークが提供する Control loop 実現のための拡張クラスおよびイディオムを利用することで，Control loop の実装が可能であり，Control loop の系統的な実装支援が一定のレベルで実現できたと言える．

さらに，実験 2 の結果より，差分として検出される Control loop を追加実装あるいは変更することで，ソフトウェア進化が実現可能であることが確認できた．ただし，イディオムによる支援はあるが，システムの動的な振舞いについては更なる支援の余地があると考えられる．また，差分検出についても，検出結果からある程度の範囲での変更箇所の絞り込みは可能であるが，ゴールモデルへの情報追加による，更なる精度向上も進化時の影響範囲同定に有益であると考えられる．

Control loop の並行実行

本研究で導入したプログラミングフレームワークでは，実行基盤とした JADE に，ビヘイビアとして実装したコンポーネントを制御するためのメソッドとライフサイクルを追加拡張することで，コンポーネントの並行実行を実現している．2 種類の実験結果からも，複数の Control loop により構成されるシステムにおいても，Control loop が並行に動作することで，期待する動作が実現できることが確認できた．ただし，Control loop を並行実行させる場合，Control loop 間での競合に留意する必要がある．本プログラミングフレームワークにおいては，Control loop に優先順位を付与し，この優先順位と activate, passivate メソッドを利用することで，Control loop 間の競合回避制御を実装可能であるが，状況によって優先させる Control loop が異なる場合などは，競合回避手順の実装が複雑になる可能性がある．本研究においては，利用される側の Control loop が競合回避の責務を持つため，競合回避手順は該当 Control loop の Analyze & Decide タイプコンポーネントに記述されることとなる．提案手法においては，gocc が競合が発生する可能性のある状況を競合チェックリストとして出力するが，厳密な競合発生状態については，開発者による特定・分析が必要となる．

一方で，提案手法ではシステムの構成要素を Control loop の単位で同定するが，本プログラミングフレームワークを利用することにより，実行中のシステムに対する Control loop の追加，削除が可能であることが確認できた．本研究では，Control loop 制御の責務を Analyze & Decide タイプコンポーネントに割り当てているが，実験 2 の進化 1，進化 2 において投入したデプロイコマンドは同コンポーネントに対する制御により実現したものである．実験結果から，Control loop に対する制御についても，本研究で導入したコンポーネント実装の activate, passivate メソッドにより，実現されていることが確認できた．

4.5.5　コンフィギュレーションに対する考察

最後に，実験結果をもとに，gocc により生成されるコンフィギュレーションを Kruchten の 4+1 アーキテクチャビュー [51]，すなわち， 論理的側面（*logical view*），プロセス的側面（*process view*），開発的側面（*development view*），物理的側面（*physical view*） およびシナリオ（*scenarios*）の観点から評価する．

まず，クラス図などで表現されるクラス間の関係に着目する論理的側面（*logical view*）としては，提案手法ではゴールモデル上に記述されたゴールとゴール間の関係により，コンポーネントとその接続形態，つまりコンフィギュレーションを決定する．このようなコンフィギュレーション生成法のもとでは，ゴールが詳細に分析（分解・洗練化）されているゴールモデル

に対しては必要以上のコンポーネントが生成される可能性があるが，gocc では，各 Control loop のアクティビティに対応するゴールのみを変換対象とすることで，生成されるコンポーネント数を抑制している．また，複数の機能に対して求められる横断的な要求 [52] に対しても，Uses ラベルを用いることにより，各機能との関連がゴールモデル上で定義され，これが gocc により結合される．従って，論理的側面の観点からは，提案手法では，システムに必要な Control loop 群をゴールモデルから抽出し，各 Control loop に必要なコンポーネントと，Control loop 間の関係をゴールモデルから効率的に抽出したコンフィギュレーションを生成していると言える．

一方，システムの実行プロセスに着目するプロセス的側面（*process view*）に対しては，まず，提案手法では複数の Control loop を並行動作させるため，プロセス実行のオーバーヘッドが生じる可能性がある．また Control loop 内のアクティビティを分割してコンポーネント実装する場合は，これらのコンポーネントの実行プロセス間でのデータ送受信やプロセス実行のオーバーヘッドも生じる可能性がある．従って，コンフィギュレーション上で得られたコンポーネントの実装方法については，性能要求や実装コスト，各コンポーネントの進化の頻度も考慮して検討する必要がある．

それに対して，提案手法のような，要求モデルからのコンフィギュレーション生成は，開発的側面（*development view*）に対する効果が期待できる．まず，要求モデルであるゴールモデルからコンフィギュレーションが決定されるため，実験 2 の実験結果が示すように，要求変更への対応が容易であるとともに，要求と設計・実装間のトレーサビリティも確立したコンフィギュレーションであると言える．また，提案手法で導入する KAOS モデリングの制約規則により，複数の Control loop が分割された形態でゴールモデルが記述され，結果として，要求分析結果に従って分割された複数の Control loop を個別に設計・実装することが可能となる．加えて，Collect，Analyze & Decide といった役割に応じてコンポーネントが分割されて抽出されるため，各コンポーネントの設計・実装に対してガイドラインを提供しやすい点も本コンフィギュレーションの特徴であると考えられる．

ソフトウェアの配置環境などの物理的側面（*physical view*）としては，本研究では，ソフトウェアシステムの構成要素として複数の Control loop を抽出・同定しているため，得られるコンフィギュレーション上に存在する Control loop 群は，一般には同一マシン上に配備し，それらの実行プロセスを制御することが期待される．この前提を満たすためには，本章で示したような，各コンポーネントを並行に動作させ，コンポーネント間での制御を容易に実装できるような実装フレームワークおよびミドルウェアが求められる．

最後に，動的側面を扱うシナリオ（*scenarios*）に関しては，ゴールモデル上での記述方法が影響を与える．例えば，実験で用いた図 4.17 のゴールモデルにおける，ごみ清掃プロセスに

関する詳細化（ゴール「個々のごみを清掃することができる」に対する詳細化）やバッテリー管理に関する詳細化（ゴール「バッテリーを管理することができる」に対する詳細化）のように，主要ゴールの詳細化に Milestone-driven パターンを用いることで，シナリオも考慮したゴール記述が Act タイプゴールの同定に対して適用され，これらがコンポーネントとして抽出されることとなる．この場合，Act タイプコンポーネントがシナリオにおける各状況の達成を，Analyze & Decide タイプコンポーネントが Act タイプコンポーネントの制御，つまりシナリオにおける各マイルストーンの達成に対する責務を担うこととなる．Act タイプゴール内における Milestone-driven パターンの適用については，コンフィギュレーション上での変化は与えないが，進化時においては，シナリオの変更がゴールモデル上の差分として検出され，これが変更すべきコンポーネントの同定を支援するため，この場合においても，シナリオを考慮したコンフィギュレーションであると考えることができる．

4.6　まとめ

本章では，進化を考慮したソフトウェアシステム開発を支援するゴールモデルコンパイラ gocc とプログラミングフレームワークについて述べた．gocc は，整形されたゴールモデルから，コンフィギュレーションやソフトウェア進化に必要な各種情報を生成するコンパイラである．

一方のプログラミングフレームワークは，Control loop を 3 種類のコンポーネントを用いて構築するための API を提供し，また，複数の Control loop の並行動作が可能である点が特徴である．加えて，Control loop を制御，デプロイ可能なコマンドセットを提供する点も特徴として挙げることができる．本章では，プログラミングフレームワーク上での Control loop 実装を支援するために導入した，Control loop 構成コンポーネントの実装のためのイディオムについても論じた．

これらのツールとプログラミングフレームワークを利用することで，ゴールモデルからの系統的なシステム実装が可能となる．また，コンポーネント連携による Control loop の構築が容易になることは，多種多様な進化を同時，あるいは時系列的に扱うことができることを意味しており，進化に対応可能なシステムの系統的な構築が期待できるといえる．

第 5 章

評価実験

5.1 はじめに

本研究では，要求の変化に伴う進化を考慮したソフトウェアシステム開発を実現するために，3 章において，ゴールモデルの整形プロセスに基づく Control loop 同定法とコンフィギュレーション生成法を導入した．また，コンフィギュレーション生成をはじめとするソフトウェア進化のための設計情報生成を目的として，4 章において，ゴールモデルコンパイラ gocc と提案する開発手法に基づいてシステムを実装，動作させるための 1 つの実装基盤を構築した．

本研究においては，これらの提案開発プロセスと支援ツールの適用可能性と有用性を検証するために，GUI ベースのモデリングツールと制御システムを対象としたソフトウェア進化実験を実施した．本章では，まず，本研究で実施した 5 種類の実験の概要について述べ，本実験の対象ソフトウェアシステムと各実験扱う進化内容について説明する．その後，各実験の内容と結果について示し，実験結果に基づいて提案手法の有効性を評価する．

5.2 実験の目的・概要

本実験の目的は，本研究により導入する Control loop モデリングに基づいて現実的なソフトウェアシステムが構築可能であることと，Control loop により構成されるソフトウェアシステムを効率的に進化させることができることを確認するところにある．

このような目的から，本実験では，大きく 5 種類の実験を実施した．まず，提案する開発プロセスに従って，Control loop モデリングに基づいてソフトウェアシステムの構築が可能であることを確認するために，本研究で導入する整形プロセスに従って既存ソフトウェアシステムに対するゴールモデルを整形し，得られたコンフィギュレーションから，Control loop を構成要素とするソフトウェアシステムを実際に構築し，その構築プロセスを評価した（実験 1）．続

いて，提案手法に従ったソフトウェアの進化が可能であることを確認するために，3 種類の進化要求に対して，実験 1 で構築したソフトウェアシステムに対する進化実験を実施した（実験 2）．実験 2 においては，すでに存在する対象ソフトウェアシステムの設計モデルと実装コードを利用し，従来のソフトウェアシステムに対しての進化と，提案手法により構築されたソフトウェアシステムに対しての進化とを比較することで，変更に伴う複雑さの観点からも提案手法の有効性を評価した．

本実験ではさらに，提案プロセスの一般性と，その有効性を評価するために，大学院博士課程前・後期の情報系学生を被験者としたソフトウェア開発実験を実施した．まず，整形プロセスにより Control loop が同定可能なゴールモデルを構築できることを確認するために，初期ゴールモデルに対する整形プロセス適用実験を実施した（実験 3）．続いて，提案手法の影響分析に対する有効性を評価するために，実験 2 で用いた 3 つの進化に対して，整形前後のゴールモデルとクラス設計図を用いた，進化の影響範囲分析実験を実施し，提案する開発プロセスのソフトウェアシステム進化時における有効性を評価した（実験 4）．

以上の実験 1〜実験 4 は，GUI ベースのモデリングツールを対象に実施したが，提案手法の適用可能性を評価することと，コード複雑化の抑制に対する本手法の有効性を評価することを目的として，清掃ロボットシミュレータ上でのロボット進化実験を実施した（実験 5）．

以降，まず，実験 1〜実験 4 において対象の GUI ベースモデリングツールとして用いた k-tool の概要と，本実験で扱う 3 つの進化について述べた後，k-tool 上での各実験結果を示す．その後，清掃ロボットシミュレータ上で実施した実験 5 の内容について示す．

5.3　実験対象モデリングツール：k-tool

本実験では，対象アプリケーションとして，ゴール指向要求分析法 KAOS のモデリングツールとして実用化されている k-tool を用いた．k-tool の画面イメージを図 5.1 に示す．k-tool では，KAOS におけるゴールモデルやオペレーションモデルなどの図形表現を図 5.1 で示したメインウィンドウ右部分に位置するメインエディタ上で描画可能であり，また，左部分には，モデルの構成要素として記述された各要素がツリー構造で表示されるインタフェースを持つ．また，図 5.1 には表示されていないが，各要素の属性情報を編集するプロパティビューも提供している．

k-tool は Java 言語により実装された GUI ベースのモデリングツールであり，その構成概要は表 5.1 に示すとおりである．k-tool は GUI をもつソフトウェアにおいて広く利用されている MVC（Model - View - Controller）モデル [49] に従って構築されている．MVC モデルにおいては，情報を管理する Model，Model の情報を GUI 上に表示する View，ユーザからの入力

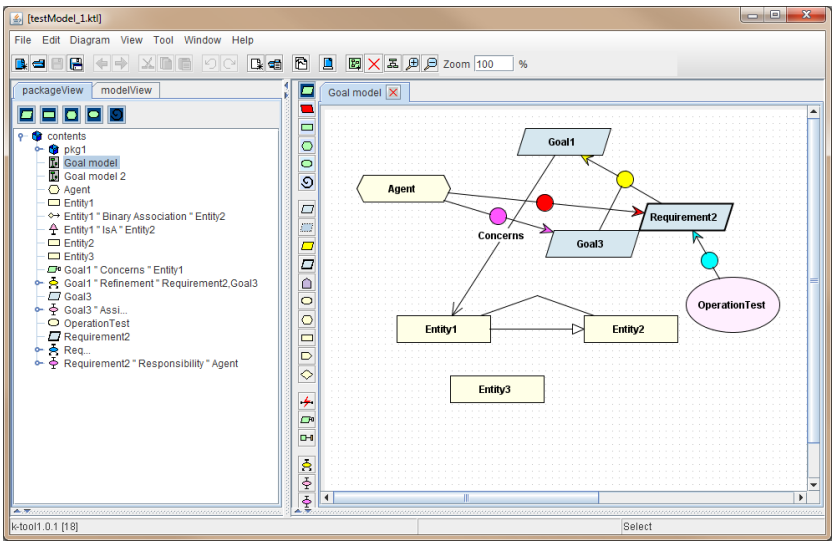

図 5.1. KAOS モデリングツール k-tool

表 5.1. k-tool の構成概要．ただし，クラス数には内部クラスは含んでいない．

メインパッケージ数	7 パッケージ (gui, io, tool, project など)
クラス数	271 クラス
総コード行数	64,803 行

を受け，View や Model を操作する Controller の 3 つの構成要素が存在し，これらの相互作用により，システムを設計・実装する．

k-tool を構成する各パッケージの役割を表 5.2 に，各パッケージ間の関係を図 5.3 に示す．k-tool においては，概して，gui パッケージが MVC モデルにおける View に，tool および print パッケージが Control に，system, project および io パッケージが Model に対応し，各パッケージを構成するクラス群が MVC モデルの該当する要素（Model，View，Controller）の各実装に該当する構造となっている．

本実験においては，まず，k-tool 構築時に作成された開発仕様書から k-tool に対する要求を抽出し，初期ゴールモデルを構築した．本実験において利用した初期ゴールモデルを図 5.2 に示す．この初期ゴールモデルにおいては，後述する進化 1，進化 2 に該当する機能と，実験の簡略化のために，プロパティ編集画面を別ウィンドウで表示する機能や，バージョン表示機能などの一部の機能を実際の k-tool から除外した状態の要求が記述されている．本論文では以降，この k-tool から一部の機能を除外したバージョンのソフトウェアを便宜上，**k’-tool** と呼び，k’-tool のゴールモデルに対して整形プロセスを適用し，Control loop 単位で構築したバー

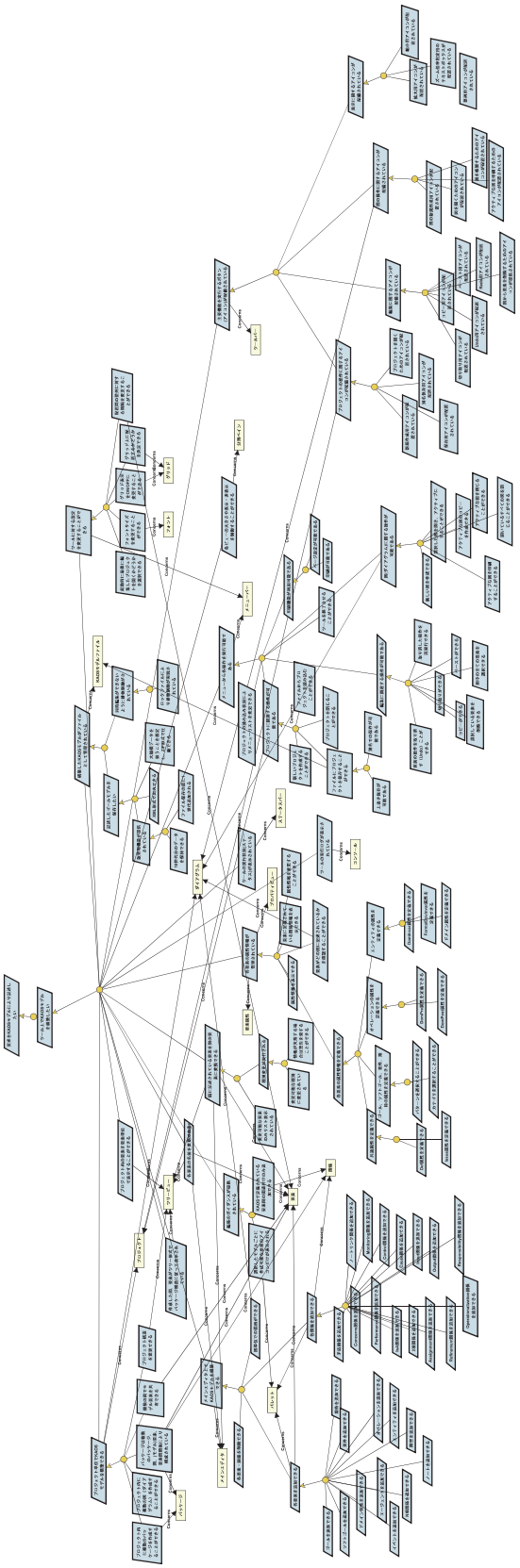

図 5.2. 実験で用いた k-tool に対する初期ゴールモデル

表 5.2. 各パッケージの役割

パッケージ名	位置づけ
gui	メインウィンドウ，メインエディタ部，メニュー，ツールバー部などの，GUI 部品および表示処理のためのクラスをまとめたもの．ゴールなど作図要素の描画クラスもここに含める．
io	各種ファイルへのアクセス（読み書き）クラスをまとめたもの．
print	印刷機能に関するクラスをまとめたもの．
project	本ツールでの編集対象である情報を保持するクラスをまとめたもの．
system	エディタの起動用メインクラス，各ダイアグラム，表示オブジェクトの管理など，ツール全体の管理機能のためのパッケージ
tool	編集操作に関する機能クラスをまとめたもの．
util	その他一般的な用途のためのクラスをまとめたもの．

ジョンのソフトウェアを **c-tool** と呼ぶ．

- **k-tool：** 従来の KAOS モデリングツール
- **k’-tool：** 従来の KAOS モデリングツールの実装から，進化 1，進化 2 に該当する機能と，一部の機能を除外（削除）したもの
- **c-tool：** k’-tool と同様の機能を持つが，本研究で導入する Control loop のモデリングに従って，設計・実装したもの

5.4 本実験で扱う進化

本実験では，k-tool に対する進化として，以下の 3 種類の機能追加を扱う．ここで進化 1，進化 2 により追加される機能については，従来の k-tool において提供されている機能である．本実験においては，進化に対する変更点を設計モデルや実装コード上で明確に把握するために，初期システムに含まれる機能ではなく，進化により追加される機能として扱うこととした．従って，本実験では，初期の k’-tool および c-tool を進化 1，進化 2 に該当する機能を削除した状態で構築するものとする．

進化 1（コンソールの追加）：KAOS モデルの編集過程を把握するためのログ出力機能をツールに付与する．画面右下に新たにログを出力するためのコンソールを追加し，ログはこのコンソールに追加するものとする．対象とする動作は，プロジェクトの開閉，モデル要素の追加削除，図の切り替えであり，これらの動作が実行されるとコンソールにその内容が出力される．コンソールに対しては，右クリックによりポップアップメニューを表示させ，コンソールの隠

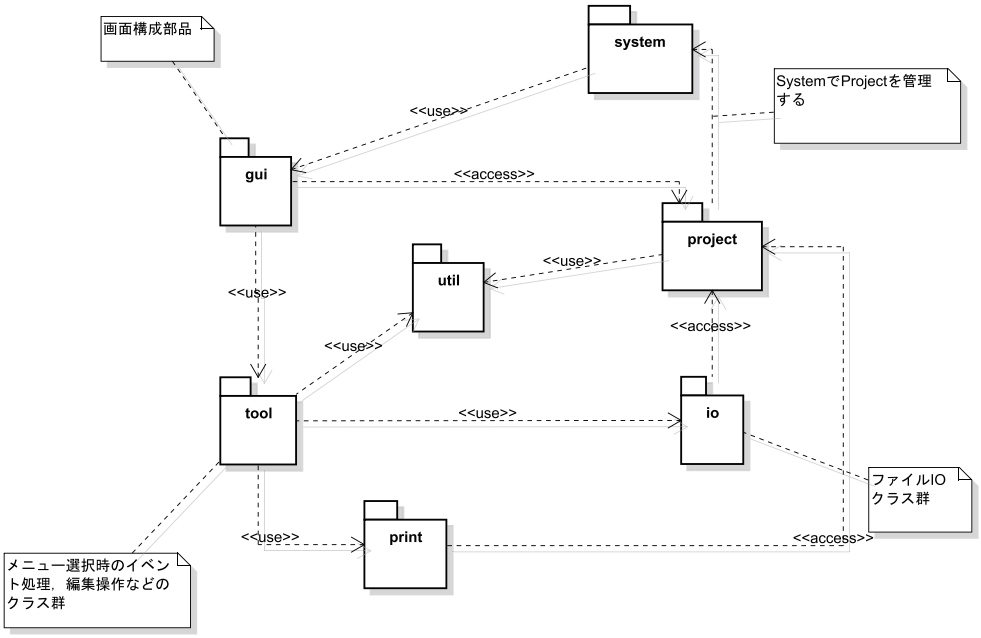

図 5.3. k-tool のパッケージ構成

蔽，表示ログの保存，表示ログの削除（クリア）機能が実行できる．

進化 2（プロパティビューの追加）：KAOS モデルの各要素に対して属性（プロパティ）を定義，編集できるように k’-tool および c-tool を進化させる．画面左側のツリービューの下側に，KAOS モデルの各要素に対して属性を定義，編集可能なプロパティビューを追加する．KAOS モデルの各要素によって持つべき属性は異なり，他の要素との関係情報などもタブの単位で分類してプロパティビューで表示する．

進化 3（Uses 関係定義タブの追加）：本研究で導入する分析手法がツール上で適用できるように，本研究で導入する利用関係（Uses）の概念が記述できるよう k’-tool および c-tool を拡張する．利用関係の定義には，メインエディタ上でゴールを選択し，プロパティビュー上の「Uses」タブを選択することで，利用するゴールの情報と，同ゴールに対する優先順位を設定できるように拡張する．

本実験では，これらの 3 つの進化に伴う機能追加を扱う．進化 1 は小規模な機能追加であり，進化 2 は大規模な機能追加に該当する．また，進化 3 は過去の進化結果に依存する進化に対応する．

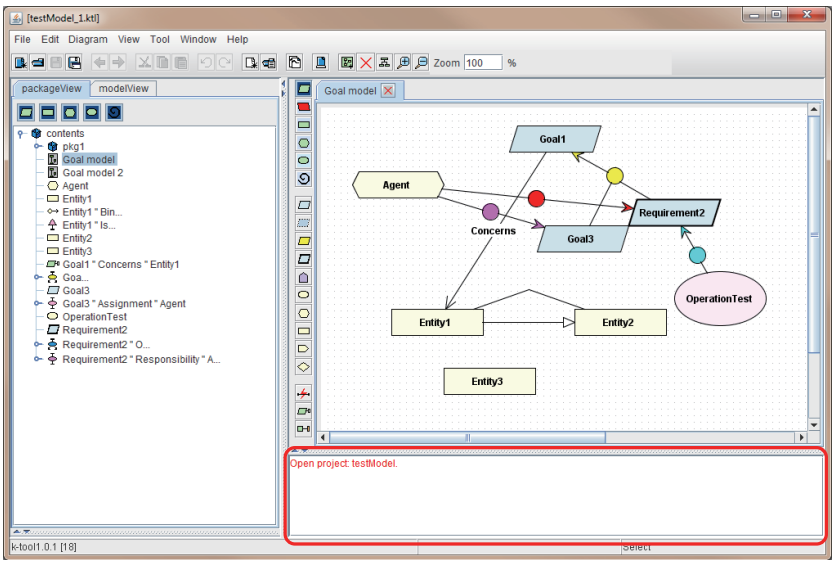

図 5.4. 進化 1：コンソール（画面右下部）の追加

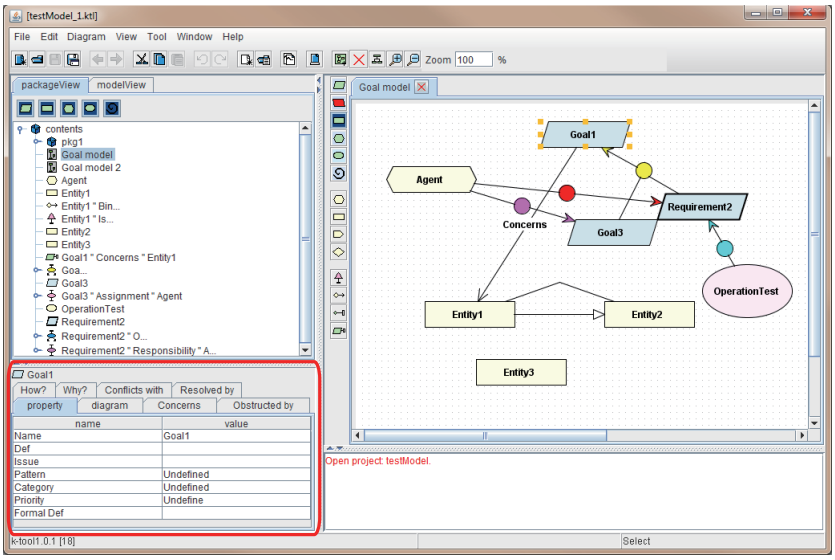

図 5.5. 進化 2：プロパティビュー（画面左下部）の追加

5.5 実験 1: c-tool 構築実験

5.5.1 実験概要

提案手法の有効性を検証するために，本研究ではまず，KAOS モデリングツールに対する要求モデルをもとに，本研究で導入する Control loop によるモデリングに基づいて，整形プロ

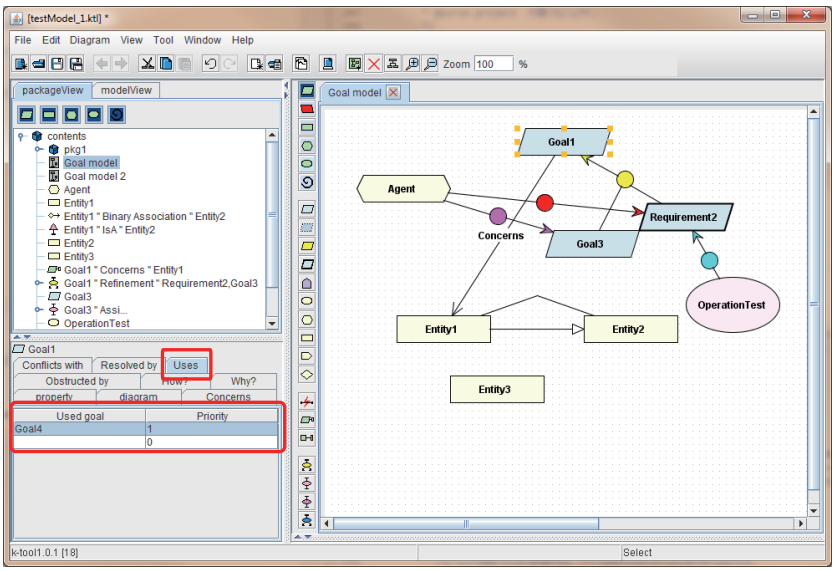

図 5.6. 進化 3：Uses 関係定義タブ（プロパティビュー内）の追加

セスやコンフィギュレーション生成を通じて，実際に c-tool（ 5.3 節で述べたとおり，Control loop モデリングにより設計，実装された機能限定版の k-tool）を構築した．

本実験では，構築すべき KAOS モデリングツールの初期要求として，図 5.2 のゴールモデルを用い，整形プロセス適用後に gocc を利用してコンフィギュレーションなどの設計支援情報を生成後，生成された各情報に基づいて c-tool を実装した．なお，実験 1,2 を通して用いたソフトウェアシステム実装環境は次のとおりである．

- OS: Windows 7 64-bit Version
- CPU: Quad-core Intel Xeon 2.66GHz
- Memory: 12.0GB
- Java: Sun JRE (JDK) 1.7.0_01
- Eclipse: Version 3.7.1 (INDIGO)
- JADE: Version 3.7

5.5.2　実験結果

本実験においては，まず，図 5.2 のゴールモデルに対して，本研究で導入する整形プロセスを適用した．整形プロセスを適用後のゴールモデルを図 5.7 に示す．

整形プロセス適用後のゴールモデルにおいては，メインエディタなどの各ビューを構成する Control loop や，それらが共通で利用する Control loop など，合計 10 個の Control loop が同定された．整形前後のゴールモデルの構成要素の比較を表 5.3 に示す．表 5.3 からは，まず，

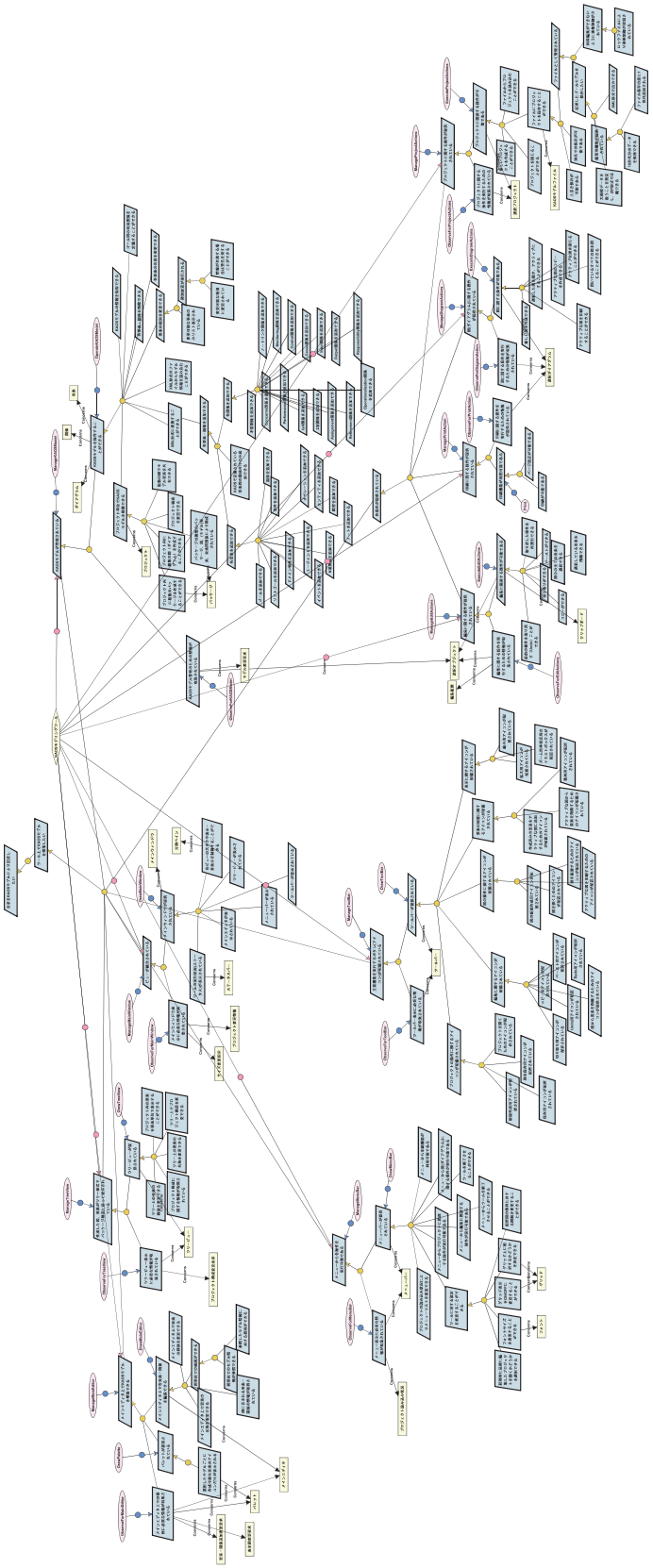

図 5.7. k-tool に対する整形後のゴールモデル

表 5.3. 整形前後におけるゴールモデルの構成変化

	整形前	整形後
ゴール数（# Goals）	120	168
エンティティ数（# Entities）	15	28
Concerns 関係数（# Concerns）	39	40
# Concerns / # Entities	2.6	1.43
Control loop 数（# Control loops）	–	10

整形後にゴール数とエンティティ数が増加していることが分かるが，これは，ゴールに関しては，共通ゴール記述の追加や Collect タイプゴールの追加によるものであり，エンティティに関しては，プロセス変数の新たな追加によるものである．その一方で，表 5.3 からはエンティティあたりの Concerns 関係数 (#Concerns/ #Entities) は減少していることも確認できる．これは，Control loop 単位でプロセス変数，つまりエンティティへのアクセスを限定することにより，各 Control loop におけるエンティティに対する責務が明確化されたことを示しているといえる．実際に，整形前のゴールモデル上で Concerns 関連が多く定義されていたエンティティである「ダイアグラム」（5 本）や「要素」（7 本）については，整形後は，主要ゴール「KAOS モデルが管理されている」内のサブゴールからの 1 本の Concerns 関連に集約されていることを確認することができた．

続いて，図 5.7 の整形後ゴールモデルを gocc へ入力として与えた．gocc から得られた情報として，Control loop の階層構造情報と階層構造を可視化したものを図 5.8 に示す．図 5.8 に表示されているコンポーネントは各 Control loop を表現しており，合計 10 個の Control loop と，Control loop 間の階層構造を確認することができた．各 Control loop の責務は表 5.4 に示すとおりである．

gocc が出力した競合リストにおいては，同一の Control loop 利用に関する競合の可能性も出力されたが，エンティティに関する競合は検出されないことを確認することができた．同一の Control loop 利用に関する競合についても，k-tool（ここでは c-tool）はユーザからの入力によるツールであり，出力された可能性のある状況に対しても，複数の Control loop が同時に同一 Control loop を利用するものはないと判断したため，本実験においては，競合に対応するための設計は施さないこととした．

続いて，gocc から得られた情報をもとに c-tool を実装した．本実験においては，以下の実装方針により，本研究で提案するプログラミングフレームワーク上に Control loop により構成される c-tool を実装した．まず，各 Control loop においては，コンフィギュレーション上では

```
ManageMainEditor Uses ManageKAOSModel
ManageMainEditor Uses ManageDiagramActions
ManageMainEditor Uses ManageEditActions
ManageMainEditor Uses ManagePrintAction
ManageTreeView Uses ManageKAOSModel
ManageTreeView Uses ManageProjectActions
ManageTreeView Uses ManageDiagramActions
ManageTreeView Uses ManageEditActions
ManageTreeView Uses ManagePrintAction
ManageMenuBar Uses ManageProjectActions
ManageMenuBar Uses ManageDiagramActions
ManageMenuBar Uses ManageEditActions
ManageMenuBar Uses ManagePrintAction
ManageToolBar Uses ManageProjectActions
ManageToolBar Uses ManageDiagramActions
ManageToolBar Uses ManageEditActions
ManageToolBar Uses ManagePrintAction
ManageMainWindow Uses ManageMainEditor
ManageMainWindow Uses ManageTreeView
ManageMainWindow Uses ManageMenuBar
ManageMainWindow Uses ManageToolBar
ManageProjectActions Uses ManageKAOSModel
ManageDiagramActions Uses ManageKAOSModel
ManageEditActions Uses ManageKAOSModel
ManagePrintAction Uses ManageKAOSModel
```

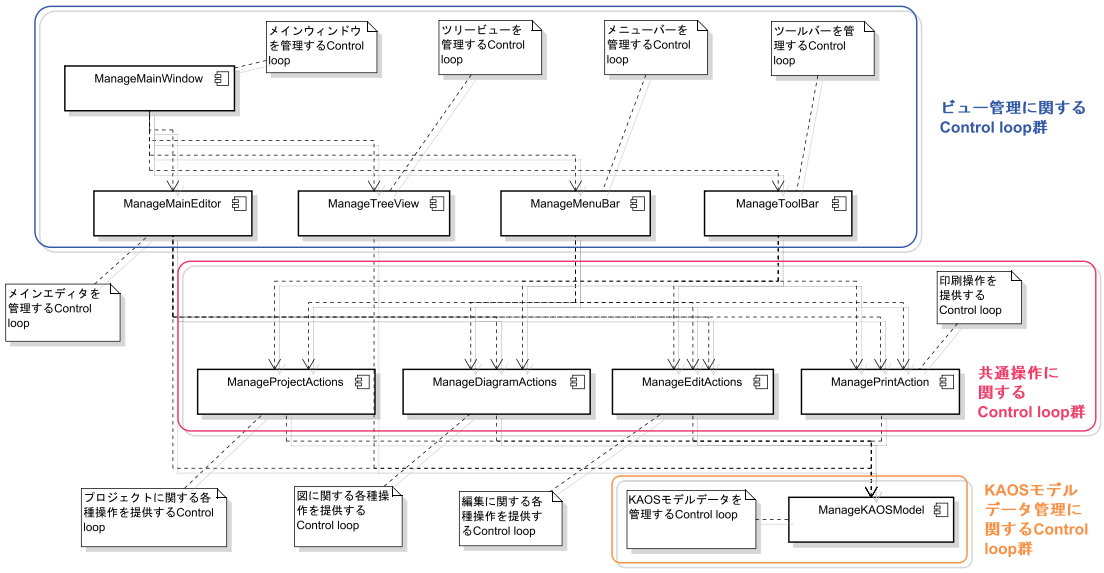

図 5.8. c-tool の Control loop 間階層構造（上段： gocc により生成される階層構造情報，下段：コンポーネント図による可視化表記）

表 5.4. 同定された各 Control loop の責務．Control loop 名には Analyze & Decide タイプのゴールに割り当てられたオペレーション名を用いている．

Control loop 名	責務
ManageMainWindow	メインウィンドウを管理する．分割ペインに対する入力に基づいて，各部品のサイズを変更したり，非表示にする．ステータスバーの表示についても責務を持つ．
ManageMainEditor	メインエディタ部を管理する責務を持つ．
ManageTreeView	ツリービュー部を管理する責務を持つ．
ManageMenuBar	メニューバー部を管理する責務を持つ．
ManageToolBar	ツールバー部を管理する責務を持つ．
ManageProjectActions	ファイルへの保存など，プロジェクトに関連する操作を提供する責務を持つ．
ManageDiagramActions	ダイアグラム（図）に関連する操作を提供する責務を持つ．
ManageEditActions	コピー，ペーストなど，編集に関連する操作を提供する責務を持つ．
ManagePrintAction	印刷操作を提供する責務を持つ．
ManageKAOSModel	KAOS モデルデータを管理する責務を持つ．

Collect，Analyze & Decide，Act タイプの 3 種類のコンポーネントが同定されるが，本実験のような GUI アプリケーションの場合は，マウスからの入力と処理結果の表示が同一の GUI 部品である場合が多いことと，同様の理由から，情報収集部である Collect タイプと，出力処理部である Act タイプの進化のタイミングが異なる可能性が少ないことから，本実験においては，Control loop の構築にあたっては，3 つのタイプのコンポーネントを明示的に分離して構築するのではなく，Analyze & Decide タイプのコンポーネントを必須のコンポーネントとして，特に必要な場合を除いて，Collect，Act タイプの責務も同コンポーネント上に実装することとした．また，情報収集の手段としては，マウスイベントの他に，モデル要素の状態を観察するためにデザインパターンの一つである Observer パターン [50] を実装し，各 Analyze & Decide タイプのコンポーネントは情報収集対象の Observer インタフェースを実装することで，対象の変化が観測できるように実装した．また，他 Control loop の利用に関しては，利用 Control loop の Analyze & Decide タイプのコンポーネント上にサービス提供用の public メソッドを用意し，同メソッドを呼び出すスタイルで実装した．これは，今回実験対象とした k-tool のような，ユーザからの入力に応じて動作する GUI アプリケーションにおいては，Control loop 間での競合が発生することは少なく，従って，Control loop の優先順位を制御する機会が少ないと判断したことによるものである．

このような実装方針に基づいて，c-tool を実装した．具体的には，図 5.8 に示される各

Control loop 内に，同名のクラスを Control loop を制御するクラスとしてそれぞれ実装し，これらのクラスが必要に応じて k-tool の構成部品クラスを呼び出すことで，各 Control loop を構築した．

各 Control loop を実装後，それらを 4 章で導入したプログラミングフレームワーク上で動作させることにより，Control loop により構成される c-tool が問題なく動作し，要求された機能を提供していることが確認できた．

5.6 実験 2：進化実験

5.6.1 実験概要

続いて，実験 1 で構築した c-tool に対して，5.4 節で定義した 3 つの進化を順次適用した．この実験 2 においては，すでに存在する k-tool の設計モデルと実装コードを利用することで，従来のソフトウェアシステムに対しての進化と，提案手法に基づいて構築されたソフトウェアシステムに対しての進化とを比較した．具体的には，5.3 節で述べたように，k-tool に対して機能を限定した k’-tool と，実験 1 で構築した c-tool とをそれぞれ進化 1〜進化 3 の順で進化させ，開発プロセスやゴールモデル，実装コードを比較することで，その違いを分析，評価した．

k’-tool に対する進化については，従来の開発プロセスを想定し，ゴールモデル上に進化に対する要求を追加した後，設計モデル上で変更箇所を同定した上で，実装モデル上で同定した変更を反映させた．ただし，進化 1，進化 2 については，k-tool においては既に実装されている機能であるため，k’-tool と k-tool の構成を比較することにより，変更箇所を評価した．

一方，c-tool に対しての，各進化における作業手順は次のとおりである．

1. **ゴールの追加，再整形：**まず，機能追加などの進化に関する要求をゴールモデル上に記述する．新たな要求を記述後には，これらの要求に対しても，整形プロセスを適用する．
2. **gocc 生成情報を用いた設計：**再整形したゴールモデルを gocc に再度入力情報として与え，進化に対する設計支援情報を獲得する．具体的には，階層構造情報を含む Control loop のコンフィギュレーションや，競合リスト，差分情報を獲得する．これらの情報を利用して，進化に伴い変更すべき箇所と変更内容を決定する．
3. **実装：**同定された変更箇所を実際に実装する．Control loop を新たに追加する場合は，Control loop を管理するクラスと Control loop が提供するサービスを実現するための構成部品クラスを追加する．既に存在する Control loop を変更する場合は，構成部品クラスを追加，あるいは変更する．実装後には，プログラミングフレームワーク上で実装コードを動作させ，実際に進化の内容が機能として提供されていることを確認する．

5.6.2　実験結果

実験結果として，まず，進化 1〜進化 3 に対する従来手法におけるゴールモデル上での変更箇所を図 5.9 に，整形プロセスに基づいて整形されたゴールモデル上での変更箇所を図 5.10 に示す．また，これらの 2 つのゴールモデルにおける各進化後の構成要素の変更状況を表 5.5 に示す．

図 5.9 と図 5.10 を比較すると，まず，整形後ゴールモデルにおいては，各進化に対する変更箇所が分散されることが分かる．これは，進化に対する要求は同一であるものの，整形プロセスを適用することにより，共通化されているゴールにも追加された要求が分散して配置されることによるものである．また，表 5.5 は，整形プロセスを適用することで，各進化において記述すべきゴールやエンティティが増加する可能性があることも示唆している．ゴール数の増加に関しては，進化に対して追加したゴールと共通化，つまり一般化されている主要ゴール内へのゴールの追加により，記述が分散する場合があることと，Control loop を形成するために Collect タイプのゴールなどを新たに追加する必要があることによるものである．エンティティ数の増加に関しても同様に，整形プロセスを適用することで Control loop に対するプロセス変数を新たに定義する必要があることによるものである．このように，提案する開発プロセスで適用するゴール整形プロセスは，従来の要求記述に加えて，記述量や記述コストが若干高まると言える．ただし，要求が構造化されていることから，分散して追加するゴールを記述すべき場所が限定されているという点，プロセス変数の同定など，追加内容や追加手順が提示されている点から，記述が極端に難しくなることはないと言える．

なお，表 5.5 からは，進化 1 と進化 2 において，システムを構成する Control loop が 1 つずつ追加されていることも確認できた．

続いて，各進化に対するコード上の変更箇所を計測した．計測結果を表 5.6〜表 5.8 に，システム構成図上での各進化における影響を図 5.11，図 5.12 に示す．まず，進化 1 のコンソール追加に関しては，表 5.6 に示す通り，従来の k’-tool においては MVC モデルのすべての要素，具体的には gui，tool，system の各パッケージにおいてクラスの修正が必要となった（修正：6 クラス）．一方の c-tool では，新たに追加するコンソール部を管理する Control loop の管理クラス（ManageConsole）を追加し，この追加した Control loop を宣言，配置するためにメインウィンドウ部の構成要素である MainFrame クラスを修正することで，進化 1 を実現することができた（新規作成：1 クラス，修正：1 クラス）．

進化 2（表 5.7）に対しても同様に，k’-tool では gui，tool，project，io の各パッケージにおいてクラスの修正が必要となり，MVC モデルのすべての要素にわたる変更が必要であった（新

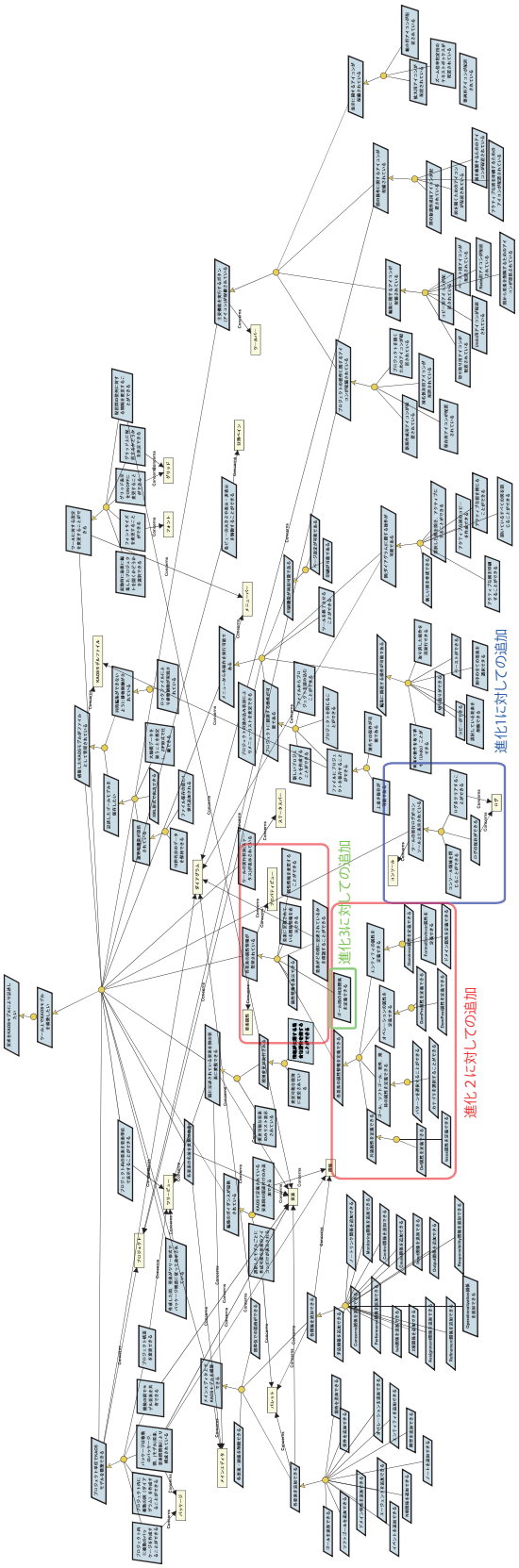

図 5.9. 進化 1〜進化 3 に対する従来のゴールモデル上での変更箇所

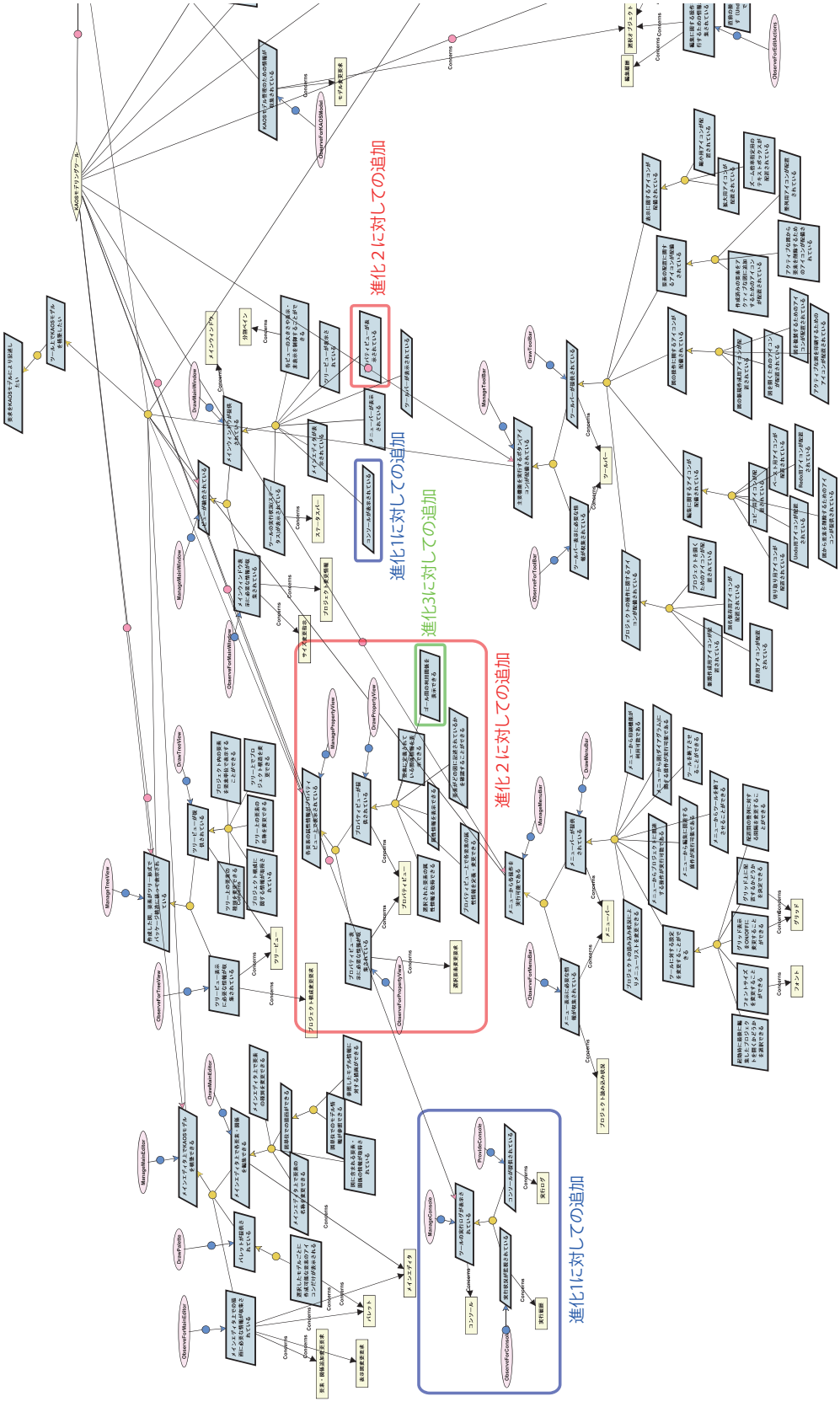

図 5.10. 進化 1〜進化 3 適用後のゴールモデル（左側）

表 5.5. 整形前後におけるゴールモデルの構成変化（括弧内の数値は進化による増減値）

■従来のゴールモデル（k'-tool 構築時に利用）

	進化前	進化 1 後		進化 2 後		進化 3 後	
# Goals	120	124	(+4)	143	(+19)	144	(+1)
# Entities	15	17	(+2)	19	(+2)	19	(±0)
# Concerns	39	42	(+3)	44	(+2)	44	(±0)
Concerns/Entity	2.6	2.47	(−0.13)	2.32	(−0.15)	2.32	(±0)
# Control loops	–	–		–		–	

■整形後ゴールモデル（c-tool 構築時に利用）

	進化前	進化 1 後		進化 2 後		進化 3 後	
# Goals	168	172	(+4)	198	(+26)	200	(+2)
# Entities	28	31	(+3)	34	(+3)	34	(±0)
# Concerns	40	43	(+3)	47	(+4)	47	(±0)
Concerns/Entity	1.43	1.39	(−0.04)	1.38	(−0.01)	1.38	(±0)
# Control loops	10	11	(+1)	12	(+1)	12	(±0)

表 5.6. 進化 1（コンソール画面の追加）に関する実装レベルでの変更内容

■ k'-tool

パッケージ	変更クラス	変更種別	変更内容	LOC
gui (View)	MainFrame	クラス修正	Console の配置，Console を隠す動作の追加，Observer の動作追加	142
	Resource	クラス修正	コンソール関係の定数宣言	1
tool (Control)	ViewActionTool	クラス修正	ログの保存，クリア機能を定義	82
	ToolResource	クラス修正	コンソール関係の定数宣言	5
system (Model)	SystemManager	クラス修正	保存機能の実装，メッセージ追加メソッドの提供	31
	Resource	クラス修正	コンソール関係の定数宣言	1
合計 (新規作成：0 クラス，クラス修正：6 クラス)				262

■ c-tool

Control loop	変更クラス	変更種別	変更内容	LOC
コンソール管理部	ManageConsole	新規作成	Console の定義，動作（保存，クリア，メッセージ追加機能）の実装，Observer の動作追加	261
メインウィンドウ管理部	MainFrame	クラス修正	Console の配置，Console を隠す動作の追加	16
合計 (新規作成：1 クラス，クラス修正：1 クラス)				277

表 5.7. 進化 2（プロパティビューの追加）に関する実装レベルでの変更内容

■ **k'-tool**

パッケージ	変更クラス	変更種別	変更内容	LOC
gui (View)	PropertyPanel	新規作成	プロパティ部のパネル	265
	PropertyPane	新規作成	プロパティ表示処理及びビュー上での操作イベントハンドラとして働く	2710
	MainFrame	クラス修正	プロパティビュー関連部定義, 各種メソッドの追加	59
	Resource	クラス修正	プロパティ関係の定数宣言	14
	EditAttributesTableModel	新規作成	プロパティ部の属性編集テーブルのモデル	80
	PropertyColumnEditor	新規作成	プロパティ編集テーブル用のエディタ	565
	PropertyRelationTableModel	新規作成	プロパティ部の関連項目一覧テーブルのモデル	60
	PropertyTableModel	新規作成	プロパティ部のプロパティ編集テーブルのモデル	63
tool (Control)	PopupActionTool	クラス修正	プロパティ関連アクションの宣言, プロパティ部の Popup 用削除アクション	24
	ToolResource	クラス修正	プロパティ関連の定数宣言	2
project (Model)	要素関連クラス（Agent, Operation, Objective など 8 クラス）	クラス修正	プロパティ変数の追加	1285
	関係線関連クラス（BinaryAssociation, Concern, Control, Input など 11 クラス）	クラス修正	プロパティ変数の追加	1123
	ElementAttribute	新規作成	属性情報を保持するクラス	126
	Role	新規作成	関係のロールクラス	105
	ElementTypeChangeSupport	クラス修正	型変更時のプロパティ変数の扱い	85
io (Model)	ProjectXMLWriter	クラス修正	プロパティ関連部の XML 記述生成部を追加するよう拡張	868
	ProjectHandler1_0	クラス修正	Uses 関係情報を XML ファイルから読み込めるよう拡張	49
合計 (新規作成：8 クラス，クラス修正：26 クラス)				7873

■ **c-tool**

Control loop	変更クラス	変更種別	変更内容	LOC
プロパティビュー管理部	ManagePropertyView	新規作成	プロパティビューの定義，動作定義, Observer の定義	282
	PropertyPane	新規作成	プロパティ表示処理及びビュー上での操作イベントハンドラとして働く	2766
	EditAttributesTableModel	新規作成	プロパティ部の属性編集テーブルのモデル	80
	PropertyColumnEditor	新規作成	プロパティ編集テーブル用のエディタ	565
	PropertyRelationTableModel	新規作成	プロパティ部の関連項目一覧テーブルのモデル	60
	PropertyTableModel	新規作成	プロパティ部のプロパティ編集テーブルのモデル	63
メインウィンドウ管理部	MainFrame	クラス修正	プロパティビューの配置，フォント変更, オブザーバの追加等	17
KAOS モデル管理部	要素関連クラス（Agent, Operation, Objective など 8 クラス）	クラス修正	プロパティ変数の追加	1285
	関係線関連クラス（BinaryAssociation, Concern, Control, Input など 11 クラス）	クラス修正	プロパティ変数の追加	1123
	ElementAttribute	新規作成	属性情報を保持するクラス	126
	Role	新規作成	関係のロールクラス	105
	ElementTypeChangeSupport	クラス修正	型変更時のプロパティ変数の扱い	85
	ProjectXMLWriter	クラス修正	プロパティ関連部の XML 記述生成部を追加するよう拡張	868
	ProjectHandler1_0	クラス修正	Uses 関係情報を XML ファイルから読み込めるよう拡張	439
合計 (新規作成：8 クラス，クラス修正：23 クラス)				7864

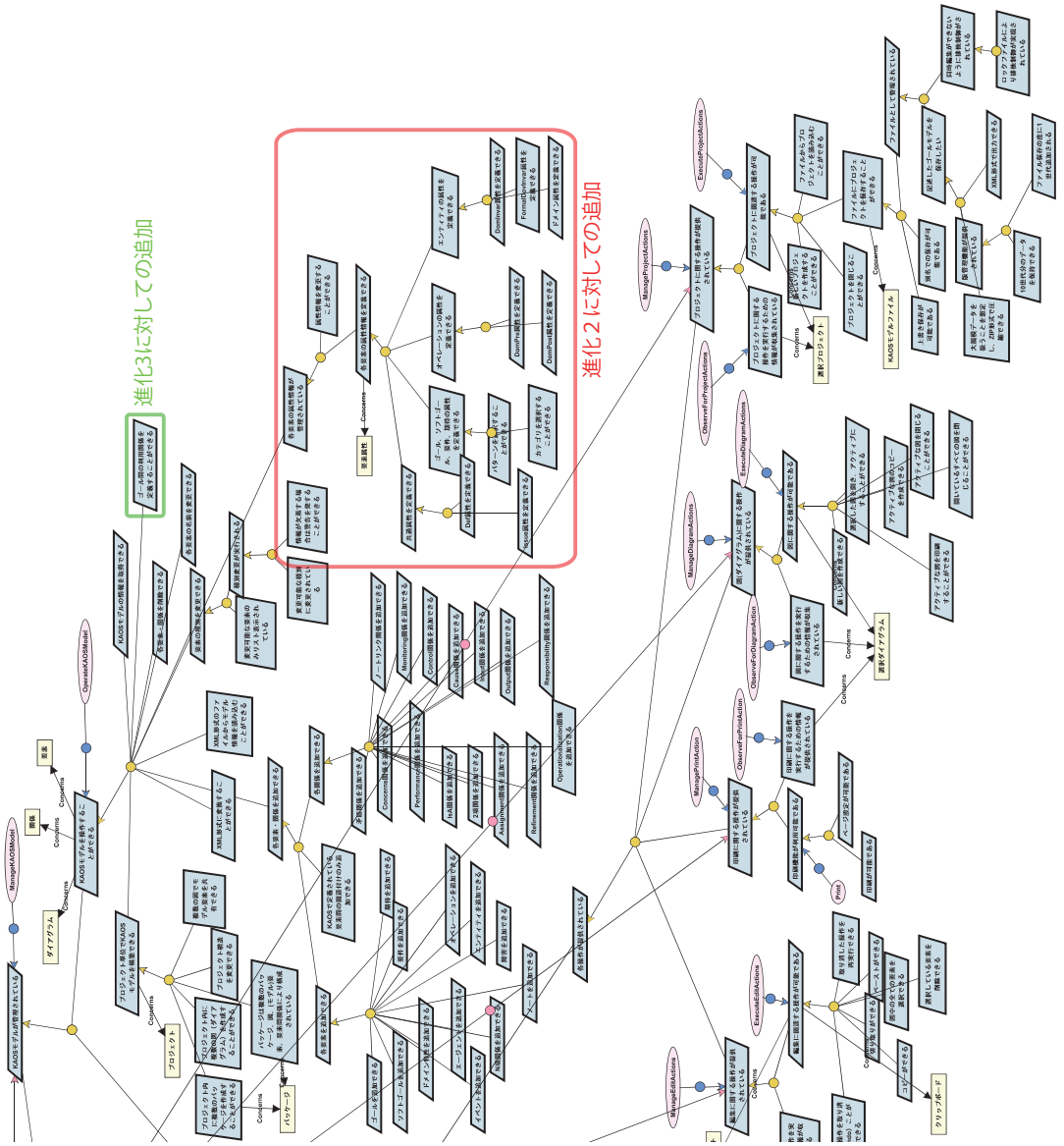

図 5.10. 進化 1〜進化 3 適用後のゴールモデル（右側）

規作成：8 クラス，修正：26 クラス）．一方で，c-tool では，新たに追加するプロパティビュー管理に関する Control loop に該当するクラス群の新規追加と，追加した Control loop を宣言，配置するためのメインウィンドウ部のクラス修正，KAOS モデルを管理する Control loop 部の修正が必要であった（新規作成：8 クラス，修正：22 クラス）．

進化 3 については，進化 1，進化 2 と異なり新たな Control loop の追加を伴わない進化であったが，この場合も k’-tool に関しては表 5.8 に示すように，gui，tool，project，io の各パッケージに対する変更が必要であり，MVC モデルのすべての要素に変更の影響が及んだ

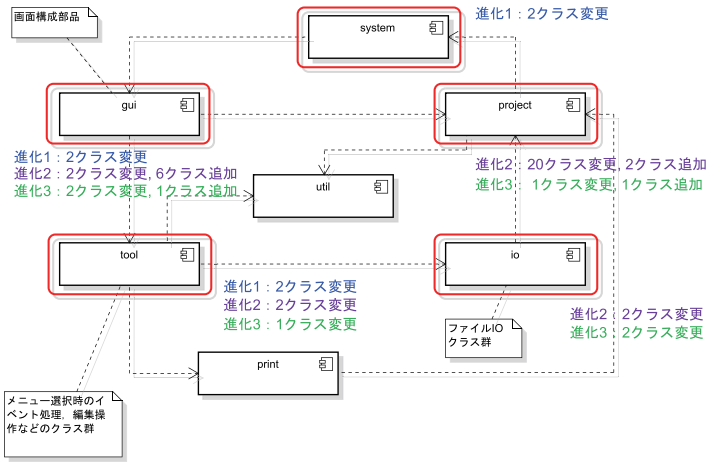

図 5.11. 進化 1〜3 に対する k’-tool における変更箇所

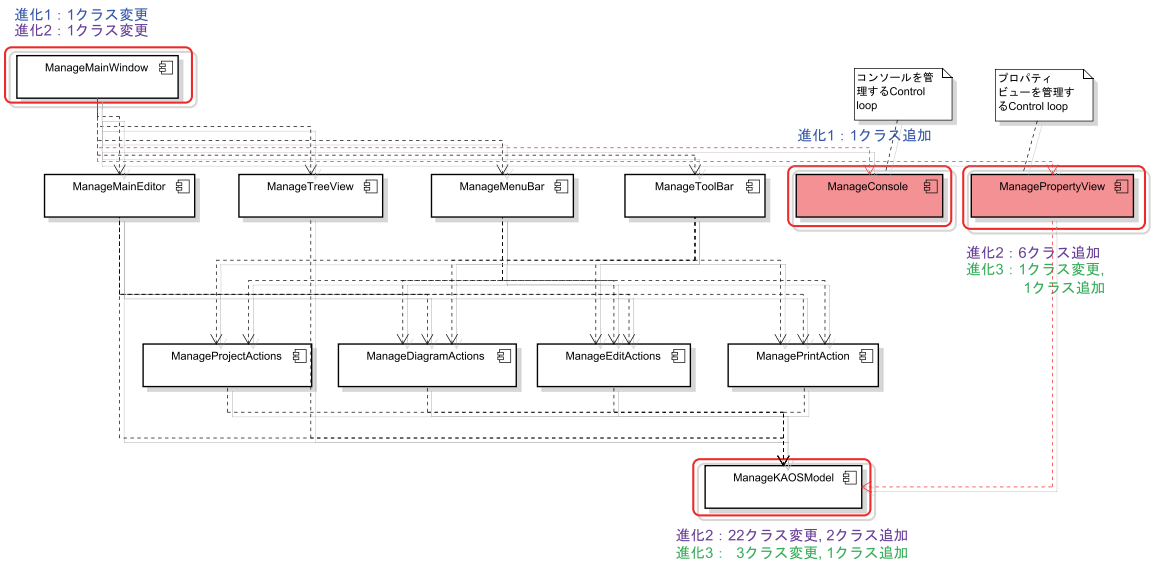

図 5.12. 進化 1〜3 に対する c-tool における変更箇所

が，c-tool においては，プロパティビューに関する Control loop と KAOS モデル管理に関する Control loop に対する変更に閉じていた．

進化 1〜進化 3 の実装コード上での変更箇所を比較すると，まず，k’-tool においては，各進化において MVC モデルのすべての要素（Model，View，Controller）に変更影響が及ぶことが分かる．MVC モデルは，新たなビューの追加などに対しては，View を追加するだけでよいという利点があるが，Model の変更を伴うような進化に対しては，変更影響がすべての要素に及ぶ可能性が高い．今回の実験においても，各進化に対して，MVC モデルのすべての要素にお

表 5.8. 進化 3（Uses 関係定義タブの追加）に関する実装レベルでの変更内容

■ k’-tool

パッケージ	変更クラス	変更種別	変更内容	LOC
gui (View)	PropertyPane	クラス修正	Uses 定義用タブの追加	141
	EditUsesTableModel	新規作成	Uses の情報を編集するテーブルのモデルを定義	74
	Resource	クラス修正	定数宣言の追加	5
tool (Control)	PopupActionTool	クラス修正	Uses 編集テーブルでの削除アクションを定義	36
project (Model)	UseGoal	新規作成	Uses 情報を管理するクラスを定義	121
	Objective	クラス修正	UseGoal クラスをメンバ変数に持つよう拡張	67
io (Model)	ProjectXMLWriter	クラス修正	Uses 関係情報を XML ファイルに保存するよう拡張	42
	ProjectHandler1_0	クラス修正	Uses 関係情報を XML ファイルから読み込めるよう拡張	49
合計 (新規作成：2 クラス，クラス修正：6 クラス)				535

■ c-tool

Control loop	変更クラス	変更種別	変更内容	LOC
プロパティビュー管理部	PropertyPane	クラス修正	Uses 定義用タブの追加	178
	EditUsesTableModel	新規作成	Uses の情報を編集するテーブルのモデルを定義	74
KAOS モデル管理部	UseGoal	新規作成	Uses 情報を管理するクラスを定義	121
	Objective	クラス修正	UseGoal クラスをメンバ変数に持つよう拡張	67
	ProjectXMLWriter	クラス修正	Uses 関係情報を XML ファイルに保存するよう拡張	42
	ProjectHandler1_0	クラス修正	Uses 関係情報を XML ファイルから読み込めるよう拡張	49
合計 (新規作成：2 クラス，クラス修正：4 クラス)				531

表 5.9. 3 つの進化におけるソースコード上の変更箇所．“#Packages” は，追加，修正されたパッケージ数・Control loop 数を，“#Classes” は，追加，修正されたクラス数を示す．

		#Packages		#Classes		
		修正	追加	修正	追加	ΔLOC
初期開発時	k’-tool	–	7	–	263	56,668
	c-tool	–	10	–	278	57,015
進化 1 後	k’-tool	3	0	6	0	262
	c-tool	1	1	1	1	277
進化 2 後	k’-tool	4	0	26	8	7,873
	c-tool	2	1	23	8	7,864
進化 3 後	k’-tool	4	0	6	2	535
	c-tool	2	0	4	2	531

いて，変更すべき箇所や変更により影響が及ぶ可能性のある箇所を特定する必要があった．

一方で，c-tool においては，まず，実装上での追加，変更箇所に該当する Control loop と，図 5.10 に示したゴールモデルの変更箇所を包含する Control loop とが一致していることが確認できた．このゴールモデル上の変更箇所と実装上の変更箇所の一致は，提案する開発プロセスが追跡可能性を有していることを示したものである．従来の開発法に従った k’-tool においては，図 5.9 に示したゴールモデルとアーキテクチャモデルである MVC モデルとを関連付けることは難しく，ゴールモデルの変化から実装モデル上での変更個所を特定することは困難である．このように，提案する開発プロセスでは，整形後ゴールモデルからシステムのアーキテクチャを決定するため，ゴールモデル整形のコストは必要であるものの，要求モデルと設計・実装モデルとを関連付けることが可能となり，結果として，要求モデル上の変化から実装モデル上の変更すべき箇所を特定することが可能となる．

進化 1～進化 3 のソースコード上での変更量を表 5.9 に示す．表 5.9 からは，提案手法に従って実装した c-tool においては，複数の Control loop を個別に実装するため，初期開発時には実装すべきクラス数およびコード行数（LOC）が若干大きくなるものの，進化に対応するために追加・変更すべきコード行数が，k’-tool における進化と比較してほぼ同程度であることが分かる．また，修正が必要なクラス数とパッケージ数（Control loop 数）が k’-tool の進化よりも少ないことが分かる．修正が必要なクラスが抑制された一つの理由は，MVC モデルに従った k’-tool では，進化による機能追加に対して，Model，View，Controller それぞれの要素内で広く共通利用するクラスに変更影響が及んだことによるものである．このような共通クラスの変更は，他の機能に対しての変更影響も考慮した修正が必要となるため，進化の実現を困難にする可能性がある．修正クラスが抑制されたもう一つの理由は，クラスの修正ではなく，クラスの追加により機能追加を実現することが出来たことによるものである．クラスの修正は，クラス内に加えた変更の影響についても考慮する必要があるため，同一の機能追加を実装する場合には，独立したクラス追加よりも変更コストが大きいと考えられる．c-tool においては，修正すべきパッケージ数（Control loop 数）が k’-tool よりも削減されているが，これはコード上の影響範囲を限定していることも意味している．以上より，表 5.9 の実験結果は，c-tool の方が進化要求に対するコード上への変更実装コストが小さいことを示している．

表 5.10. 各被験者によるゴールモデルの整形結果

	整形前	整形後 (実験 1 結果)	被験者 1 整形後	被験者 2 整形後	被験者 3 整形後	被験者 4 整形後
ゴール数（# Goals）	120	168	163	159	131	148
エンティティ数（# Entities）	15	28	36	39	41	23
Concerns 関係数（# Concerns）	39	40	41	99	57	29
# Concerns / # Entities	2.6	1.43	1.14	2.54	1.39	1.26
Control loop 数（# Control loops）	–	10	8	12	9	10

5.7 実験 3：ゴールモデル整形実験

5.7.1 実験概要

続いて，提案する開発プロセスの利用性と有用性を検証するために，情報系の大学院前・後期学生 4 名を被験者として，c-tool の構築と進化に関するシステム設計実験を実施した．

まず，実験 3 においては，整形プロセスの利用性を評価するために，4 名の被験者それぞれに図 5.2 の整形前ゴールモデルを与え，整形プロセスに従って整形してもらい，これを実験 1 で整形した図 5.7 のゴールモデルと比較することにより，各被験者の整形後ゴールモデルを評価した．

5.7.2 実験結果

各被験者により整形されたゴールモデルの各構成要素数を，整形前のゴールモデル，実験 1 での整形結果と併せて表 5.10 に示す．表 5.10 からは，それぞれ差異があるものの，実験 1 の整形結果に近い数の Control loop が抽出されることが確認できる．また，ゴール数，エンティティ数がともに整形前より増加する一方で，被験者 2 を除いて，エンティティあたりの Concerns 関係数 (#Concerns/ #Entities) が大きく減少していることが確認できる．

整形後の各ゴールモデルを確認したところ，まず，被験者 1 の整形結果では，GUI の各部品を管理する Control loop が 6 つ，KAOS モデルを管理する Control loop が 2 つ抽出され，前者の Control loop 群が関連する後者の Control loop 群を利用する（`Uses` 関係により参照する）という構成であった．被験者 1 の整形結果については，各 Control loop の構築結果は妥当なものであったと考えるが，操作に関するゴール群がモデル管理の Control loop に集約されているという点が，実験 1 の整形結果との大きな違いであった．被験者 1 の整形結果から得られる Control loop 群は，システムを構成するものとしては過不足のないものであるが，操作の追加

に関する進化要求が何度もあった場合には，モデル管理の Control loop が肥大化する可能性があると考えられる．

次に，被験者 2 の整形結果からは，GUI の各部品を管理する Control loop が 5 つ，操作，共通操作に関する Control loop が 6 つ，KAOS モデルを管理する Control loop が 1 つ抽出された．これは，実験 1 の整形結果とほぼ同様の Control loop 階層構造が抽出できる整形結果であったといえる．エンティティあたりの Concerns 関係数 (#Concerns/ #Entities) が高かった理由としては，Collect タイプと同一階層に複数のサブゴールが Act タイプとして位置し，Collect タイプを除くこれらのサブゴールすべてが同一のエンティティを参照するような記述が複数見られたことによるものであり，Control loop 間での競合が発生するような箇所は見られなかった．

被験者 3 の整形結果は，GUI の各部品を管理する Control loop が 4 つ，操作，共通操作に関する Control loop が 4 つ，KAOS モデルを管理する Control loop が 1 つ抽出されるものであった．ツールバーの各操作がメニューバーの必要な操作を利用するという利用関係が定義されている点が，共通の機能を個別の Control loop として抽出した実験 1 の結果とは異なっていたが，被験者 2 の整形結果同様に，実験 1 の整形結果とほぼ同様の Control loop 階層構造が抽出できる整形結果であったといえる．また，被験者 4 の整形結果は，GUI の各部品を管理する Control loop が 5 つ，操作，共通操作に関する Control loop が 4 つ，KAOS モデルを管理する Control loop が 1 つ抽出されるものであり，これは，実験 1 で抽出した Control loop の構成に一致するものであった．

以上の結果から，一部複数の Control loop を包含している Control loop や，詳細に細分化された Control loop もあり，Control loop の個数には若干の違いがあったものの，実験 1 の結果も含めた 5 つの整形後ゴールモデルからは，概して GUI 部品を管理する Control loop，操作を管理する Control loop，KAOS モデルを管理する Control loop のいずれかに属する Control loop が抽出され，それらが同様の階層構造により形成されることが確認できた．

ただし，被験者 1 においては，肥大化する可能性のある Control loop が抽出されたことから，Control loop の進化に適した粒度に関するガイダンスを提供する余地はあると考えられる．

5.8　実験 4：影響範囲分析実験

5.8.1　実験概要

実験 4 では，提案手法の進化時の影響分析に対する有効性を評価する実験を実施した．具体的には，実験 3 を実施した 4 名の被験者に，従来の k’-tool と提案手法により構築された c-tool

それぞれのソフトウェア進化における設計モデル上の影響範囲を分析してもらった．本実験では，進化における影響範囲を分析するためのソフトウェア開発用文書として，要求が記述されたゴールモデルと，主要クラスと主要クラス間の関係が記載されたクラス図により構成されるクラス設計書を用意した．k’-tool のゴールモデルとしては実験 1 で用いた図 5.2 の初期（整形前）ゴールモデルを用い，c-tool のゴールモデルとしては実験 1 で整形した図 5.7 の整形後ゴールモデルを用いた．本分析実験では，進化範囲の分析は被験者のソフトウェア開発スキルにも大きく依存すると考え，k’-tool と c-tool の分析実験を同一の被験者 4 名により実施した．ただし，k’-tool 分析の前に c-tool 分析においてゴールモデルを再整形すると，k’-tool 分析での進化影響範囲同定においても提案手法の効果を与えてしまうと考えられるため，k’-tool, c-tool の順で分析を実施した．

k’-tool と c-tool いずれの進化分析においても，ゴールモデル上で進化に対する要求を分析・記述した後に，クラス設計書上での追加・変更箇所を同定してもらった．c-tool における分析では，3.5 節で示したプロセスに従って，進化に対する要求分析後にゴールモデルを再整形し，その後，gocc により生成される情報をもとに，クラス設計書上での追加・変更箇所を同定してもらった．

得られた分析結果は，各被験者ごとに各進化における追加・変更が必要なクラスとして指摘したクラス数をもとに正解率を算出し，それらを k’-tool と c-tool とで比較することにより評価した．正解率には再現率（recall）を用い，以下の数式により算出した．ここで，C は追加・変更すべきクラスの集合であり，P は被験者により同定されたクラスの集合である．

$$\text{正解率 } \textit{(recall)} = \frac{|C \cap P|}{|C|}$$

5.8.2 実験結果

本実験の実験結果として，まず，各被験者の各進化ごとの正解率を表 5.11 に示す．表 5.11 の実験結果からは，進化 2 と進化 3 において，各被験者とも c-tool の正解率が向上していることが分かる．これは，1 つは c-tool の方が，k-tool よりも，追加・変更すべきクラス数が少ないことによるものである．k’-tool においては修正すべきクラスが，MVC モデル内の各要素，つまり Model，View，Control 内の共通クラスに広く分散する傾向があったのに対して，c-tool においては追加すべき機能に閉じた Control loop の追加や修正にとどまっていた．

別の理由としては，c-tool においては MVC モデルにおける Model に関する変更影響を分析できたことが挙げられる．k’-tool に対する分析については，全被験者とも，ビューの更新と追加すべき操作（MVC における View と Controller）については，追加・修正が必要なクラスや変更内容の多くを指摘することができたが，進化 2，進化 3 で必要となる Model の変更につい

表 5.11. 実験 4 における正解率 r (=recall)

■ k'-tool

		被験者 A		被験者 B		被験者 C		被験者 D		平均	
	$\|C\|$	$\|P\|$	r	$\|P\|$	r	$\|P\|$	r	$\|P\|$	r	$\|P\|$	r
進化 1	6	2	0.33	3	0.50	3	0.50	2	0.33	2.5	0.42
進化 2	34	5	0.15	6	0.18	8	0.24	12	0.35	7.75	0.23
進化 3	8	3	0.38	4	0.50	3	0.38	1	0.125	2.75	0.34

■ c-tool

		被験者 A		被験者 B		被験者 C		被験者 D		平均	
	$\|C\|$	$\|P\|$	r	$\|P\|$	r	$\|P\|$	r	$\|P\|$	r	$\|P\|$	r
進化 1	2	2	1.00	2	1.00	2	1.00	2	1.00	2	1.00
進化 2	31	24	0.77	27	0.87	27	0.87	25	0.81	25.75	0.83
進化 3	6	4	0.67	6	1.00	6	1.00	4	0.66	5.0	0.83

ては，ほとんど指摘することができなかった．一方，提案する開発プロセスを用いた c-tool に対する分析においては，実験 2 で示したように，ゴールモデル上での変更箇所から，変更の可能性のある Control loop を特定することが可能であり，例えば，進化 2 における KAOS モデルの各要素・関係への属性情報追加など，KAOS モデル管理に関する Control loop にも変更の可能性があることを被験者が認識することができた．ただし，c-tool においても，すべての変更箇所を同定できるわけではなかった．例えば進化 2，進化 3 において，属性情報が追加されることに伴い，KAOS モデル管理に関する Control loop 内で，XML 形式への保存，XML 形式からの読込に関するクラスのメソッドを修正する必要があるが，この KAOS モデルデータの永続化に関するクラス修正を指摘できない被験者もいた．従って，提案手法においては，システムを構成する Control loop とゴールモデルの構造を関連付けることで，変更影響のある Control loop を同定することができるが，各 Control loop 内での影響の同定は，設計者のスキルに依存する部分が残ることも分かった．

さらに，共通変数を定義しているクラスの存在も正解率の違いの要因として挙げられる．実験 2 で示したように，k'-tool の進化においては共通変数を扱うクラスに対しても変更影響が及ぶが，共通変数の変更をクラス設計書上で分析するのは容易ではなく，変更の影響範囲の把握を難しくしていると言える．

5.9 実験 5：制御システムの進化

5.9.1 実験概要

続いて，進化による複雑さの変化を評価するために，清掃ロボットシミュレータ上での進化実験を実施した．本実験は制御システムでの提案手法適用を想定し，シミュレータ上の清掃ロボットの進化を対象とした．本実験では清掃ロボットに対してバッテリー管理機能や積載量管理機能，新たな清掃手段などの 6 種類の機能追加を進化として繰り返し，実装コードの複雑さの変化と修正コストを計測した．

本実験では，提案手法と比較するベースラインの手法として，集中型の Control loop をシステムアーキテクチャとする開発スタイルを想定した．本実験で扱った 6 種類の機能追加，すなわち進化は以下の通りである．

- 初期開発状態（Ver01）：吸引（Vacuum）機能を持つ清掃ロボット．ごみが存在する地点まで移動し，吸引によりごみを清掃することを繰り返す．
- 進化 1（Ver02 への機能追加）：長時間の清掃が可能となるように，バッテリー管理機能を追加する．
- 進化 2（Ver03 への機能追加）：紙ごみや缶なども清掃できるように，新たな清掃手段として Pick up 機能を追加する．
- 進化 3（Ver04 への機能追加）：多くのごみを継続して清掃するために，ごみの積載量管理機能を追加する．ごみの積載量を監視し，積載量が一定値以上になった場合，ごみ箱の場所を探し，ごみ箱の場所へ移動後，積載しているごみをごみ箱に捨てる．
- 進化 4（Ver05 への機能追加）：新たな清掃手段として Wipe 機能を追加する．
- 進化 5（Ver06 への機能追加）：通過不能であるオブジェクト（障害物）を迂回する機能を追加する．
- 進化 6（Ver07 への機能追加）：通常はごみの外見により清掃手段を決定するが，清掃に失敗した際に清掃手段を切り替える機能を追加する．

複雑さの計測には，McCabe が提唱する循環的複雑度 (Cyclomatic Complexity) [53] を用い，進化後の各プログラムコードの各メソッドに対してその値を計測した．また，本実験では，既存コードの修正コスト計測のために次のようなメトリクスを定義し，進化後の各値を計測した．

$$CMC_e = \Sigma_{v_i \in V_e} \Sigma_{m_j \in M_e} c_{ij} \cdot LOC(m_j)$$

表 5.12. 清掃ロボットに対するインクリメンタルな進化の計測結果．#Classes 欄括弧内の “M:” および “A:” ラベルは，それぞれ修正クラス数，追加クラス数を表わす．

■ベースライン手法

Ver	追加機能	#CLs	#Classes		ΔV_e	CMC	LOC
01	初期開発	1	6	–	–	–	562
02	バッテリー管理機能	1	9	(M:2, A:3)	5	638	711
03	新たな清掃手段 (Pick up)	1	10	(M:2, A:1)	1	135	818
04	積載量管理機能	1	14	(M:4, A:4)	6	942	1,106
05	新たな清掃手段 (Wipe)	1	15	(M:1, A:1)	1	277	1,150
06	障害物回避機能	1	15	(M:3)	5	1,447	1,383
07	失敗時の清掃手段切り替え	1	15	(M:4)	4	458	1,450

■提案手法

Ver	追加機能	#CLs	#Classes		ΔV_e	CMC	LOC
01	初期開発	2	7	–	–	–	537
02	バッテリー管理機能	3	10	(M:3, A:3)	5	308	733
03	新たな清掃手段 (Pick up)	3	12	(M:1, A:2)	1	80	836
04	積載量管理機能	5	19	(M:3, A:7)	2	96	1,256
05	新たな清掃手段 (Wipe)	5	20	(M:1, A:1)	1	89	1,300
06	障害物回避機能	5	20	(M:2)	5	698	1,503
07	失敗時の清掃手段切り替え	5	20	(M:4)	4	205	1,557

ここで，V_e は，進化 e において既存クラス上に新たに定義された変数および，進化 e において取り得る値が追加された変数の集合であり，M_e は，進化 e の直前に既存クラス内に定義されていたメソッドの集合である．また，c_{ij} は次のような二値関数である．

$$c_{ij} = \begin{cases} 1 & \text{if } v_i \text{ はメソッド } m_j \text{ 内に出現する} \\ 0 & \text{if } v_i \text{ はメソッド } m_j \text{ 内に出現しない} \end{cases}$$

5.9.2　実験結果

実験結果を表 5.12 と図 5.13 に示す．表 5.12 からは，提案手法においては Control loop が増えるタイミングで追加するクラス数とコード行数（LOC）が増加する傾向にあるが，一方のベースライン手法においては，修正コスト（CMC）が提案手法よりも多く要することが確認された．例えば，ver02，ver06，ver07 の構築時には，ベースライン手法は提案手法の約 2 倍の修正コストを要していた．これは，ベースライン手法で用いる集中型制御方式においては，現在

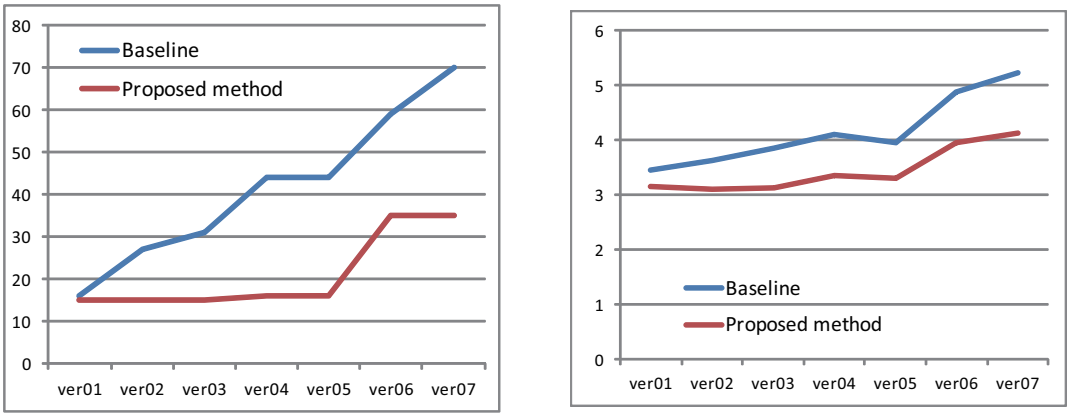

図 5.13. Cyclomatic 値: (左) 最大値，(右) 平均値

の状況を判断するための条件分岐が 1 つのメソッドに集中化する傾向があり，進化，つまり要求の変化時に，同メソッドが変更される可能性が高いことによるものである．一方で，提案手法のような開発スタイルでは，同様の条件分岐は，扱う機能や対象により，該当する Control loop に分散化されることとなる．

図 5.13 の実験結果も同様の特徴を示している．Cyclomatic 数はソフトウェアモジュールの複雑さを表現したものであり，コード内の分岐数の増加に伴い値は大きくなる．図 5.13 からは，ベースライン手法において各進化のたびに Cyclomatic 数が増加していることが分かる．これは集中化された制御部において各進化の都度，機能追加に伴い新たに定義される状態を扱うための条件分岐が追加されたことによるものである．一方，同図からは，提案手法における Cyclomatic 数が抑制されていることが分かる．これは各進化におけるコード上の変更が集中化されず，Control loop 単位で局所化しており，Cyclomatic 数が最大となるメソッドが進化後も常に同一というわけではないことによるものである．

図 5.13 の右側のグラフは Cyclomatic 数の平均値を計測した結果を示したものである．このグラフからも，提案手法においては Cyclomatic 数が抑制されていることが分かる．これは，ベースライン手法では現在の状態を把握するために複雑な分岐が必要となることと，提案手法では Control loop ごとに関心事に基づいた状態把握がなされるため，条件分岐が複数のメソッドに分散化されていることによるものである．

5.10 まとめ

本章では，提案する開発プロセスの有効性を評価するために GUI ベースのモデリングツールと制御システムを対象ソフトウェアとした 5 種類の進化実験と，その結果について述べた．

まず，実験 1〜実験 4 として，GUI ベースのモデリングツールとして，KAOS モデリング

ツール k-tool を対象としたソフトウェアの進化を扱った．実験 1 からは，提案する開発プロセスを適用することで，システムに対する要求が記述されているゴールモデルを，Control loop を包含するゴールモデルに整形することが可能であり，整形後のゴールモデルから得られた設計支援情報をもとに，実用化されているシステムを Control loop により構成されるシステム構成により実装できることが確認できた．実用化されているシステムにおいては，多くの Control loop が構成要素として同定されるが，gocc により生成される Control loop 間の階層構造情報は，システムにおける Control loop 間の関係を把握するのに有効であった．

実験 2 では，実験 1 により実装されたソフトウェアシステムが提案手法により効果的に進化ができることを確認した．本実験では，従来の k-tool のシステム構成に対しても同様の進化実験を実施したが，従来の MVC モデルに従った GUI システムにおいては，Model，View，Control に変更が及ぶような進化に対しては，進化の影響が広く及んでしまうのに対して，提案手法で用いる Control loop モデリングでは，変更箇所をゴールモデル上で同定される Control loop 内に限定することができることが確認できた．

実験 3 では，複数の被験者が整形したゴールモデルから，ほぼ同様の Control loop と Control loop の階層構造が得られることが確認され，整形プロセスの妥当性を確認することができた．ただし，一部の被験者において，肥大化する可能性のある Control loop を構築したことは，Control loop の進化に適した粒度に関する議論が必要であることを示しているといえよう．

実験 4 では，提案する開発プロセスがソフトウェア進化時の影響分析に対して有効であることが確認できた．これは，ゴールモデル上での変更箇所から，修正の可能性のある Control loop を検出することが可能であるという特徴によるところであると考えられる．この特徴は，特にシステムが大規模になり，Control loop の数が増加した場合に，影響範囲を限定するという意味で有効であると考えられる．ただし，実験結果からも示されたように，各 Control loop 内での影響の同定は，依然設計者のスキルに依存する部分が残る．従って，設計者が Control loop 内での進化の影響を同定するための，更なる支援情報の提供が必要と考えられる．

最後に，実験 5 として，制御システムでの提案手法適用を想定した清掃ロボットシミュレータ上での進化実験結果も示した．実験 5 からは，提案手法に従って抽出した Control loop 単位でシステムを構築した場合，従来の集中型の Control loop の場合よりも，実装コードの変更箇所と進化によって増加するコードの複雑さを軽減できることが確認できた．

以上の実験結果から，更なる改善の余地はあるものの，提案する開発プロセスが，GUI ベースのアプリケーションや制御システムなどの複数ドメインのシステム進化に対して有効であることが確認できた．

第 6 章

考察

本章では，ソフトウェア進化の支援を目的として本研究で提案した開発プロセスに対して，実験結果をもとに有効性と適用範囲を考察する．まず，1 章で定義した各要件に対して提案手法を評価し，続いて本開発プロセスの適用範囲について論じる．その後，提案する開発プロセスの自動化の可能性について議論する．

6.1 要件との対応

まず，4 章，5 章の実験結果に基づいて，1 章で定義したソフトウェア進化の観点から開発プロセスに求められる要件，つまり，影響分析精度の向上，コード複雑化の抑制に対して提案手法を評価する．

6.1.1 影響分析精度の向上

まず，4 章の清掃ロボットに対する実験や 5 章の k-tool 構築実験で示した各進化に対しては，いずれもゴールモデルの整形結果に基づいて，Control loop の追加・変更の範囲内で進化が実現可能であることを確認することができた．進化による変更影響の範囲を分析するためには，変更要求に対して，設計モデル上の対応箇所が同定可能な開発手法が求められる．本研究では，複数の Control loop により構成されるシステム構成をアーキテクチャとして想定し，ゴールモデルの整形により，ゴールモデル上でソフトウェアシステムに対する要求を Control loop という制御単位で分割し，システムの各機能を独立化させ，システム構成要素の責務を明確化している．本開発プロセスにおいては，一般に，主要ゴール自体の追加・削除は，Control loop の追加・削除に該当し，主要ゴール内の変更については，Control loop の振る舞いの変更に該当する．

このような特性を利用することで，4 章の清掃ロボットに対する実験結果や，5 章の実験 4 の影響範囲分析実験の結果が示すように，提案する開発プロセスでは，進化時に要求の変化によってシステム上に生じる影響の範囲をゴールモデル上で分析，限定化することが可能であった．例えば，5 章の k-tool 構築実験では，従来の MVC モデルに従った場合には，進化 2 や進化 3 のような Model の変更を伴うような進化に対して，変更影響が Model，View，Control のすべての要素に及んだ一方で，提案手法では変更箇所がゴールモデル上で同定される Control loop 内に限定されることが確認された．また，4 章の清掃ロボットに対する実験からは，gocc のコンフィギュレーションの生成機能と差分検出機能を用いることで，変化が与える影響を効果的に明確化できることも確認できた．

Control loop によるモデリングを用いた場合，変数に対する責務も同定することができる．Control loop は，入力変数，操作変数などのプロセス変数を持つモデルであることから，Control loop をモデリングの単位とすることで，各 Control loop が参照，操作する変数の範囲，つまり変数に対する責務を同定することが可能となる．例えば，Collect タイプのゴールに関与するエンティティに対しては情報収集，つまり read 権限のアクセスに相当し，Act タイプのゴールが関与するエンティティに対しては，操作変数であれば処理，つまり write 権限に相当するアクセスが発生することとなる．変数に対する責務が同定できることは，変数に対して発生する影響範囲，つまり競合の可能性を検出できることを意味している．本研究では，この競合の発生に対しては，2 つの Conflict パターンを導入し，その検出を自動化する機能を gocc に付与している．このような競合をゴールモデル上で検出し，必要に応じて対処しておくことは，進化時における他構成要素に対する変更の影響を防ぐという意味で有益であると考えられる．

影響分析精度の向上に対する提案手法の限界としては，次の 2 つが考えられる．1 つは，本開発プロセスにおいては上述の通り，主要ゴール自体の追加・削除は，Control loop の追加・削除に該当し，主要ゴール内の変更については，Control loop の振る舞いの変更に該当するが，Uses 関係における被利用側の Control loop の変更については，利用側の Control loop において変更が必要かどうかを確認する必要がある．これは変更影響が Control loop の外部に及んでいることを意味している．ただし，このような変更についても，提案手法のような Control loop 同定・構築法に基づいている場合，被利用側 Control loop の出力となる提供サービスに変化があるかどうかを検査することで，変更の影響が利用側に及ぶかどうかを判断することができる．また，提案手法のように Uses 関係を明確に定義しておくことにより，ゴールモデル上で Control loop 間の影響範囲を特定することもできる．

もう 1 つは，5 章の実験 4 の影響範囲分析実験でも示されたように，Control loop 内での影響範囲分析に関しては，ゴール記述の差分情報から設計者が判断する必要がある点である．この点においては，各 Control loop が過度に大きくならないように主要ゴールを同定する工夫が

必要と考えられる．この点については，Control loop の粒度を決定するようなドメイン固有の知識に基づいたガイドラインの提供が有効であると考えられる．

6.1.2 コード複雑化の抑制

5 章で示した清掃ロボットの進化実験（実験 5）からは，提案するゴールモデル整形に基づいた複数 Control loop の設計・実装により，従来の集中型の制御方式と比較して，既存コードの修正コストと Cyclomatic 数により計測されたコードの複雑さが軽減されていることが確認できた．また，5 章の k-tool 進化実験においても，提案手法においては，機能を横断するような共通クラスがなく，Control loop 内での限定されたコードの修正となるため，進化を繰り返すことによるコード複雑化の抑制がなされていることが確認できた．Bhattacharya [54] らは，ソフトウェア進化のための指標となるメトリクスの提案とともに，「高い集積性を持つソフトウェアモジュールは修正コストが低い」という仮説を立て，その仮説が有効であることを実証している．本提案手法では，複数の Control loop を抽出することで，各 Control loop 内に Collect，Analyze，Decide，Act という情報収集から処理の実行までの一連の振る舞いを凝縮し，構成要素間の依存関係の最小限化を図ることでシステム構成要素のモジュール性を高めているが，これにより既存コードの修正頻度が縮小されることは，Bhattacharya らの立証した仮説を裏付けていると言える．また，5 章の 2 種類のソフトウェアの進化実験で示されたように，提案手法を用いた場合，機能追加を既存 Control loop の修正ではなく，新規 Control loop の追加により実現できる場合がある．このような既存コードの修正を避ける機能追加も，修正コストを軽減していると言えよう．

4 章で導入した Control loop の実装法では，進化の種類による変更の影響を限定化できるという効果がある．本研究では，Control loop のアクティビティから，Control loop を構成するコンポーネントを，Collect タイプ，Analyze & Decide タイプ，Act タイプの 3 種類に分類したが，Collect タイプ，Act タイプは環境に関与，作用するコンポーネントであり，Analyze & Decide タイプは Control loop の状態遷移を管理し，適切な制御を決定するコンポーネントに該当する．従って，例えば，プラットフォームやセンサ，アクチュエータなどの環境変更を扱う進化であれば，外部とのインタフェースとなる Collect タイプ，Act タイプの必要箇所を変更すればよく，要求の変化に対しては，主に Analyze & Decide タイプが変更対象となり，必要に応じて，Collect タイプ，Act タイプのコンポーネントを変更することになる．従って，本研究で導入する Control loop モデルは，進化の要因の観点からも変更の影響を限定化し，これによりコードの複雑さを抑制していると考えることができる．

提案手法の限界としては，複雑さを十分に抑制するために，Control loop の粒度に関して注

意を払う必要がある点が挙げられる．もし，複数の機能要求を同一の Control loop 内に包含させた場合，例え影響範囲が Control loop 内に閉じていたとしても，他方の機能要求の実装部分に影響が及ぶ可能性があるからである．この点については，前節の影響分析精度の向上に関する議論でも取り上げたように，Control loop の粒度を決定するようなドメイン固有の知識に基づいたガイドラインの提供が有効であると考えられる．

6.2　適用範囲

続いて，提案する開発プロセスの適用範囲について論じる．

6.2.1　ゴールモデル上における判断

まず，本研究において構築するシステムの構成はゴールモデルにより決定されるため，ゴールモデル上での適用範囲の判断について述べる．本研究ではエンティティの責務を Control loop ごとに分離することで，各 Control loop におけるプロセス変数を同定している．従って，Control loop 間で生じる競合や干渉については，これらのプロセス変数がエンティティとして同定されていることと，他の Control loop に影響を与えないようなプロセス変数の割り当てが必要となる．本研究では，Entity-conflict パターンによるエンティティ競合の回避を経て Control loop を分離することで，Control loop に関与するプロセス変数に対する競合の解消を図っている．従って，もしこれらのプロセス変数を上手く単一の Control loop に割り当てられない場合は，該当 Control loop の責務の範囲が大きくなり，結果として，システムを構成する Control loop の数が減少することとなる．

本手法では，整形プロセスにより整形されたゴールモデル上で，システムの構成要素，つまり進化の単位となる Control loop を同定するため，Control loop が少数しか抽出されない場合は適用の効果が小さくなり，複数の Control loop が同定される場合に本開発プロセスの効果が期待できる．少数の Control loop しか抽出されないゴールモデルにおいては，一つは，システムにおける多くのアクションが，同一の入力変数に依存するような場合が考えられる．このようなゴールモデルとなる可能性のあるシステムとして，例えば，数値計算アプリケーションや，財務・会計システムなどがこのクラスに属すると考えられる．

一方で，多種にわたる入力変数と入力変数ごとに独立したアクションが定義されるシステムにおいては，複数の Control loop が抽出される可能性が高い．5 章の実験結果からも示されたように，進化を考慮した場合には，影響分析の観点からも，複雑さ抑制の観点からも，各 Control loop は過度に大きくならず，機能の単位で Control loop が形成されるのが理想的であるといえる．

このようなシステムとしては，例えば豊富な GUI を持つシステムや Web アプリケーションが該当する．これらのシステムは，部品（Widget）群ごとに入力変数やアクションが独立していることから，Control loop が複数構築されることが期待できる．また，アクションが独立していることは，Control loop 単位での変数の集約化も可能であることを示していると言える．これらのシステムにおいては，部品群単位での Control loop 構築が可能であり，Control loop 単位での進化が容易であると言える．

他に，多種多様な情報（入力変数）を扱い，状況に応じて取るべきアクションが異なるシステムとして，ユビキタスアプリケーションを挙げることができる．ユビキタスアプリケーションに対しても，入力情報収集とアクション実行に関する責務が集約され，進化の影響範囲を分析しやすいという特徴は同様である．また，Control loop をシステム構成要素とすることは，状況変化への対応が必要であるユビキタスアプリケーションにおいて，状況に応じた制御の切り替えという観点からも有益であると考えられる．

6.2.2 非機能要求への対応

非機能要求の観点から提案手法の適用範囲について議論する前に，ゴールモデルの変更と Control loop の対応関係について整理する．提案手法では，進化の要因となる要求の変化をゴールモデル上で抽出し，ゴールモデルの整形により，進化要求を固有の Control loop に対応付ける．整形により，進化要求は複数の Control loop に分配される場合もあるが，定義 3.3.4 の条件 1 により，進化要求のうち機能要求に関するものは Control loop に対応する主要ゴール以下に含まれる．非機能要求に関しては，固有の Control loop に関連するものは該当する主要ゴール以下に含める．ここで，各主要ゴールに包含される非機能要求については，基本的には該当する Control loop に割り当てられる責務として考えることができる．一方で，各主要ゴールに包含されない非機能要求や，システム全体に影響すると考えられる非機能要求については，本手法の想定するアーキテクチャの特徴により満足されるかどうかを判断する必要がある．合致しないものについては必要に応じてアーキテクチャを再検討することとなる．

ここでは，POSA のソフトウェアアーキテクチャ非機能特性 [55] に沿って，本手法で想定するアーキテクチャの特徴について議論する．変更容易性（Changeability）については，先のコード複雑化の抑制の議論の通り，Control loop 単位で進化を扱うことにより，継続的な進化に対しても変更に柔軟に対応可能なアーキテクチャであると判断できる．他システムとの相互運用性（Interoperability）の観点からは，他システムからの入力を入力変数に，他システムへの出力を被制御変数，あるいは操作変数としてモデリングすることで，他システムとの相互運用を意識したシステム開発が可能であるといえる．一方，効率性（Efficiency）の観点からは，

Control loop として同定された各構成要素を並行に動作させる場合には，プロセス実行のオーバーヘッドを考慮する必要がある．信頼性（Reliability）については，各 Control loop が独立した制御プロセスを形成することから，特定の Control loop 内に閉じた障害に対しては一定の頑強性を期待することができる．テスト容易性（Testability）については，Control loop 単位での機能テストが可能である一方で，各 Control loop の統合に対するコストを考慮する必要がある．特に，複数の Control loop を並行動作させる場合には，テストが難しくなると言える．最後に再利用性（Reusability）については，提案手法が Control loop の観点からシステムを細分化することで各構成要素間の依存関係を減少させることを目指したものであることから，各 Control loop 単位での再利用により，ソフトウェア開発におけるモジュールの再利用が期待できる．また，ゴールモデルの段階でモジュール化を進めるという特徴は，再利用時における既存モジュールの要求把握を支援しているとも言える．もし，これらの特徴に矛盾する非機能要求が存在する場合は，システム設計段階において本研究で示したアーキテクチャが利用できないなど，本開発プロセスの適用が限定的なものとなる．

6.2.3　競合の解消

本研究で導入した整形プロセスにおいては，3.3.6 節で説明したように，複数 Control loop からの操作変数への共通アクセスを競合として検出し，また，同変数に対する責務の集約を目的とした解消法に従ってゴールモデルを整形する．3.3.6 節で導入した解消法を適用することで，操作変数へのアクセスは特定の Control loop に集約されることになるが，この解消法を複数回適用すると，複数の Control loop が共通する複数の Control loop を利用（Use）する構造になる場合がある（図 6.1）．

図 6.1 のような Control loop 階層構造の場合，操作変数に関する競合は解消されるが，一方で，被利用側 Control loop のサービス提供ポリシーを適切に設定しなければ，サービス利用側 Control loop 群において互いにサービス提供待ちとなる，いわゆる Control loop 利用に関する競合（デッドロック）が発生する可能性がある．この場合，以下のような対応が有効と考えられる．まず，被利用側 Control loop 群において統一した利用側 Control loop の優先順位を設定する手段が考えられる（対応 1）．これは Control loop の階層構造，つまりゴールモデルの構造を変更しない対応として有効な手段であるが，すべての場合においてこのようなルールが適用できるわけではない．対応 1 の適用が難しい場合は，競合回避のメカニズム実装は複雑となるため，ゴールモデルをさらに整形する対応が有効と考えられる．

図 6.2 は，図 6.1 のような Control loop 利用に関する競合を回避する 2 種類の整形手段を示したものである．まず 1 つの対応として，被利用側 Control loop を統合するという整形が考

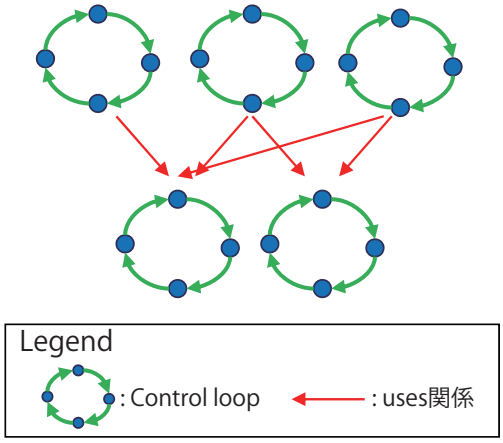

図 6.1. 複数の Control loop による複数 Control loop の共通利用

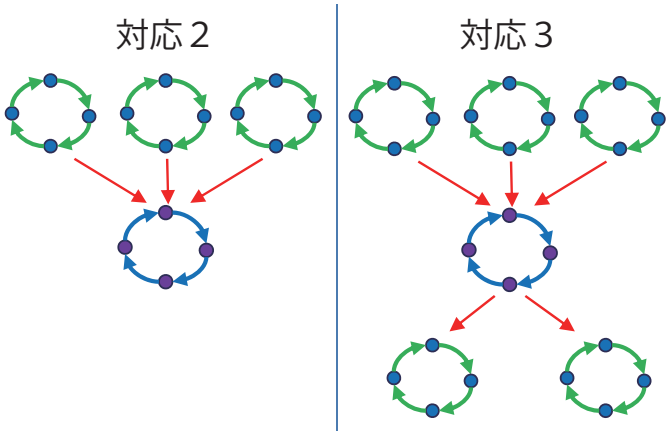

図 6.2. 複数 Control loop の共通利用（図 6.1）への対応法.（左図：被利用側 Control loop の統合，右図：調整用 Control loop の導入）

えられる（対応 2）. ただし，このような対応は Control loop を肥大化させるため，被利用側 Control loop のゴール群が集約するに適した場合にのみ適用すべきであろう. もう 1 つの整形は，被利用側 Control loop を利用する新たな Control loop を導入するというものである（対応 3）. この場合，新たに導入する Control loop は，利用側 Control loop に対してどの Control loop のサービスを提供するかを状況に応じて決定するという責務を持つことになる. いずれの対応が適用可能であるかは競合の内容によって異なるが，Control loop の粒度の維持や Control loop 階層の複雑化抑制を考慮すると，対応 1〜対応 3 の順で適用可能性を検討するのが良いと考えられる.

6.2.4　ゴール指向要求分析法

続いて，提案手法において用いた各要素技術に関する適用範囲について論じる．まず，本研究ではゴールモデルの構築，整形に KAOS モデルを利用している．KAOS はシステムの要求に関連するゴール間の関係を定義するだけでなく，*Concerns* や責務割り当て関係といった概念間の関係や，システムの動的側面を記述するオペレーションもモデル構成要素として持ち，ゴールとの明確な対応関係を記述することができるため，設計支援情報を生成する提案手法においては，利用する要求記述法として適している．提案手法では，整形プロセスにおいてゴールとエンティティとの関係を利用し，クラステンプレート生成に，ゴールとオペレーションとの関係を利用している．今後さらに要求記述と実装コードとの関連を強化する場合，ゴールとエンティティやオペレーションとの関係を記述できる KAOS モデルの特徴が有益であると考える．本研究では進化を支援する設計情報を生成する `gocc` を構築したが，要求分析結果を構造化された文書，つまり XML 形式で出力できるツールが存在する点も，要求分析法に求められる要件であるといえる．

ゴール指向要求分析法の代表的な手法としては，KAOS の他に i*[56] がある．i*は KAOS と同様に，ゴールモデルを記述可能な分析法であるが，ステークホルダー間の要求を分析するといった初期の要求フェーズの活動に重点を置いた手法であり，特定のゴールに対するシステムへの責務割り当てなど，システム設計を意識した厳密な記述が難しいことから，本研究で導入する開発プロセスにおいては KAOS の利用が適していると考えられる．

その他の代表的なゴール指向要求分析法として，NFR（Non-Functional Requirement）[57] があるが，NFR はその名の通り，性能などの非機能要求を構成要素，すなわちゴールとして扱ったゴールモデルを構築する．提案する開発プロセスは機能要求が記述されていることを整形プロセスの前提条件としていることから，NFR のモデルを KAOS モデルの代替とすることはできない．

本研究では, ゴールモデルの整形プロセスにおいて Control loop パターンと 2 種類の Conflict パターンを導入した．ゴールモデルにおけるパターンは，自然言語によるゴール記述よりも形式的な分析が可能である一方で，時相論理によるゴール記述よりも適用が容易であることから，一定の品質を保証したゴールの詳細化が可能である．ただし，時相論理によるゴール記述を用いる場合と比較して，厳密な競合状態の検出や回避策の検討ができないため，厳密な競合対処や情報収集法の検討が必要になる場合は，追加の検証が必要となる．

6.2.5 プログラミングフレームワーク

本研究で実装したプログラミングフレームワークは JADE 上で動作するため，その適用は JADE が動作する環境上に制限される．ただし，JADE は Java 言語により構築されたプラットフォームであることから，PC 環境およびサーバ環境の Java である JavaSE，JavaEE が動作する環境上での適用が可能である．また，JADE には小型デバイス用の Java である JavaME に対応した，Leap [58] と呼ばれる軽量版が存在する．従って，本研究で拡張実装した各クラスを同様に軽量化することにより，Leap が動作する環境上でのシステム実装も可能となる．

なお，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，本研究ではコンポーネントを制御するために CyclicBehaviour や SimpleBehaviour といった JADE 上のスーパークラスを利用して実装用のコンポーネントクラスを API として提供している．従って，異なるマルチスレッド・プログラミング環境上でフレームワークを構築する場合は，これらのスーパークラス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．マルチスレッドプログラミングをサポートしていないプログラミング言語を利用する場合には，Control loop の並行動作が困難であり，本手法の適用は限定的となる．

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．プログラミングフレームワークの実現にあたり拡張利用した JADE においては，実行の最小単位がビヘイビアであり，ラウンドロビンでビヘイビアの実行スケジュールが制御されるため，適用時には各コンポーネントの実行サイクルに留意する必要がある．例えば，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，該当 Control loop の周期が保証すべき単位時間内に収まっていなければならない．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するなどの手段を検討する余地はある．また，本プログラミングフレームワーク上で Control loop を実装した場合，構成コンポーネントが待機状態であったとしても，コンポーネントのプロセスは継続することとなる．このため，携帯端末などの主記憶容量に制限のある環境でコンポーネント数の多いシステムを実装する場合にも，適用に際して留意する必要がある．このような場合，*finalize* メソッドを利用した同時起動コンポーネント数の抑制などを検討する必要がある．

6.3　自動化の可能性

最後に，提案手法の自動化の可能性について議論する．まず，コンフィギュレーション生成などのような，ゴールモデルの構造やゴールモデル上の各要素の情報を利用した範囲での情報抽出，モデル変換については，自動化が可能である．本章で導入した gocc においても，大半の出力はゴールモデルの構造およびゴールモデル上の各要素の情報を利用したものである．

現在 gocc が生成する情報の多くは，システムの構造的な側面に関する情報である．各 Control loop の振る舞い，つまり動作を検討する場合も，ゴールモデル上のゴールに付与された情報を形式的に収集・抽出することは有効であると考えられる．現在の提案手法では，時相論理式などを用いて記述されるゴール記述は利用していないが，Maintain や Achieve などのゴール記述に対するパターンを効果的に利用することで，Control loop が取るべき動作の種類を同定する助けにはなるであろう．

一方で，ゴールモデルの構築，ゴールモデルの整形，およびコードの実装に対しては自動化が難しい．まず，ゴールモデルの構築に関しては，主にサブゴールへの詳細化を設計者の手作業により実施する必要がある．しかし，サブゴールへの詳細化は，どのような観点により詳細化，つまり分解するかによりサブゴールが全く異なったものとなる可能性があるとともに，詳細化の段階でサブゴールの抽出漏れが発生する場合も少なくないため，自動化のみならず，品質の維持も容易ではないといえる．従って，ゴールモデルの構築については自動化が困難である．ただし，いくつかの既存研究やガイドラインを提示することで，設計者の負担を軽減することは必要であろう．例えば，提案手法ではゴールモデルとコンフィギュレーション，つまりシステム構成とが対応付けられるため，反復的な分析を支援するような自動化により，コンフィギュレーションの提示がゴールモデル詳細化の 1 つのガイドラインとなり得る可能性がある．また，文献 [40] に示されたゴール詳細化のパターンや Control loop 同定を促す独自パターンの提供，チェック機能の自動化も，ゴールモデルの品質向上に有効であろう．

ゴールモデルの整形についても，主要な部分の多くが自明でない情報の追加，つまり, エンティティの追加や共通ゴールの切り出し，プロセス変数の同定などであるため，大半は設計者の判断が介入する必要があり，自動化の範囲は限定される．しかし，充足性の判定や，プロセス変数のチェック，責務の割り当てなどについてはゴールモデルの解析により自動化が可能であり，ゴールモデル整形の一部の支援は可能である．

実装モデル構築に対する自動化に関しては，現在のテンプレート生成機能などで利用している，構造情報に基づいたコード生成は可能である．しかしその一方で，Control loop やコンポーネントの動作，つまり振舞いに対するコードの自動生成は一般に難しい．振舞いに対す

るコード生成を推し進めるには，モデル駆動アーキテクチャ [59] で用いられる制約記述言語 OCL [60] や，形式仕様記述言語 [61, 62] に分類される VDM [63, 64]，JML [65] などの利用が有効である．ただし，実装モデル構築の自動化を推進するためには，本開発スタイルにおける，Control loop の仕様記述文法を定義し，その仕様文法に従った仕様を記述，決定するための属性追加や記述制約などの拡張が必要となる．

第 7 章

関連研究

7.1　はじめに

本章では，本研究で用いた各要素技術に対する関連研究について論じる．まず，要求記述と設計モデルとの連携に関する研究について概観し，続いてアーキテクチャモデルと，実装フレームワークに関する研究について述べる．その後，ソフトウェア進化に関する実装技術の関連研究として，ソフトウェアシステムの変更のタイミングの観点から関連する各研究の動向について述べ，本研究の位置づけについて議論する．最後に，提案手法の拡張分野として考えられる，自己適応システムにおける関連研究について述べる．

7.2　提案する開発プロセスに対する関連研究

7.2.1　要求記述と設計モデルとの連携に関する研究

Nuseibeh [28] が指摘するように，ソフトウェアに対する要求とシステムアーキテクチャとは相関関係にあり，要求分析結果をシステムアーキテクチャに反映させることは重要である．しかしながら，要求記述とシステムアーキテクチャとを明示的に関連付ける研究は多くはない．

ゴール指向要求記述を利用したシステム構成決定法として，Lamsweerde [33] は，KAOS 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．この手法においては，各ゴール達成の責務を持つエージェントをコンポーネントに割当て，エージェント間のデータフローの情報から，コンポーネント間の接続を抽出する．しかしながらこのような構築法では，まずシステムに必要なエージェントをゴールの責務割り当てにより決定する必要があるが，この判断は単純なものではない．例えば，過度に多くのゴールをエージェントに割当てると，結果としてコンポーネントの単位が大きく，変化の影響を受ける範囲を明確に分析できなかったり，

新機能の追加など，変更に対する影響がコンポーネント内の他の部分に及んでしまったりと，進化に対して十分な設計ができない．清掃ロボットシミュレータの例では，通常は移動やごみ清掃の機能は清掃ロボットに割当てると考えるが，この場合，すべてのゴールが清掃ロボットに割り当てられ，コンポーネントとして清掃ロボットしか抽出されないため，ごみ積載量管理の機能を追加する際に，清掃ロボットのコンポーネントを変更するという情報しか得ることができず，影響範囲を分析することができない．従って，ソフトウェア進化を考えた場合に，Lamsweerde の手法においては，エージェントへの責務割り当ての難しさが，進化時の影響分析を難しくしていると言える．

同じくゴール指向要求記述を利用したシステム構成決定法として，Yu らの研究 [34, 66] がある．Yu らは，機能に複数のバリエーションがある携帯電話や家電機器などのプロダクトライン型開発 [67, 68] を想定し，高い多様性を持つソフトウェアシステムの設計にゴールモデルを用いる．Yu らの手法においては，本研究同様に，ゴールモデル中のゴールモデルをコンポーネントに変換し，AND-refinement リンクと OR-refinement リンクをコンポーネント間の接続に変換する．この手法はゴールモデルに一対一対応するコンフィギュレーションを生成可能であるという点で，ソフトウェア進化を考えた場合にも，要求変化に対するコンフィギュレーション上での変更箇所の同定という観点では有効のように見える．しかしながら，Yu らの手法においては，入力となるゴールモデルの条件や変換対象とするゴールの範囲については言及していないため，コンポーネントの独立性や責務が十分に明確化されないことになる．その結果，ゴールモデルに新たなゴール群を追加した場合に，少なくとも結合部分に変更の影響が及ぶと考えられるが，ゴールモデル上での変更をシステムの設計，実装上でどう扱うべきかは明らかではない．例えば，本論文で用いた清掃ロボット開発の例において，ごみ積載量管理に対する記述を追加した際に，既存のゴールモデルに対する結合部分に変更の影響が及ぶこととなるが，これを設計段階でどのように扱うべきかは明確ではない．また，ゴールに一対一対応してコンポーネントを変更するため，類似のゴールが複数あった場合に，それらが共通の変数をアクセスすることで競合発生の可能性が生じることとなる．これらの問題は，進化時の影響範囲の把握を困難にする要因となる．

ゴールモデルから設計モデルへの変換手法としては，Heaven ら [69] の研究がある．Heaven らは，KAOS モデルの要素，つまりゴール，要求，エージェント，オペレーションなどを UML モデル図中に，ステレオタイプとタグで表現することで，KAOS 記法から UML 記法へとマッピングする手法を提案している．しかしながら，この変換は，UML モデル中に KAOS モデルを組込むことを意味しており，コンポーネントの導入やシステムアーキテクチャの決定などの，設計上の観点からの新たな解釈を提供するものではない．Sombat ら [70] は，KAOS モデルの要素とその構造情報を利用して UML のクラス図を生成する変換手法を提案している．

Sombat らの手法では，KAOS モデルにおけるオペレーションやエンティティ，イベントの接続関係から，クラス図の構造を決定し，ゴールモデルの構造から，クラスの振る舞いの制約となる OCL 制約 [60] の記述箇所を同定する．Sombat らの手法は要求記述から設計モデルの構造を決定する一つの手法であるが，本研究のように影響範囲の局所化や，それにより生じるコード複雑化の抑制などの進化の容易性を考慮したものではない．

ゴールモデルを用いた他の設計モデルへの変換としては，フィーチャーモデル [71] への変換を実現する研究がある．Uno ら [72] は，複数のゴールモデルを統合することで，多様性を含んだフィーチャモデルを構築する手法を提案している．また，Asadi ら [73] は，ゴールモデルとフィーチャーモデルとを対応付けることにより，顧客の要求に必要な機能をフィーチャーモデル上で特定する手法を提案している．これらの研究は，多様な要求に対して，シリーズ製品が持つべき機能を統合して記述する場合や，その機能群から特定プロダクトの構成を一意に決定する場合に有効な手法である．

また，ゴールモデルからの設計情報の抽出法として，Yu ら [74] はゴールモデルからのアスペクトの発見法を提案している．Yu らの手法では，機能要求と非機能要求とを達成するタスクを発見し，そのタスクを機能に関するタスクと，非機能要求を達成するためのアドバイスタスクに分離することにより，横断的関心事を発見している．この手法においてはアスペクトを発見することが目的であるが，関心事，つまりシステムに実装すべき機能を分離するという観点では本研究と類似していると言えよう．

一方で，ゴールモデルを変換元のモデル（ソースモデル）ではなく，変換対象，つまりターゲットモデルとする研究もある．Yu ら [75] は，ソースコードからゴールモデルを導出するリバースエンジニアリングを実現するために，ステートチャート図を経由した変換手法を提案している．ソフトウェアの進化を考えた場合は，初回開発時の要求記述を入手できない，あるいは進化により追跡可能性を喪失しているシステムも考慮する必要があり，本研究で提案した開発プロセスを適用する場合，このような逆方向の変換手法により生成されたゴールモデルを整形プロセスの入力とする場合も考えられる．

本研究では，Control loop を要求記述上で分離するためにゴールモデルを用いたが，ゴールモデル以外の要求記述を利用した，ソフトウェア進化を考慮したシステム設計モデル構築法もいくつか提案されている．Omote ら [76] は，要求の記述にユースケース図 [77] を用い，アクションの順序関係を定義したアクティビティ図を経由し，クラス図を決定する開発プロセスを提案している．Omote らの研究においては，各モデル間にリンクを定義しておき，進化時の要求変化をユースケース図に記載した後に，リンクをたどることで，各モデルに変更を伝播させる．進化を要求の変化から捉えるという点では本研究と同様であるが，設計モデルの構築に変換手法を用いているわけではなく，設計モデルとの追跡可能性をリンクとして手動で設定す

る点が異なる．ユースケース図上での変化を考慮した設計手段としては，Ecklund らが提案する Change case [78] を利用する方法もある．Change case は，システムが将来サポートしなければならなくなるかもしれない潜在的な要件を記述するユースケースであり，通常のユースケースと，変更要求とを関連付けた様式で記述するものである．Change case に対して分析，設計を進めることにより，設計・実装モデルに対する進化の影響を予測することができるが，分析・設計モデルとの追跡可能性を明示的に提供しているわけではなく，変更による影響範囲の分析については設計者の分析能力に依存することになる．また，本研究で導入するコンフィギュレーションの決定法のように，要求モデルに適した設計モデルを提示するものではない．

要求記述から設計モデルへの追跡可能性（Traceability）[29, 30] も，ソフトウェア進化に伴う変更の確実な実装に重要な性質である．追跡可能性を扱った研究の多くは，要求記述中に出現する語（word）をもとに，設計モデル，あるいは実装コード上の対応箇所を決定するものである．例えば，Cleland-Huang ら [79] は，非機能要求をノードとして表現したゴールモデルである NFR（Non-Functional Requirement）[57] のリンクを利用したトレーサビリティの管理手法を提案している．この手法では，ドキュメント内での単語の出現頻度をもとに，確率モデルを利用することで関連オブジェクトを決定している．また，Hayes ら [80] は，Latent Semantic Indexing (LST) という特徴ベクトルの次元を下げる手法を利用することで，単純なキーワードマッチングではない相関関係を抽出した追跡手段を提案している．これらの手法の特徴としては，要求記述中に出現する語（word）をもとに，情報検索（IR）技術を利用することで設計モデル上の対応箇所を決定するということであり，多くの研究では設計文書中の語との関連を決定している．このため，進化時の厳密な影響分析を実現するには，設計文書と，その他の図形表現などによる設計モデルや実装コードとの対応関係を明確に定義しておく必要がある．また，効果的なソフトウェア進化を実現するためには，本研究における整形プロセスのような変更の影響範囲を限定するような仕組みも別途必要となる．これらのアプローチとは異なったものとして，Umemura ら [81] は，要求記述と設計モデル上に出現する単語をもとに非機能要求に関するスペクトル分析をそれぞれのモデルに対して実施し，その値を比較することで，モデル間の整合性を検証する手法を提案している．この手法では，非機能要求ごとに分離された各成分値を比較することで，要求記述と設計モデル上のギャップを判断することができる．ただし，この手法は整合性の検証を目的としたものであり，本研究で扱ったような変更箇所の分析を実現するには，検証結果をもとに設計モデルの変更箇所を別途決定する必要がある．

ソフトウェア進化に関連する特性としては，要求記述と後継モデル間の追跡可能性のほかに，コード間の依存関係（Dependencies）についても着目すべきである．依存関係は大きく，コードの構造的制約から生じる構造的依存関係（Structural dependencies）と，機能の変更により生じる暗示的な依存関係である論理的依存関係（Logical dependencies）[82] に分類するこ

とができる．構造的依存関係は機械的に判断できるものであり，グラフ構造に基づいて変更影響の伝播をモデル化するものや [83]，DSM (Dependency Structure Matrix) と呼ばれるソースコード上の依存関係が記述された行例に対して，設計者があらかじめ定義した設計ルールが遵守されているかを検査する手法 [84] などが提案されている．

ただし，近年の研究は，当初想定されていなかった機能の追加など，ソフトウェアの意味的な変更を扱う必要が増えていることもあり，論理的依存関係を扱ったものが多い．例えば，Oliva ら [82] は，Java FLOSS と呼ばれるグループウェアソフトウェア開発プロジェクトにおけるバージョンリポジトリ情報を統計的に解析することで，論理的依存関係の発生源を分類しようと試みている．また，D’Ambros らは，依存関係をレーダー状のモデル上で可視化する Evolution Radar [85] を提案している．これらの論理的依存関係の分析には，通常はバージョン管理システムの変更履歴などを利用するため，一般に，論理的依存関係の同定には実装コードとその変更履歴が必要となる．また，構造的依存関係の分析にも実装コードは必要となる．これに対して本研究では，変更影響の局所化を目指したものであり，ゴールモデルの整形により依存関係を Control loop 内と Uses 関係に限定する．このような整形は，依存関係の集約化を支援することとなるが，これは同時に依存関係の同定についても支援しているといえる．

ソフトウェア進化に類似した開発プロセスとして，派生開発 [86] がある．派生開発は，既存のソフトウェアに対して，新しい機能を追加することで新しいソフトウェアを構築する開発手段であり，清水が提唱する XDDP （eXtreme Derivative Development Process）[86] がその代表的な方法論として知られている．XDDP の特徴は厳密な変更制御にあり，変更要求に関連して変更すべき仕様を変更要求仕様書上に詳細にリストアップし，トレーサビリティ・マトリクスと呼ばれる行列を利用することで，変更要求仕様書上の各項目と実装コード上の変更すべきモジュールを明確に関連付ける．従って，変更要求とソフトウェア仕様とを関連付けるという点では，本研究と同様の問題意識をもつものである．ただし，XDDP においては仕様書が存在しない場合も考慮しているため，変更箇所の同定はコード上の調査を前提としている．これに対して本研究は，整形プロセスを導入することで要求記述上でモジュール化を実現し，これにより設計モデル・コード上の変更影響の局所化を狙っている．このコード上の変更影響の局所化は XDDP におけるトレーサビリティ・マトリクスの構築を支援するものであることから，要求記述が管理可能な場合においては，本研究で導入した整形プロセスは XDDP のプロセスを支援するものであると考えられる．

ゴール指向要求分析法においては，3 章で紹介したもの以外にも，分析を支援するためのいくつかのパターンが提案されている [39]．Darimont ら [40] は，環境からの刺激に対してシステムが反応するという動作を表現するための Stimulus-respond パターンを提案している．本研究で提案する Control loop パターンは，環境情報と処理内容を組にして記述するという観点で

は Stimulus-respond パターンと類似しているが，構成要素として，ゴールだけでなく，プロセス変数，つまりエンティティも包含しているパターンである点，分解したサブゴールだけでなく，親ゴールにも制御の責務を割当てている点が異なる．また，Collect や Analyze & Decide などラベル付けされた各ゴールをコンポーネントとして抽出するという点で，パターンの利用目的も異なる．

複数の Control loop をシステムの構成要素とする場合，Control loop 間の競合を検出する必要がある．競合検出に関するパターンとして，Lamsweerde らは，曖昧なゴール分析結果から矛盾を検出するための Obstacle resolution パターン [87] を提案している．本章で述べた Entity-conflict パターンも競合個所を特定するという点では似ているが，Entity-conflict パターンでは特にエンティティ，つまり変数に対する競合を扱う点が異なる．また，Obstacle resolution パターンでは，ゴールを洗練化（分解）する際に，矛盾を検出するための適切なドメイン知識を用いてゴールを分解する必要があるが，Conflict パターンはゴールモデルの構造を形式的に解析して競合するリソースを検出するため，競合の検出手段も異なる．

ゴールモデル上での競合検出に関しては，Obstacle resolution パターンを含めて，形式的な検出法 [41, 87, 39] が提案されている．これらの検出法は，ゴールモデルの構造と，各ゴールに記述されている時相論理記述を利用することで，ゴールモデル上に潜んでいる矛盾を検出するものである．例えば文献 [41] 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という手法が提案されている．しかしながら，これらの手法においては，競合検出のために適当なドメイン知識を用意しておく必要がある．そのため，KAOS の検証ツールとして知られる FAUST [88] 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 [89] など，現段階では手動による検出が必要である．ただし，競合検出に関しては，Obstacle resolution パターンやヒューリスティック [41] が提案され，検出に要する負荷は軽減の方向にある．当分野における今後の成果が期待される．

7.2.2 アーキテクチャモデル，実装，フレームワークに関するもの

ソフトウェアアーキテクチャとしてのコンポーネントの観点からは，本研究で用いた拡張 Darwin モデルの他にも，多くのコンポーネントモデルが提案されている．例えば Taylor らの C2 [90] や Shaw らの UniCon [91] は，コネクタを明示的にアーキテクチャの構成要素とし，C2 はシステムの動的変更を実現している点，Unicon はリアルタイム性に関する解析を支援しようとしている点が特徴である．また，Allen や Garlan の Wright [92, 93] は，コンポーネントの接続部を形式化することにより，インタラクションの正当性を証明することを目指し

たコンポーネントモデルである．Luckham らの Rapide [94] は，イベントベースの分散システムを記述するコンポーネントモデルであり，イベントの半順序集合を出力するシミュレーションが可能である点が特徴である．これらのコンポーネントモデルと比較して，本研究で用いた Darwin とその拡張モデルは分散化されたシステムを対象としたモデルであり，特に拡張 Darwin モデルにおいては，コンポーネントの状態の可視化が可能である点が特徴である．本研究では，状態の可視化と依存関係の記述といった特徴が，Control loop 内でのコンポーネントの制御記述に有効と考えられたことから，拡張 Darwin モデルを Control loop を構成するコンポーネントの実装用モデルとして用いた．

ソースコードの進化に関連する研究は数多くあるが，ここでは本研究に関連するもののみ挙げる．まず，本研究ではゴールモデルの差分情報を進化に必要な設計情報の収集に利用したが，ソースコードレベルでの差分検出については，Kim と Notkin の LSdiff (Logical Structural Diff) [95] や Dagenais らの SemDiff [96] がある．LSdiff では，コード上の差分検出結果から，構造としての変化を推測し，例外を含めて差分結果を生成する．例えば「クラス A を除く，クラス B のサブクラスのすべてのメソッド b が削除された」などを意味する差分結果出力が可能である．また，SemDiff では，プログラミングフレームワークの API に対して，public API として表面には出てこない内部の変更箇所も特定し，利用側プログラムに必要な変更を提示するものである．これらの手法は従来の diff によるプログラム比較よりも，構造やグループを考慮した差分結果を出力するという意味で有効なツールであるが，既に変更が施されたコードを対象とするため，ソフトウェアの進化要求をコード上に実装する局面で利用するものではない．また，それぞれのツール単体では進化の要因となる要求の変化を把握することは出来ないため，要求の変化が漏れなく正確に設計モデル上にマッピングされていることが前提となる．その他のソースコードに関する研究については，7.3 節で変更のタイミングの観点に基づいて議論する．

本研究においては，Control loop 実装を支援するフレームワークとして，エージェントプラットフォーム JADE に対して，特に，コンポーネントの自発的起動と，Control loop を構成するコンポーネント間の制御機構を追加実装したプログラミングフレームワークを導入し，同フレームワーク上で Control loop を実装するためのイディオムを提案した．ここでは，JADE 拡張に関する関連フレームワークについて述べ，Control loop を実現するためのプログラミングフレームワークについては 7.4.3 節で関連研究を示す．

JADE を機能拡張したプログラミングフレームワークや JADE ビヘイビアの記述方法に言及した開発手法については，いくつかの既存研究がある．Griss らは [97] において，階層化されたステートマシンモデルを導入することで，エージェント，つまりシステム構成要素間の通信機能の実装を支援するイベント駆動型アーキテクチャ SmartAgent を提案している．

SmartAgent では中央に各イベントを処理するコントローラを設置し，コントローラが各ビヘイビアにイベントを配分するモデルを採用しているため，本機構を拡張することで，本論文で拡張すべき点として取り上げた自発的な起動メカニズムを実現できる可能性はある．しかしながら並列実行するビヘイビア間の連携については未解決であり，コンポーネント間の連携を実現できるものではない．Moraïtis ら [98] は，設計フェーズまでを支援するエージェント指向開発方法論 Gaia [99] による設計結果を JADE 上で実装する方法について提案し，Nikraz ら [100] はユースケースによるエージェントの同定から JADE を用いたエージェント実装までを包括する方法論を提案している．しかしながら，いずれも JADE が提供する既存のビヘイビアを用いた実装を推奨するにとどまり，本研究で Control loop を構成するコンポーネント間連携で必要となる，並行動作ビヘイビアの実装方法に関しては言及していない．

WADE（Workflows and Agents Development Environment）[101] は JADE のエージェントにワークフローに従ったタスクを実行させることを目的とした拡張である．しかしながら WADE もその目的から，ビヘイビアの実行順序を制御するにとどまり，本論文で扱う Control loop の並行動作を制御するための制御記述を支援するものではない．

Jadex [102] は，JADE フレームワーク上に BDI アーキテクチャ [103] モデルに従ったエージェントを構築可能とした JADE の 1 つの拡張アドオンである．BDI アーキテクチャとは，Belief–Desire–Intention の 3 つの心的モデルにより適切な行動を決定するプランニングを実行基盤としたエージェントアーキテクチャであり，Jadex ではビヘイビアを実装するのではなく，実行アクションであるプランを記述するスタイルを取る．このようなプランを記述する方法により Control loop を実装することも可能であるが，推論エンジンが単体である場合には，複数の Control loop が並行動作するシステムを構築するのは容易ではない．

7.3 変更のタイミング

本研究で導入したプログラミングフレームワークにおいては, 進化により追加された Control loop の実行時追加や，変更された Control loop の停止を実現するコマンドセットを提供している．このように，ソフトウェアシステムを動作させながら機能を追加，変更する手法は，**動的ソフトウェア進化**（dynamic software evolution）[104, 105] と呼ばれる領域で研究が進められている．本節では，まず，ソフトウェア進化に対する変更のタイミングの観点から，各関連研究について説明し，その後，変更のタイミングの観点からの本研究の位置付けについて議論する．

ソフトウェア進化に関しては，様々な観点からその特徴を分類づけることができるが，Buckley ら [19] は，ソフトウェア変更の一つの分類として，変更のタイミングを挙げている．

- **静的な (static) 変更：**ソースコード上でのソフトウェア変更であり，変更を反映する場合，再コンパイルが必要となるもの．コンパイル時の進化とも呼ばれる．
- **ロード時の (load-time) 変更：**ソフトウェアシステムに対して，構成要素をロードするタイミングで変更を反映させるもの
- **動的な (dynamic) 変更：**ソフトウェアシステムが実行中に変更を反映させるもの

この変更のタイミングは，開発環境やプログラミング言語に依存するものである．静的な変更は，システムが停止している間に変更を反映させるものものであり，その利点として，システム実行中の変化において扱わなければならない，状態遷移の問題や実行中スレッドの問題を解決する必要が無い点を挙げることができる．一方で，問題点として，システム停止によるサービス可用性の低下が挙げられる．

動的な変更は，コンポーネントのホットスワッピングなど，システムを停止することなく変更を反映させるものである．サービスの可用性を維持することができる一方で，状態遷移の問題や実行中スレッドの問題があり，真に動的な変更を実現するためには，現在もなお解決すべき課題が残されている．

ロード時の変更は，静的な変更と動的な変更の中間に位置する変更である．この種の変更では，Java におけるクラスローダ (Classloader) [106] などのような動的にソフトウェア構成要素をロード可能なメカニズムを利用して，ロード時に変更を実現するものである．

以降，ソフトウェア進化を実現するための要素技術を，コンパイル時の変更，ロード時の変更，実行時の変更の 3 つの分類に基づいて，実装技術の観点から論じる．

7.3.1　静的な変更

まず，静的な変更（コンパイル時の変更）は，システムを停止した上での変更であるため，任意の変更を施すことができる．コンパイル時の変更においては，進化とコードの再利用の観点から，継承を用いた変更が用いられる場合が多い．継承を用いた変更は，オーバーライドなどによりメソッドを再定義することができるが，基本的には古いクラスの拡張となる．また，多くの場合は，継承だけでは不十分であり，クラスの分離やクラスの統合が発生する．

継承を用いた変更においては，バージョン間の型の不整合を解決する必要がある．例えば親クラスを変更すると，継承先のクラスに影響を与えることとなる．Java 言語において親クラスに抽象メソッド（abstract 型メソッド）を追加すると，継承しているクラスにおいてそのメソッドを実装する必要が生じる．また，進化によりメソッドの振舞いが変わると，進化後に呼び出し側が想定する結果が得られなくなる可能性があり，また，進化後に当初想定していなかった状況でメソッドが呼び出される可能性がある．このような不整合の検出，解決の一つの

アプローチとして，契約による設計（Design by Contract）[107, 108] がある．

契約による設計とは，クラス間の連携において，互いに厳密な契約を介して協調するように設計するソフトウェアの構築手法 [108] であり，契約としては，一般に，メソッドに対する事前条件 (precondition) ，事後条件 (postcondition) や，メソッドによる処理の間常に成り立つ不変条件 (invariant) の 3 種類の条件が用いられる．メソッドを呼ぶ側が事前条件と不変条件を満たす義務を負い，呼ばれた側は事後条件と不変条件を義務として負うというのが契約による設計の基本概念である．

契約は振舞いの正確さを示すものであり，契約を用いることで，コンパイル時における振舞いという動的側面のチェックが可能となる．契約による設計は，提唱者である Meyer が設計したオブジェクト指向プログラミング言語 Eiffel [109] において実践が可能であるが，本研究で導入したプログラミングフレームワークでプログラミング言語として想定している Java 言語においても，JML（Java Modeling Language）[65] とその検証ツールである ESC/Java2 [110] を利用することで適用が可能である．

アスペクト指向プログラミング（AOP: Aspect Oriented Programming）[111] もソフトウェア進化に有効な技術の一つである．アスペクト指向プログラミングは，プログラム中から，そのプログラム自身をデータとして取り扱い, 計算の対象とするリフレクション（Reflection）[112] の研究成果を発展させた技法であり，基本的関心事に散在する横断的関心事（Crosscutting concerns）を分離することで，従来のオブジェクト指向プログラミングでは分散して記述されていた横断的関心事を独立したモジュールとして記述可能とする技法である．代表的なアスペクト指向プログラミング言語として，AspectJ [113] などがある．アスペクト指向プログラミングにおいては，モジュールとして分離記述された横断的関心事は，コンパイル時にジョインポイントと呼ばれるコード内に定義されたポイントに織り込まれる．この行為は Weaving と呼ばれている．

Walker ら [114] は，アスペクト指向プログラミングとソフトウェア変更との親和性について議論している．Walker らは，AspectJ と既存のオブジェクト指向言語とを用いた場合でのコード修正や変更容易性に関する実証実験を実施し，アスペクトのインタフェースが狭い（narrow），つまり関心事の分離が完全であり，各アスペクトの役割，与える影響が明確である場合には，アスペクト指向プログラミングは効果を発揮するという実験結果を示している．

ソフトウェア進化を考慮した場合，横断的関心事の追加，変更については，アスペクト指向技術適用の効果がある．しかし，その一方で，アスペクトの追加や変更を繰り返すことで，プログラム全体の可読性が低下するという問題点が指摘されている．また，アスペクトが複数存在する場合，それらが開発者の意図しない形で干渉する可能性がある．加えて，予期しない変更を考慮する場合，横断的関心事を記述するジョインポイントの場所が限定されているという

制約により，効果的にアスペクト指向技術を適用できない場合もある．従って，ソフトウェアシステム進化時にアスペクト指向技術を直接適用する場合は，事前に十分な影響分析と適用可能性の検討が必要になる．

アスペクト指向プログラミングについては，ロード時や実行時に Weaving 可能なフレームワークも存在することから，動的進化に対する要素技術としても期待されている．ただし，現状のロード時・実行時に Weaving 可能なフレームワークにおいては，コンパイル時と比較して，Weaving 可能なジョインポイントに制約がある．例えば，Seasar2 フレームワーク [115] においては，実行時の Weaving が可能であるが，フィールドへのアクセス時に対して Weaving ができないなどの制約がある．

7.3.2　ロード時の変更

ロード時の変更に対する実装技術としては，Java のクラスローダ（Classloader）[48] が良く知られている．Java のクラスローダを利用することで，システムが動作している状態においても新たなクラスファイルをロードすることが可能である．ただし，4 章でも述べたとおり，Java のクラスローダにおいては，通常はクラスの更新（再読み込み）を認めていない．クラスを再読み込みするには，一度アンロードした後にリロードする必要があるが，アンロードするには，ロードしたクラスローダを破棄しなければならない．しかしながら，通常のロードでは，システムクラスローダと呼ばれる親ローダがクラスをロードし，システムクラスローダは破棄をすることができないため，通常のロードにより読み込んだクラスはアンロードをすることができず，従って，更新できない．この問題に対しては，クラスパス上にクラスファイルを置かずに，固有のクラスローダから該当クラスファイルをロードするという手段と，リロードが可能なクラスローダ，つまり Oracle JVM とは異なったクラスローダを利用するという手段がある．後者に関しては，Tomcat[116] のクラスローダが該当する．本研究のプログラミングフレームワークも Java のクラスローダ，つまりロード時の変更技術を用いているが，その実装においては，前者のアプローチを用いている．

7.3.3　動的な変更

動的な進化には，実装環境において標準的に提供されている機能を用いるものと，仮想マシンなどを修正した独自の環境を用いるものの 2 種類がある．

JPDA（Java Platform Debugger Architecture）[117] は，Java プログラムの動的な変更を実現するデバッガであり，クラス定義の動的な変更を可能とする Hot Swap 機能により，実行時にクラスを変更することができる．これは，Oracle が提供する JDK に含まれている環境であり，

標準的な Java 実行環境での利用が可能であるが，一方で Hot Swap の制約により，変更が可能なのはメソッドのボディのみに限られるという制限がある．従って，メソッドやインスタンス変数の追加，削除などのクラススキーマの変更は実現できない．

一般に，動的進化を考えた場合，Java や C++ などの既存の主流なオブジェクト指向プログラミング言語にはクラスローダに関する制約が存在する．Java などのプログラミング環境においては，クラスをロード後に，メモリ内にオブジェクトが既に存在している状態でのクラスの書き換えは，一部の限られた側面を除いて通常は許可されていない．このクラスローダの制約に関しては，DVM [118]，Iguana/J [119] などのような仮想マシンの修正・拡張やクラスローダのメカニズムを変更 [120] することで対処している試みが見られるものの，本来の実行環境の動作を変更するため，互換性の問題が生じることとなる．

また，動的進化を実現するためには，新たなプログラムをシステムに追加して動作を開始させるとともに，動作しているプログラムの一部を差し替える必要がある．しかしながら，特に後者に関しては，アクティブなスレッド（active threads），状態転送（state transfer），不確実性（uncertainty）が動的進化を困難にする要因として指摘されている [8, 104]．アクティブなスレッドとは，実行状態がオブジェクトの状態として表現されるオブジェクト指向のプログラムにおいて，実行中のオブジェクトの修正が状態矛盾を引き起こすという問題である．状態転送は，エンティティ（オブジェクト）を古いものから新しいものへと交換した際に，新しいエンティティに対して古いエンティティの状態を移し替える必要があることを指している．不確実性は，実行中のシステムを変更することは，テストフェーズを持たないため，変更結果が予期しない振舞いをもたらすかどうかを検証できないことを示している．

Oriol は，これらの問題の本質を実装コード間の依存関係に帰着させ，実装コードが他のコードと依存関係をもつ場合として，以下の条件を定義している [104].

- 他クラスを継承している時
- 他オブジェクトを参照している時
- 他クラスのブロッキング・メソッドを呼び出している時
- コード間での同期制約が存在する場合

Oriol [104] はこのような依存関係を排除するために，サービス指向アーキテクチャのアプローチを利用したフレームワーク LuckyJ を提案している．LuckyJ においては，自律的な構成要素であるエンティティが存在し，サービスマネージャがエンティティ間の調整役となることで，サービス要求ごとにエンティティ間をバインドし，これにより，他オブジェクトの参照と，メソッド呼び出しに関する依存関係を排除しようとしている．

Oreizy ら [105] は，ソースコードレベルではなく，システムアーキテクチャの観点からの動

的進化法を提案している．Oreizy らは，コンポーネント間のインタラクションを実現するコネクタ（connectors）に着目し，コンポーネントとコネクタにより構成されるコンポーネントモデルである C2 [90] を用いた，コネクタを制御することによるコンポーネントの切り替えを実現している．

7.3.4　変更のタイミングの観点における本研究の位置づけ

最後に，本研究で導入した実装技術，つまりプログラミングフレームワークの位置づけを，変更のタイミングの観点から議論する．

本研究で導入したプログラミングフレームワークは，動的な変更を扱う技術に基づいたものではなく，Java のクラスローダ，つまりロード時の変更技術を用いることで動的な進化を実現している．従って，Java クラスローダという既存の確立された技術に基づいた動的進化の可能性を有するものであるが，一方で，活性化している Control loop の進化に対応できないという制約がある．4 章で構築した清掃ロボットに対しては，ごみ積載量管理機能の追加（進化 1）に関する Control loop の追加と，障害物回避機能の追加（進化 2）に関する Control loop の変更を清掃ロボットの動作中においても，プログラミングフレームワーク上でコマンド投入をすることにより，進化が実現できることを確認したが，特に進化 2 においては，目標物への到達に関する Control loop を一時停止させる必要がある．

動的な変更を扱った関連研究と比較すると，本研究は，コンフィギュレーションの配置に従った進化を実現するという点では，Oreizy らの研究と類似している．また，自律的な構成要素によりシステムを構築するという観点で，Oriol の研究と類似している．一方で，Oreizy らの手法とは，進化の影響を考慮した構成要素単位を扱っているという点で異なり，また，Oriol らのフレームワークとは，サービスマネージャーのような集中的な調整役を用いていないという点で異なる．また，いずれの手法も，コンポーネントやエンティティといったシステム構成要素の決定法については言及していないが，本研究においては，ゴールモデルと整形プロセスを用いることで，構成要素の同定手段も扱い，これにより妥当なコンフィギュレーションを決定する手段を提供している点が異なる．

7.4　自己適応システム

自己適応システム（Self-adaptive systems）[121, 35, 122] とは，環境の変化を検知することで状況を判断し，自発的に振舞いを変化させることで，状況変化に対応できるシステムであり，状況変化への対応，つまり適応のための振舞いや構成変更を実現する意思決定メカニズムが組み込まれているシステムを指す．そのような背景から，本研究でシステム構成要素として

利用した Control loop を自己適応システムの振舞いモデルとして位置づける研究も少なくない [123, 124]．自己構成（Self-configuration），自己修復（Self-healing）などのソフトウェアの自律化を掲げるオートノミックコンピューティング（Autonomic computing）[36, 125] の研究においても，制御ループと同様の概念が MAPE (Monitor, Analyze, Plan, Execute) loop として用いられている．

本節では以降，自己適応システムに関する研究動向を，本研究に関連する，要求記述，アーキテクチャ，フレームワーク，検証技術の観点から述べる．

7.4.1 要求記述に関する議論

本研究では，自己適応システムの進化を支援する開発プロセスを実現するために，進化の要因となる要求の変化が分析・同定可能である要求記述に着目したが，自己適応システムのような動的な適応を扱うシステムにおいては，適応のタイミングと適応後の動作を決定するために，システム自身が要求の達成状況を管理する必要がある．従って，自己適応システムにおいても，本研究のアプローチ同様に，要求記述はシステムにおいて核となるモデルであると言え，自己適応システムの研究領域では要求分析に関するものや要求記述を扱った研究が少なくない．

Fickas ら [126] は，環境の変化に対してもシステムを安定稼働させるためには要求の監視が重要であるとの主張から，システム動作中の要求監視法を提案している．Salifu ら [127] は，環境の変化を検出するために必要なモニタリングに関する要求と，変化に適応して要求を満足するために必要な振舞い切替（スイッチング）要求に着目し，これらを要求記述上で仕様化する手段を提案している．この研究では，適応時の振舞いに対する要求の記述に，システム開発の対象となる領域，システム，システムに対する要求とこれらの間の関連を記述する問題フレーム（Problem Frames）[128] を用いている．Cheng らは，不確実な状況を記述するために RELAX 言語 [129] を提案している．この RELAX 言語は，要求記述に用いる自然言語に対して，不確かさを記述する固有のオペレータを導入することで時制を緩める（relax）ことを目的としたものであり，従来の「. . . すべきである（shall）」の記述に，「いずれは（eventually）」，「なるべく早く（as early as possible）」などを導入することで，適応に関する要求記述を可能にしている．

自己適応システムの要求記述モデルとして，本研究で用いた KAOS や，i* [130] などのゴールモデルを用いる研究も少なくない．Cheng ら [131] は，彼女らが提案した RELAX 言語を用いて自己適応システムに対するゴールモデルの構築法を提案している．Lapouchnian ら [132] も自己適応システムに対する要求をゴールモデル上に記述し，各状況においてどの振舞いが可

能であるかの決定と最適な振舞いの選出にこのゴールモデルを利用している．Landtsheer ら [133] は，制御システムを対象としたイベント駆動型の仕様生成手法を提案している．この手法は，ゴールモデルから表型記述言語に基づいた仕様を生成するものである．Wang ら [134] は，レガシーなソフトウェアシステムに高い可変性（variability）を付与することを目的として，ゴールモデルを用いたシステムの監視・再構成手法を提案している．これらのアプローチはいずれも，ゴールモデルをシステムの振舞い記述や振舞いの決定のために用いたものである．Chen ら [135] は，Web アプリケーションの生存性（survivability）を保証することを目的として，ゴールモデルに基づいた自己適応手段を提案している．この手法では，ゴールモデル上の各ゴールに付与された引機能要求に対する *relative parent values* と呼ばれる貢献度をもとに，各状況においてどのゴールを優先しすべきかを決定し，コンフィギュレーションを決定する．この手法でも Control loop の概念を採用しているが，コンフィギュレーションの変更タイミングを決定するために利用するという，従来のシステム制御としての利用法であり，本研究のようなコンフィギュレーション決定のためのものではない．

本論文で導入した開発プロセスはソフトウェアシステムの進化を対象としたものであるが，ゴールモデルから実装コードへの開発プロセスが，自己適応システムの実現メカニズムに対応すると考えることもできる．例えば，要求や環境の変化は，ゴールモデルの変更に対応し，変化がシステム構成の変更，つまりコンフィギュレーションの変更を触発し，実装コンポーネントを切り替えることにより，環境変化に適応すると考えると，両者のプロセスは類似している．

7.4.2　アーキテクチャに関する議論

次に，自己適応システムのアーキテクチャに関する研究動向について述べる．自己適応システムやオートノミックコンピューティング [36, 125, 37] を実現するためのアーキテクチャモデルとしては，複数のコンポーネントを接続したモデルが有効であると考えられている．

Kramer と Magee は，自身の振舞いを管理可能なシステムが取るべきアーキテクチャとして，階層構造からなる 3 層アーキテクチャモデル [5] を提案している．このアーキテクチャモデルは，ロボティクスの分野で提唱された Gat の 3 層モデル [136] を応用したものであり，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（Goal management layer）と，振舞いを実現する最下層のコンポーネント制御層（Component control layer），さらにこれらの 2 層の中間に位置し，現在の状況と目的の達成状況からコンポーネントの構成切り替えを決定する変更管理層（Change management layer）の 3 層により構成され，各階層がそれぞれの抽象度に応じた適応を扱うことが特徴である．この 3 層アーキテクチャモデルは，現在多くの自己適応システムに関する研究において基本アーキテクチャモデルとして用いられている．Taylor ら

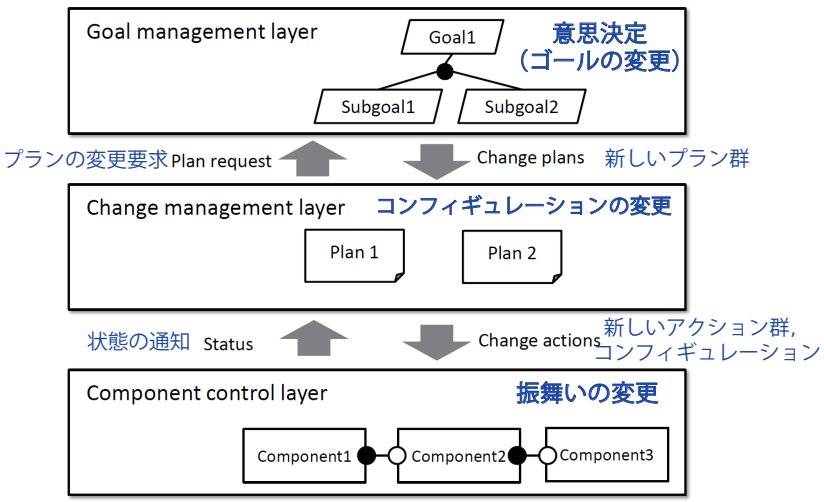

図 7.1. 3 層アーキテクチャモデル [5]

[35, 137] も，動的な振舞いの切り替えを実現するために，前節で述べたコンポーネントの動的切換えを実現するためのアーキテクチャ [105] を自己適応システムのアーキテクチャモデルに発展させている．彼らが提案する RAS アーキテクチャモデル [137] は，コンポーネントとコネクタにより構成され，抽象化に応じた複数の層を持つモデルである．RAS アーキテクチャモデルにおいては，コネクタが層間の通信を実現する要素として用いられている．

これらの研究は，動的な適応を扱うためのアーキテクチャを提案するものであるが，設計・実装すべきコンポーネントの決定やコンポーネント間の接続関係の決定関して言及したものではない．従って，開発者はこれらのモデルを採用するとしても，具体的なコンフィギュレーションや，目的管理とコンフィギュレーションとの対応関係を整合を取りながら検討する必要がある．特に，複数のゴールやアクションをもつ自己適応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，コンポーネント間で発生し得る競合を考慮しながら，接続関係を決定することは通常容易ではない．3 層アーキテクチャモデルに関しては，実現に向けた関連研究が文献 [138, 139] などいくつか挙げられるものの，各層間での無矛盾性の保証などの研究課題も多く，3 層アーキテクチャを実現するための実装フレームワークは現段階では確立されていない．

本研究で導入するコンフィギュレーション決定法は，自己適応システムの構築においても，設計・実装すべきコンポーネントとコンポーネント間の接続関係を決定するために利用することができる．本研究で生成されるコンフィギュレーションは，例えば Kramer らの 3 層アーキテクチャモデルにおいては，最下層のコンポーネント制御層のコンポーネント間構成として利用することができる．提案手法が生成するコンフィギュレーションでは，Analyze & Decide

タイプのコンポーネントが上位階層，例えば 3 層アーキテクチャモデルにおける変更管理層やゴール管理層との接続の役割を担うことになる．加えて，ゴールモデルの階層構造を利用してコンポーネント間の接続関係を決定するため，生成されるコンフィギュレーションは，目的との対応関係も考慮したものであることが期待できる．

自己適応システムのアーキテクチャモデルとしては，3 層アーキテクチャモデル，RAS アーキテクチャモデルの他に，Garlan らの Rainbow [140] がある．Rainbow はセンサ・イフェクタにより構成される System-layer，適応を実現する Architecture-layer と，これらを仲介する Translation infrastructure により構成されるアーキテクチャモデルを持つ．Rainbow は，アーキテクチャモデルだけでなく実装フレームワークも提供し，また Control loop をモデルとした実装が可能であるが，1 つの Control loop がシステム全体を管理するという集中的な制御機構を採用しているため，複数の適応を扱う場合にはその記述は急激に複雑化するという特徴がある．

7.4.3　フレームワークに関する議論

自己適応システムの実装フレームワークとしては，前節で述べた Rainbow の他に，StarMX [141] や Adaptive Server Framework (ASF) [142]，JADE [143, 144, 145] [*1]などが提案されている．StarMX は，Java 言語とポリシー記述言語で動作を記述する実装フレームワークである．StarMX の特徴としては，Java のアプリケーション管理フレームワークである Java Management Extensions (JMX) [146] を利用して，センサやイフェクタ，アプリケーションを監視する点にある．しかしながら，StarMX は振舞いの制御を基本的にはポリシー言語で記述するため，複数の適応を扱う場合にポリシー記述部が急激に複雑化する可能性がある．ASF も JMX を用いてアプリケーションを監視するが，Control loop モデルに従った実装が可能である．ただし，Rainbow 同様に，1 つの Control loop による集中的な制御機構であるため，複数の適応を扱う場合は，記述が複雑化する．JADE は，自己修復機能を有するフレームワークであり，コンポーネントの状態やコンポーネントの異常終了を監視することで，事前に定められたポリシーに従って，システムを安定状態に戻すようにコンポーネントの構成を切り替えることが可能なコンポーネントベースのフレームワークである．JADE はサーバの安定運用を目的としたフレームワークであり，J2EE 上に実装されている．ただし，サーバの安定運用を目的としていることから，コンポーネントの切り替えは，クラスタ構成されている同一クラスのコンポーネントへの切り替えを想定したものである．

本研究同様に，エージェントプラットフォームを利用した自己適応システム実装フレーム

[*1] 本研究で利用するエージェントプラットフォームの JADE と同名であるが，関連性は無い．

ワークの提案もある．Morandini ら [147] は，Belief–Desire–Intention の 3 つの心的モデルにより適切な行動を決定する BDI モデル [103] を実装基盤とした Jadex [102] を自己適応システムの実装プラットフォームとして利用している．Jadex ではビヘイビアを実装するのではなく，実行アクションであるプランを記述するスタイルを取るが，プランを起動するためのゴールとの関係を定義する必要があり，また，ゴールの状態により動作するプランが決定するため，並行プロセスの記述は困難である．

本研究で導入したフレームワークは，複数の Control loop を扱うことができるという点から，自己適応システムのフレームワークの基礎部分として利用が可能であると考える．本フレームワークを自己適応システム実装に利用するためには，要求を管理するメカニズムと，意思決定メカニズム，意思決定に基づいた自動構成変更メカニズムの追加が必要である．

7.4.4 動作検証に関する議論

最後に，自己適応システムにおいては，ソフトウェア進化と同様に，変更後の検証が必要である．従って，自己適応システムの研究においては，システムが適応後に期待する振舞いを実行することを保証するための検証技術が大きな研究課題として認識されている．この一つの流れとして，自己適応システムを対象としたモデル検査法に関する研究がある．Zhang と Cheng [148, 149] は，動的に振舞いを変えるシステムにおける振舞いの検証方法としてモデル駆動型の検証法を提案している．Zhang らは，適応に関する振舞いと適応に関連しない振舞いとを分離したモデル検査法を用い，時相論理式で記述された要求記述と，実装との間の無矛盾性を保証しようとしている．本研究においては，システムの振舞いを Control loop 単位で分割しているため，Zhang らの分離によるモデル検査が進化時のテスティングに応用できる可能性がある．

また，Schaefer ら [150] は，適応に関する振舞いと機能実現に関する振舞いとを分離し，スライシングによる状態数の抑制手段を検討している．スライシング [151] とは，プログラム等のモデルから，関心のある特定の部分だけを抽出し，簡約化する手法であり，近年，UML 図などの設計モデルに対してもモデルの抽象化や関心部の抽出のために適用が検討されている．例えば，Lano ら [152] は，モデル変換時に着目する部分だけを変換する，クラス図とステートチャート図上でのスライシング手法を提案している．本研究においては，プロセス変数を意識して Control loop を分割しているため，プロセス変数やその関連要素に着目したスライシングにより，効果的な検証を実施できる可能性がある．

7.5　まとめ

本章では，本研究で用いた各要素技術に対する関連研究として，特に，要求記述と設計モデルとの連携に関する研究と，アーキテクチャモデル・実装フレームワークに関する関連研究について述べた．また，ソフトウェア進化に関する実装技術の関連研究として，ソフトウェアシステムの変更のタイミングの観点から関連する各研究の動向について述べ，提案する開発手法の拡張分野として考えられる動的進化に対する提案手法の位置づけについて論じた．また，もう一つの拡張分野として，自発的に振舞いを制御することで環境の変化に適応する，自己適応システムの研究動向について述べた．自己適応システムにおいては Control loop をシステムの構成要素として位置付けている研究もあることから，本論文で提案する開発手法との親和性も高いと考えられる．

第 8 章

結論

8.1 本研究の成果と得られた知見

本論文では，進化を考慮したソフトウェアシステムの開発プロセスについて述べた．ソフトウェアシステムの進化を要求分析段階から捉えるために，本研究では，要求の構造化が可能であるゴール指向要求分析法に着目し，進化の影響を限定化するために Control loop をシステム構成要素とした，ゴールモデルからシステム構成を決定するためのゴールモデル成形法を導入した．また，提案する開発プロセスを支援するツール群として，システムの構成を表現したコンフィギュレーションの生成や，進化に際してシステムを設計・実装するための各種情報を生成するコンパイラ gocc と，Control loop の実装を支援する 1 つのプログラミングフレームワークを導入し，これらにより構成されるソフトウェアシステム開発プロセス GCLD を提案した．

また，本研究で提案する開発プロセスの有効性を評価するために，実用化されているモデリングツール k-tool と制御システムを想定した清掃ロボットを題材としたソフトウェアシステム進化開発実験を実施し，実験結果をもとに提案手法の有効性を評価した．

以下に，本研究の成果と得られた知見をまとめる．

8.1.1 進化を考慮したソフトウェアシステム開発プロセス GCLD

本研究では，進化を考慮したソフトウェアシステム開発プロセスを実現するために，要求が構造化して記述されるゴールモデルを開発プロセスの中心となるモデルと定義した．提案する開発プロセス GCLD においては，ゴールモデルから Control loop を同定するための整形プロセスを導入している．整形プロセスの大きな目的は，ゴールモデル内において主要な役割を担う主要ゴールを同定する点と，同定した主要ゴールに対して，Control loop パターンを導入す

ることによる Control loop のゴールモデル上での配備にある．整形プロセスにおいては，ゴールモデル中のゴール，エンティティ間関係を特に利用している．主要ゴール同定時には，主要ゴールの候補となり得るゴールをエンティティ間の関係を利用して選定し，Control loop ごとに関連エンティティを分離する．また，主要ゴール同定後は，Control loop ごとに割り当てられた各エンティティに対して，プロセス変数としての責務を割当てている．

整形プロセスにおいては，2 種類のパターンを導入した．一つは，Control loop 間で発生し得る競合を検出するための，Entity-conflict パターンであり，もう一方は，Control loop 内での責務を同定するための Control loop パターンである．前者は，モデル上に望ましくない状態（競合状態）が存在するかどうかを検査するためのパターンであり，後者は，期待する状態までモデリングを推進するためのパターンである．このように本研究では，ゴールモデルに対して，期待する状態，つまりシステム構成が決定可能な状態に到達するための整形プロセスと，整形を支援するパターンを導入したが，記述の自由度が高くモデリングが困難であると指摘されているゴールモデルに対して，一定の水準を要求するモデリングの支援に，本手法のようなプロセスおよびパターンの導入は効果があるということが，評価実験からも確認できた．

また，本研究においては，整形したゴールモデルの構造を利用した，コンフィギュレーションの決定法を導入した．コンフィギュレーションとはシステムのアーキテクチャを示すものであり，本研究では，ゴールモデル上の一部のゴールとコンフィギュレーション上のコンポーネントとを対応付けたモデル変換により，コンフィギュレーションの生成を実現している．このようなコンフィギュレーションの決定法においては，ソフトウェアシステムの進化を考慮した場合にも，要求記述上の変更箇所を反映したコンフィギュレーションの獲得が可能であることを，評価実験からも確認することができた．

このようなゴールモデルを利用した開発プロセスにおいては，ゴールモデルへの分析結果により，適用範囲が判断される．整形後のゴールモデル上で Control loop が少数しか抽出されない場合は，提案する開発プロセスを適用する効果が薄くなると考えられる．システムにおける多くのアクションが，同一の入力変数に依存するようなシステムなどは，このようなゴールモデルとなる可能性があり，例えば，数値計算や，会計処理アプリケーションなどはこのクラスに属すると考えられる．一方で，整形後のゴールモデルにおいて複数の Control loop が構築されている場合は，本開発プロセスの効果が期待できる．このようなシステムは，多種にわたる入力変数を扱うシステムであり，また，入力変数ごとに取り得るアクションが異なるシステムであり，例えば豊富な GUI を持つシステムや Web アプリケーション，ユビキタスアプリケーションなどが該当する．

8.1.2 ゴールモデルコンパイラ gocc の構築

本研究では，提案する開発プロセス GCLD を支援するためのツールも導入した．その一つは，ゴールモデルコンパイラ gocc である．gocc は，整形されたゴールモデルから，システム構成を表現するコンフィギュレーションを生成するが，その過程で利用する情報を，ソフトウェア進化に必要な各種情報として生成するコンパイラである．

まず，ゴールモデルを解析することで，本研究で導入する 2 種類の競合パターン，つまり Entity-conflict パターンと Goal-conflict パターンに一致する箇所を検出し，その結果を競合リストとして生成する．コンフィギュレーションを生成した後には，Collect タイプなど，Control loop 内での責務を同定した結果をもとに，クラステンプレートを生成する．また，進化前のゴールモデルも入力情報とし，ゴールモデルと生成されるコンフィギュレーションにおいてそれぞれ差分を検出することで，進化が及ぼすゴールモデル上およびコンフィギュレーション上の影響を検出することができる．

このように，gocc を利用することで，ソフトウェア進化に必要な様々な情報を，ゴールモデルから抽出できることが確認できた．ただし，現段階で抽出できる多くの情報は，システムの構造的な側面に対する情報であることから，更なる Control loop の設計・実装支援には，システムの振舞いに関する情報の抽出法の検討が有益であると考える．

8.1.3 Control loop 実装のためのプログラミングフレームワークの構築

本研究では，gocc とともに，本研究で提案する開発プロセス GCLD を支援するプログラミングフレームワークを実装した．このプログラミングフレームワークは，Control loop の実装を支援し，また構築された Control loop 群をシステムとして動作させるプラットフォームとしての役割を持つ．

実装支援という観点での特徴としては，Control loop を 3 種類のコンポーネントにより構築し，複数の Control loop を並行動作させる点が挙げられる．実装に関しては，本研究ではコンポーネント実装時のイディオムも導入し，コンポーネントの構成により，Control loop の実現が可能であることが確認できた．

一方で，プラットフォームとしての観点からは，開発者がシステム動作中に，Control loop を制御，デプロイ可能なコマンドセットを提供する点が特徴として挙げられる．このコマンドセットを利用することで，一部の動的進化が実現可能であることも確認できた．本プログラミングフレームワークを利用することで，コンポーネント連携による Control loop の構築が可能であり，Control loop を構成要素とするような自己適応システムのプログラミングフレーム

ワークとしての応用も期待できる．

ただし，提案する開発プロセスが想定するシステム構成においては複数の Control loop が存在するため，複数 Control loop の並行動作や，複数コンポーネントの並行動作によるオーバーヘッドが存在するのは事実である．従って，実用を考えた場合には，性能面に関する検討も今後必要と思われる．

8.2　今後の展望

本研究では，ソフトウェア進化を扱うための包括的な開発プロセスを検討し，開発を支援するツール群を構築したが，確実なソフトウェア進化の実現に向けては，解決すべき課題はまだ多い．以下に，本研究の今後の展望について述べる．

8.2.1　振舞いの側面からの設計・実装支援

本研究においては，進化を扱う開発プロセスにおける開発者支援として，コンフィギュレーションの生成やクラステンプレート生成などの自動化を実現したが，その多くは，システムの構造的な側面に対しての支援である．一方で，システムの振舞い的な側面に対する支援については，ゴールモデルの差分検出や競合リストの生成など，その範囲が限定的である．従って，ソフトウェアシステムの進化を十分に扱うには，今後はシステムの振舞い的側面に対する支援の検討が必要と考える．

システムの振舞い的側面に対する設計・実装の支援を考えた場合，追跡可能性の観点からも，進化に対する要求が記述されるゴールモデルを活用することが望ましい．ゴールモデルの情報を活用する場合，その活用手段としては，大きく 2 つのアプローチが考えられる．一つは，本研究における Control loop 抽出やコンフィギュレーションの決定，競合検出にも利用したように，ゴールモデルの構造，つまり要素間関係の情報を利用するアプローチであり，もう一方は，各要素の属性情報を利用するアプローチである．

まず，ゴールモデルの構造を利用するアプローチとしては，更なるゴールモデルの整形が考えられる．システムの機能的な振舞いは Control loop における処理部，つまり本研究における Act タイプコンポーネントに記述されるが，本研究では，ゴールモデル上の Act タイプゴールが，コンフィギュレーションや実装コード上の Act タイプコンポーネントに該当する．従って，Act タイプコンポーネントの振舞いをゴールモデルの構造を利用して詳細化することも振舞い定義の厳密化に寄与すると考えられる．Act タイプゴールをさらに，ゴール達成のために経由しなければならない状態群への分解を促す Milestone-driven パターンや Case-driven パターンなどで詳細化することで，システムの振舞いをより特定することができるであろう．

しかしながら，やはり振舞いの記述を考えた場合には，構造によるアプローチには限界がある．よって，振舞いの設計・実装支援には，各要素の属性情報利用のアプローチは必須のものであるといえる．このアプローチにおいては，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 [59] に基づいたシステム進化と，仕様から実装コードへの詳細化という観点からの，形式手法（formal methods）[62] による段階的詳細化の適用が考えられる．

形式手法は，集合論や論理学など離散数学上の理論を用いてシステムの仕様を記述する開発手法であり，UML [153] などの図形表現と比較して記述に曖昧さが無く，システムの厳密なモデル化が可能な手法である．また，記述言語の多くが段階的詳細化によるソフトウェア開発を可能としている点も形式手法の特徴である．段階的詳細化とは，その妥当性が確認しやすい抽象的な記述をもとに徐々に記述の内容を具体的なプログラムによる実行形式に近い記述に変換する開発スタイルであり，形式仕様言語を用いることで，詳細化の各段階で正しさを検証しながらモデルの具体化を進め，そのプロセスで仕様上のバグ混入を排除することができる．例えば，VDM++[154, 64] はオブジェクト指向設計に基づいた仕様記述ができる一方で，拡張前の VDM-SL[155, 63] 同様に，どのように機能が実現されるかを記述する明示的記述（explicit specification）と，何が機能として要求されるかを記述する非明示的記述（implicit specification）の 2 つの記述スタイルを持つ段階的詳細化が可能な形式仕様言語である．

具体的には，ゴールモデルの各ゴールに対する仕様を形式的に記述し，それをコンポーネントの振舞いに関連付け，段階的詳細化を用いて徐々にコードへと近づけていく開発プロセスが考えられる．ゴールモデルからのモデル変換による実装コードへの到達には，各工程において情報の追加が必要になるが，その段階的詳細化の過程でゴールの仕様を制約として用いることで，要求分析結果に矛盾しないシステムの進化が期待できる．

従って，今後はまず，より確実な振舞いの進化を実現するために，形式手法を導入したソフトウェアシステムの進化法について検討を進めたい．

8.2.2 競合解決とテスティング

6 章で議論したように，本研究ではシステム構成要素を Control loop とすることにより，ソフトウェアシステムの進化時における影響範囲を限定化している．しかしながら，その一方で，Control loop が並行に動作する場合の競合解決やテスティングに対する支援が現状は限られている．

Control loop 間で生じる競合の厳密な検出や動作の検証には，並行プロセスの検証技術として着目されている SPIN [156] などを用いたモデル検査が有効である．しかしながら，モデル検査においては，検査を適用する箇所の特定と，設計モデルや実装コードとの整合性維持が適

用時の一つの難しさとして知られている．

そこで，本研究では，形式仕様を用いた 2 段階の検証が有効ではないかと考えている．例えば，VDM++ では仕様を実行することによるテストが可能であるため，形式仕様上でのテストと，仕様テストで検証できない箇所に対してのモデル検査の実施という 2 段階の検証が考えられる．特に本研究では，Control loop をシステムの構成要素として想定するため，Control loop の振舞いを考慮した，効率的な検証手法を検討する必要がある．併せて，競合を検出した場合の，競合対応法に対する設計・実装指針も示したい．

8.2.3　動的進化・自己適応システムへの適用

本研究で想定するシステムは複数の Control loop により構成されるシステムであり，ソフトウェアの進化を Control loop で扱う場合，各 Control loop の制御を独立化させることにより，システム実行時における追加や削除，つまり動的進化が可能となる．また，7 章で論じたように，自発的に振舞いを変化させることで状況変化に対応できる自己適応システムの実現にも，複数の Control loop によりシステムを構築するというアプローチと，ゴールモデルとコンポーネントが関連付けられているという本手法の特性は有益であると考えられる．

従って今後は，これらの分野に対しても，本研究の成果を応用させたい．動的進化に対しては，まず，進化の種類や程度，想定環境により，動的進化が可能な場合とそうでない場合があると考えられるため，適用範囲の同定を進める必要があろう．また，動的進化の前後におけるシステムの動作を保証する必要もある．プログラミングフレームワークに関しては，適用範囲を拡大する機能拡張とともに，安全な動的進化を実現するメカニズムの導入が必要となる．

図 8.1 は，動的進化の立場から，本研究で導入した開発プロセスの各ステップと生成モデルを示したものである．4 章の清掃ロボットシミュレータ上での実験では，図中の実装コードを動的進化プロセス，つまりコマンド列により，進化対象システムに動的に組み込んだが，今後さらに上位のステップとの関係を形式化することで，ゴールモデルの変更による動的進化の実現が期待できると考えている．

自己適応システムへの応用に関しては，本研究で提案した開発プロセスにおける各活動の自動化が鍵になると考えている．まず，自己適応システムにおいては，ゴールモデルをシステム自身が管理・把握することとなるため，ゴールモデルをシステムが理解・管理できるような形式化が必要となる．また，要求や環境の変化にも対応可能とするには，ゴールモデルの変更から，システム構成の変更箇所，つまりコンフィギュレーションの差分を発見し，構成を切り替えることによる振舞いの実現も自動化する必要がある．

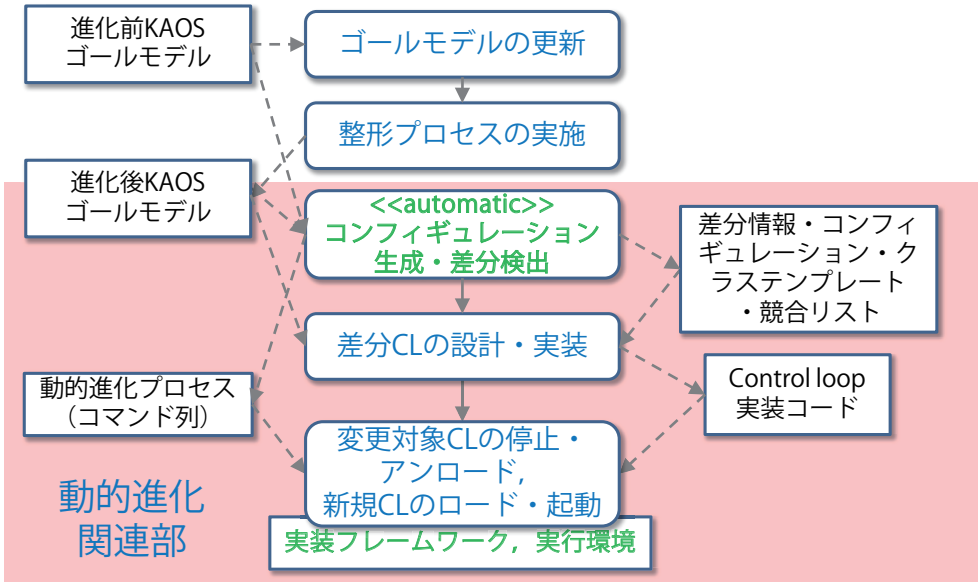

図 8.1. 動的進化の立場から見た本研究で導入する開発プロセス

8.3 まとめ

ソフトウェアシステムの活躍する場面が広がり，長寿化するソフトウェアシステムが増える一方で，要求や環境の変化への対応に代表される，ソフトウェア進化の機会は増加傾向にある．特に，Parnas [16] が “Software aging” と呼び，Lehman[15] が “Laws of software evolution” の一つとして定義しているように，ソフトウェアシステムの品質は徐々に低下するという性質を持ち，ソフトウェアシステムの進化は，ソフトウェアシステムのライフサイクルにおいて必須のアクティビティと考えなければならない．

本研究では，ソフトウェアの進化を考慮した効果的なソフトウェアシステム開発手法として，ゴール指向要求記述を利用した Control loop の同定法を提案した．この Control loop の同定のために，本研究では，ゴール指向要求記述上での整形プロセスを定義し，整形プロセスを経たゴールモデルに対してシステム構成，つまりコンフィギュレーションの生成法を提案した．この生成法は自動化が可能であることから，コンフィギュレーション生成と，得られたコンフィギュレーションを用いてコンポーネントを実装するための情報を生成する支援ツールを導入した．また，実装フレームワークを提案し，その有効性と実現可能性をシステム構築実験を通じて評価した．

本研究では，支援ツールを含んだ開発プロセスと，実装のためのプログラミングフレーム

ワークとを統合することで，システム開発の上流工程から実装までの一つの開発環境を提供した．このような開発環境の提供は，要求分析結果に合致したソフトウェアシステム進化の実現を支援するものであり，本研究により，要求変化により生じる影響を分析でき，変更に対する影響が限定的であるシステム進化法が提供されると考える．

確実なシステム進化の実現に関しては，実装モデルの更なる連携手段 [157] や，進化後のシステムに対する効率的な回帰テストの実現方法 [158, 159]，さらには進化時の矛盾の無い要求記述の更新手段 [160] など，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システムの構築に対する一つの有効な手段となれば幸いである．

謝辞

本論文をまとめるにあたり，終始温かい御指導を賜りました 早稲田大学 深澤良彰 教授に深甚なる感謝の意を捧げます．また，多大なる御指導と御鞭撻を賜りました中島達夫 教授と，鷲崎弘宜 准教授に心より感謝申し上げます．

また，在学時代から今日まで終始あたたかいご指導を賜りました，国立情報学研究所／東京大学 本位田真一 教授に厚く御礼申し上げます．修士課程在籍時には研究方法の基礎から論文の執筆に至るまで丁寧にご指導いただき，研究室卒業後も貴重な研究活動の機会を与えて下さった御蔭で，今日まで有意義に研究を進めることができました．本当にありがとうございました．また，日常の研究生活において数々の親切なご教示を賜るとともに，温かく見守ってくださいました電気通信大学 大須賀昭彦 教授および田原 康之 准教授に謝意を表します．講座内で学生指導を通じて共に研究を進めさせて頂きました 清 雄一 助教，中山健 氏，研究室のさまざまな事務を快く引き受けて下さいました 利 百合子 女史，日々共に研究を進めたシステム設計基礎学講座（大須賀・田原研究室）の学生の皆様にも感謝の意を表します．研究・学生指導に対しまして，日頃よりご指導を頂いております電気通信大学 情報システム学研究科の先生方にも感謝申し上げます．

さらに，本研究について多くの有益な御助言を頂きました国立情報学研究所 吉岡信和 准教授，田口研治 特任教授（現在，産業技術総合研究所 招聘研究員）をはじめとした先生方，修士課程および博士課程在籍時にお世話になりました東京大学 情報理工学系研究科 創造情報学専攻の先生方，当時博士課程の学生の立場からご指導くださった石川冬樹 博士（現在，国立情報学研究所 准教授），松崎和賢 博士， Eric Platon 博士，苅部卓哉氏，土肥拓生氏，鄭顕志 博士（現在，国立情報学研究所 助教），末永俊一郎 博士，同期の 丹羽智史氏，Truong Khanh Quan 氏，石黒真氏，神谷友輔氏をはじめとする東京大学 本位田研究室／早稲田大学 深澤研究室／電気通信大学 大須賀研究室の皆様方や諸先輩方，そして本研究に限らずあらゆる分野にわたって御協力下さった学生の皆様にも感謝の意を表します．

また，Lero (Irish Software Engineering Research Centre)/ Open University の Bashar Nuesibeh 教授，Imperial College London の Jeff Kramer 教授，Lero の Kevin Ryan 名誉教授，University

of Washington の David Notkin 教授，Open University の Yijun Yu 専任講師からも，個別の議論を通じて本研究を進めるにあたっての有益なご示唆を頂きました．ここに深く感謝の意を表します．

鹿島建設株式会社在籍時に，社会人学生として研究の機会を下さいました鹿島建設株式会社 IT ソリューション部 松田元男 前部長（現在，カジマアイシーティ社長），渡邉克彦 部長にも心から御礼申し上げます．また，共に建設業における情報技術の実用可能性を検討して下さいました，狩野茂氏（現在，カジマアイシーティ 取締役），松川剛一氏，伊藤一宏氏，天野和洋氏，河村 一グループ長，伊藤智尋氏，森田順也氏をはじめとする IT ソリューション部員の皆様に感謝の意を表します．

また，大阪大学 基礎工学部 情報工学科在籍時に，多大な御指導と御鞭撻を賜りました，北橋忠宏 教授，馬場口登 助教授（現在，大阪大学 教授），角所考 助手（現在，関西学院大学 教授），大原剛三 博士（現在，青山学院大学 准教授），桂田浩一 博士（現在，豊橋技術科学大学 准教授）をはじめとする北橋研究室の皆様方，基礎工学部 情報工学科の先生方に深く感謝致します．

最後に，日々の研究活動を心身両面に渡って支えてくれた妻 美樹と娘 玲南，杏南に心から感謝します．

参考文献

[1] Mary Shaw. Beyond objects: a software design paradigm based on process control. *SIGSOFT Software Engineering Notes*, Vol. 20, pp. 27–38, January 1995.

[2] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, Vol. 1, No. 2, pp. 223–259, 2006.

[3] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proc. of the 5th European Software Engineering Conference (ESEC '95)*, pp. 137–153, Sitges, Spain, September 1995. Springer-Verlag.

[4] Dan Hirsch, Jeff Kramer, Jeff Magee, and Sebastián Uchitel. Modes for software architectures. In *EWSA*, pp. 113–126. LNCS, 2006.

[5] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. *Future of Software Engineering (FOSE '07)*, pp. 259–268, 2007.

[6] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley Professional, 2003.

[7] Ian Sommerville. *Software engineering 8*. Addison-Wesley, 2007.

[8] Peter Ebraert, Yves Vandewoude, Theo D'Hondt, and Yolande Berbers. Pitfalls in unanticipated dynamic software evolution. In *Proc. of the Workshop on Reflection, AOP and Meta-Data for Software Evolution in conjunction with the 18th European Conference on Object-Oriented Programming (ECOOP '05)*, pp. 41–49, 2005.

[9] Alain April and Alain Abran. *Software Maintenance Management: Evaluation and Continuous Improvement*. IEEE CS, 2008.

[10] Meir M. Lehman. Programs, life cycles and laws of software evolution. In *Proc. of IEEE (Special Issue on Software Engineering)*, Vol. 68, pp. 1060–1076. IEEE, 1980.

[11] M. M. Lehman. Software engineering, the software process and their support. *Software Engineering Journal - Special issue on software process and its support*, Vol. 6, pp. 243–258, September 1991.

[12] M. M Lehman. Laws of program evolution - rules and tools for programming management. In *Proc. of Infotech State of the Art Conference, Why Software Projects Fail?*, 11, pp. 1–25, 1978.

[13] M. M. Lehman. Laws of software evolution revisited. In *Proc. of the 5th European Workshop on Software Process Technology (EWSPT '96)*, EWSPT '96, pp. 108–124, London, UK, 1996. Springer-Verlag.

[14] M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. Metrics and laws of software evolution - the nineties view. In *Proc. of the 4th International Symposium on Software Metrics (METRICS '97)*, pp. 20–32. IEEE Computer Society, 1997.

[15] M. M. Lehman and Juan F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, Vol. 11, No. 1, pp. 15–44, 2001.

[16] David Lorge Parnas. Software aging. In *Proc. of the 16th international conference on Software engineering (ICSE '94)*, pp. 279–287. IEEE CS, 1994.

[17] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[18] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proc. of the Conference on The Future of Software Engineering (FOSE '00) in ICSE '00*, pp. 73–87. ACM, 2000.

[19] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice - Unanticipated Software Evolution*, Vol. 17, No. 5, pp. 309–332, September 2005.

[20] Eclipse plugin. http://www.eclipseplugincentral.com/.

[21] mozilla. Add-ons for firefox. https://addons.mozilla.org/firefox/.

[22] Mikael Lindvall and Kristian Sandahl. How well do experienced software developers predict software change? *Journal of Systems and Software*, Vol. 43, pp. 19–27, October 1998.

[23] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. In *Proc. of the 34th International Conference on Software Engineering (ICSE '12)*, pp. 430–440. IEEE, 2012.

[24] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean

Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proc. of the 26th International Conference on Software Engineering (ICSE '04)*, pp. 491–500. IEEE CS, 2004.

[25] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Communications of the ACM*, Vol. 36, No. 11, pp. 81–94, November 1993.

[26] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proc. of the 31st International Conference on Software Engineering (ICSE'09)*, pp. 78–88. IEEE CS, 2009.

[27] Hongyu Pei Breivold, Ivica Crnkovic, and Peter J. Eriksson. Analyzing software evolvability. In *Proc. of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC '08)*, COMPSAC '08, pp. 327–330. IEEE Computer Society, 2008.

[28] Bashar Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, Vol. 34, No. 3, pp. 115–117, 2001.

[29] B. Ramesh, T. Powers, C. Stubbs, and M. Edwards. Implementing requirements traceability: a case study. In *Proc. of the Second IEEE International Symposium on Requirements Engineering (RE '95)*, RE '95, pp. 89–95. IEEE Computer Society, 1995.

[30] Benedikt Burgstaller and Alexander Egyed. Understanding where requirements are implemented. In *Proc. of the 2010 IEEE International Conference on Software Maintenance (ICSM '10)*, ICSM '10, pp. 1–5, Washington, DC, USA, 2010. IEEE Computer Society.

[31] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proc. of the Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, pp. 249–262. IEEE CS, 2001.

[32] Annie I. Antón and Colin Potts. The use of goals to surface requirements for evolving systems. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pp. 157–166, Kyoto, Japan, 1998. IEEE Computer Society.

[33] Axel van Lamsweerde. From system goals to software architecture. In *Proc. of Formal Methods for Software Architectures*, Vol. LNCS 2804, pp. 25–43. Springer, 2003.

[34] Yijun Yu, Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Julio C. S. P. Leite. From goals to high-variability software design. In *Proc. of the 17th international conference on Foundations of intelligent systems (ISMIS'08)*, pp. 1–16. Springer-Verlag, 2008.

[35] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory

Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, Vol. 14, No. 3, pp. 54–62, 1999.

[36] IBM. An architectural blueprint for autonomic computing. http://www-03.ibm.com/autonomic/pdfs/ACJune 2005.

[37] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, Vol. 40, No. 3, pp. 1–28, 2008.

[38] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, Vol. 20, No. 1-2, pp. 3–50, 1993.

[39] Emmanuel Letier. *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD thesis, Universite Catholique de Louvain, 2001.

[40] Robert Darimont and Axel van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proc. of the 4th ACM SIGSOFT symposium on Foundations of software engineering (SIGSOFT/FSE'96)*, pp. 179–190, 1996.

[41] Axel van Lamsweerde, Emmanual Letier, and Robert Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, Vol. 24, No. 11, pp. 908–926, 1998.

[42] Robert Darimont. *Process Support for Requirements Elaboration*. PhD thesis, Universite Catholique de Louvain, 1995.

[43] Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proc. of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14)*, pp. 197–207. ACM, 2006.

[44] CEDITI. Objectiver. http://www.objectiver.com/.

[45] Dia. http://live.gnome.org/Dia.

[46] Eya Ben Charrada and Martin Glinz. An automated hint generation approach for supporting the evolution of requirements specifications. In *Proc. of the Joint Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE-EVOL '10)*, pp. 58–62. ACM, 2010.

[47] Telecom Italia. JADE: Java agent development framework. http://jade.tilab.com/.

[48] Tim Lindholm and Frank Yellin. The java virtual machine specification (2nd edition). `http://java.sun.com/docs/books/jvms/second_edition/html/`

`VMSpecTOC.doc.html`.

[49] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.

[50] Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides, 本位田 真一 (翻訳), 吉田 和樹 (翻訳). オブジェクト指向における再利用のためのデザインパターン. ソフトバンククリエイティブ, 1999.

[51] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, Vol. 12, pp. 42–50, 1995.

[52] Bashar Nuseibeh. Crosscutting requirements. In *Proc. of the 3rd international conference on Aspect-oriented software development (AOSD'04)*, AOSD '04, pp. 3–4. ACM, 2004.

[53] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, pp. 308 – 320, Dec. 1976.

[54] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *Proc. of the 34th International Conference on Software Engineering (ICSE '12)*, pp. 419–429. IEEE, 2012.

[55] Frank Buschmann, Hans Rohnert, Michael Stal, Regine Meunier, Peter Sommerlad, 金沢 典子 (翻訳), 水野 貴之 (翻訳), 桜井 麻里 (翻訳), 関 富登志 (翻訳), 千葉 寛之 (翻訳). ソフトウェアアーキテクチャ: ソフトウェア開発のためのパターン体系. 近代科学社, 2000.

[56] Eric S. K. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *In Proc. of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, pp. 226–235, 1997.

[57] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer, 2000.

[58] Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood. Running JADE agents on mobile devices. In *developing multi-agent systems with JADE*, chapter 8, pp. 145–171. WILEY, 2007.

[59] Object Management Group (OMG). Model Driven Architecture. http://www.omg.org/mda/.

[60] OMG. OCL 2.0 specification. http://www.omg.org/docs/ptc/05-06-06.pdf, 2005.

[61] 荒木啓二郎, 張漢明. プログラム仕様記述論. オーム社, 2002.

[62] 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.

[63] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems, Practical Tools and Techniques in Software Development*. Cambridge University Press, 1998.

[64] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, 2005.

[65] The Java Modeling Language (JML). http://www.cs.ucf.edu/~leavens/JML/.

[66] Yijun Yu, Julio Cesar Sampaio do Prado Leite, Alexei Lapouchnian, and John Mylopoulos. Configuring features with stakeholder goals. In *Proc. of the 2008 ACM symposium on Applied computing (SAC '08)*, SAC '08, pp. 645–649, New York, NY, USA, 2008. ACM.

[67] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[68] Jack Greenfield and Keith Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[69] W. Heaven and A. Finkelstein. UML profile to support requirements engineering with KAOS. *Software, IEE Proceedings*, Vol. 151, No. 1, pp. 10–27, Feb. 2004.

[70] Sombat Chanvilai, Kozo Honda, Hiroyuki Nakagawa, Yasuyuki Tahara, and Akihiko Ohsuga. Generating uml class diagrams and ocl constraints from kaos model. In *Proc. of the 27th ACM Symposium On Applied Computing (SAC '12)*, 2012.

[71] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[72] Kohei Uno, Shinpei Hayashi, and Motoshi Saeki. Constructing feature models using goal-oriented analysis. In *Proc. of the 2009 Ninth International Conference on Quality Software (QSIC '09)*, QSIC '09, pp. 412–417, Washington, DC, USA, 2009. IEEE Computer Society.

[73] Mohsen Asadi, Ebrahim Bagheri, Dragan Gašević, Marek Hatala, and Bardia Mohabbati. Goal-driven software product line engineering. In *Proc. of the 2011 ACM Symposium on Applied Computing (SAC '11)*, SAC '11, pp. 691–698, New York, NY, USA, 2011. ACM.

[74] Yijun Yu, Julio Cesar Sampaio do Prado Leite, and John Mylopoulos. From goals to aspects: Discovering aspects from requirements goal models. In *Proc. of the 12th IEEE International Conference on Requirements Engineering (RE '04)*, pp. 38–47. IEEE CS, 2004.

[75] Yijun Yu, Yiqiao Wang, John Mylopoulos, Sotirios Liaskos, Alexei Lapouchnian, and Julio Cesar Sampaio do Prado Leite. Reverse engineering goal models from legacy code. In *Proc. of the 13th IEEE International Conference on Requirements Engineering (RE '05)*,

pp. 363–372, Washington, DC, USA, 2005. IEEE Computer Society.

[76] Hidekazu Omote, Kohta Sasaki, Haruhiko Kaiya, and Kenji Kaijiri. Software Evolution Support Using Traceability Link between UML diagrams. In *Proc. of the Sixth Joint Conference on Knowledge-Based Software Engineering (JCKBSE04)*, pp. 15–23. IOS Press, August 2004. Proc. of the 6th JCKBSE.

[77] Doug Rosenberg and Matt Stephens. *Use Case Driven Object Modeling with UML Theory and Practice*. Apress, 2007.

[78] Earl F. Ecklund, Jr., Lois M. L. Delcambre, and Michael J. Freiling. Change cases: use cases that identify future requirements. In *Proc. of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '96)*, OOPSLA '96, pp. 342–358, New York, NY, USA, 1996. ACM.

[79] Jane Cleland-Huang, Raffaella Settimi, Oussama BenKhadra, Eugenia Berezhanskaya, and Selvia Christina. Goal-centric traceability for managing non-functional requirements. In *Proc. of the 27th international conference on Software engineering (ICSE '05)*, ICSE '05, pp. 362–371. ACM, 2005.

[80] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, Vol. 32, No. 1, pp. 4–19, January 2006.

[81] Masahiro Umemura, Haruhiko Kaiya, Shinpei Ogata, and Kenji Kaijiri. Validating quality requirements considerations in a design document using spectrum analysis. In *Proc. of the 10th Joint Conference on Knowledge-Based Software Engineering (JCKBSE '12)*, pp. 88–97. IOS Press, 2012.

[82] Gustavo A. Oliva, Francisco W.S. Santana, Marco A. Gerosa, and Cleidson R.B. de Souza. Towards a classification of logical dependencies origins: a case study. In *Proc. of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution (IWPSE-EVOL '11)*, IWPSE-EVOL '11, pp. 31–40, New York, NY, USA, 2011. ACM.

[83] Vaclav Rajlich. A model for change propagation based on graph rewriting. In *Proc. of the International Conference on Software Maintenance (ICSM97)*, ICSM '97, pp. 84–91. IEEE CS, 1997.

[84] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proc. of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA*

'05), OOPSLA '05, pp. 167–176. ACM, 2005.

[85] M. D'Ambros, M. Lanza, and M. Lungu. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering*, Vol. 35, No. 5, pp. 720 –735, sept.-oct. 2009.

[86] 清水吉男. 「派生開発」を成功させるプロセス改善の技術と極意. 技術評論社, 2007.

[87] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, Vol. 26, No. 10, pp. 978–1005, 2000.

[88] CETIC. Faust. http://faust.cetic.be.

[89] C. Ponsard, P. Massonet, J. F. Molderez, A. Rifaut, A. Van Lamsweerde, and H. Tran Van. Early verification and validation of mission critical systems. *Formal Methods in System Design*, Vol. 30, No. 3, pp. 233–247, 2007.

[90] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Jr. Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, pp. 390–406, June 1996.

[91] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering - Special issue on software architecture*, Vol. 21, No. 4, pp. 314–335, April 1995.

[92] Robert Allen and David Garlan. Formalizing architectural connection. In *Proc. of the 16th International Conference on Software engineering (ICSE '94)*, ICSE '94, pp. 71–80, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[93] Robert John Allen. *A formal approach to software architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1997. AAI9813815.

[94] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering - Special issue on software architecture*, Vol. 21, No. 4, pp. 336–355, April 1995.

[95] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proc. of the 31st International Conference on Software Engineering (ICSE'09)*, ICSE '09, pp. 309–319. IEEE Computer Society, 2009.

[96] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for frame-

work evolution. In *Proc. of the 30th international conference on Software engineering (ICSE'08)*, ICSE '08, pp. 481–490. ACM, 2008.

[97] Martin L. Griss, Steven P. Fonseca, Richard M. Cowan, and Robert R. Kessler. Using UML state machine models for more precise and flexible JADE agent behaviors. In *Agent-Oriented Software Engineering III (AOSE 2002)*, pp. 113–125, Bologna, Italy, July 2002. Springer.

[98] Pavlos Moraïtis, Eleftheria Petraki, and Nikolaos I. Spanoudakis. Engineering JADE agents with the Gaia methodology. In *Agent Technologies, Infrastructures, Tools, and Applications for E-Services*, Vol. 2592, pp. 77–91. Springer, 2003.

[99] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, Vol. 12, No. 3, pp. 317–370, 2003.

[100] Magid Nikraz, Giovanni Caire, and Parisa A. Bahri. A methodology for the analysis and design of multi agent systems using JADE. http://jade.tilab.com/doc/tutorials/JADE_methodology_website_version.pdf, 2006.

[101] Telecom Italia. Workflows and agents development environment (WADE). http://jade.tilab.com/wade/ index.html.

[102] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: Implementing a BDI-infrastructure for JADE agents. *EXP - in search of innovation (Special Issue on JADE)*, Vol. 3, No. 3, pp. 76–85, September 2003.

[103] Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In *ICMAS*, pp. 312–319. The MIT Press, 1995.

[104] Manuel Oriol. *An Approach to the Dynamic Evolution of Software Systems*. PhD thesis, University of Genéva, 2004.

[105] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proc. of the 20th international conference on Software engineering (ICSE '98)*, pp. 177–186. IEEE Computer Society, 1998.

[106] Oracle. Understanding extension class loading. `http://download.oracle.com/javase/tutorial/ext/basics/load.html`.

[107] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, Vol. 25, pp. 40–51, October 1992.

[108] Bertrand Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., 1997.

[109] Eiffel Software. Eiffel: The reference. `http://archive.eiffel.com/nice/`

language/.

[110] David R. Cok and Joseph R. Kiniry. Esc/java2: Uniting esc/java and jml –progress and issues in building and using esc/java2. In *Proc. of International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS '04)*. Springer-Verlag, 2004.

[111] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. of European Conference on Object-Oriented Programming (ECOOP '97)*, pp. 220–242. Springer, 1997.

[112] 千葉滋, 立堀道昭, 佐藤芳樹, 中川清志. リフレクションの高速化技術. コンピュータソフトウェア, Vol. 21, No. 6, pp. 427–437, 2004.

[113] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proc. of the 15th European Conference on Object-Oriented Programming (ECOOP01)*, ECOOP '01, pp. 327–353. Springer-Verlag, 2001.

[114] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pp. 120–130, New York, NY, USA, 1999. ACM.

[115] The Seasar Foundation. Seasar. http://www.seasar.org/.

[116] The Apache Software Foundation. Tomcat. http://tomcat.apache.org/.

[117] ORACLE. Java platform debugger architecture. http://java.sun.com/javase/technologies/core/toolsapis/jpda/.

[118] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic java classes. In *Proc. of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, pp. 337–361, London, UK, 2000. Springer-Verlag.

[119] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pp. 205–230, London, UK, 2002. Springer-Verlag.

[120] Yoshiki Sato and Shigeru Chiba. Negligent class loaders for software evolution. In *PROC. of ECOOP '04 WORKSHOP ON REFLECTION, AOP AND META-DATA FOR SOFTWARE EVOLUTION (RAM-SE '04)*, pp. 53–58, 2004.

[121] Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, and et al. Software engineering for self-adaptive systems: A research road map. In *Dagstuhl Seminar Proceedings 08031*, 2008.

[122] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research

challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, Vol. 4, No. 2, pp. 1–42, 2009.

[123] Hausi Müller, Mauro Pezzè, and Mary Shaw. Visibility of control in adaptive systems. In *Proc. of the 2nd international workshop on Ultra-large-scale software-intensive systems*, ULSSIS '08, pp. 23–26. ACM, 2008.

[124] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. On interacting control loops in self-adaptive systems. In *Proc. of the 6th international symposium on Software engineering for adaptive and self-managing systems (SEAMS'11)*, pp. 202–207. ACM, 2011.

[125] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, Vol. 36, No. 1, pp. 41–50, 2003.

[126] S. Fickas and M.S. Feather. Requirements monitoring in dynamic environments. In *Proc. of the Second IEEE International Symposium on Requirements Engineering (RE'95)*, pp. 140–147. IEEE Computer Society, 1995.

[127] M. Salifu, Yijun Yu, and B. Nuseibeh. Specifying monitoring and switching problems in context. In *Proc. of the 15th IEEE International Requirements Engineering Conference (RE '07)*, pp. 211–220. IEEE CS, 2007.

[128] Michael Jackson. *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley, 2000.

[129] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H.C. Cheng, and Jean-Michel Bruel. RELAX: incorporating uncertainty into the specification of self-adaptive systems. In *Proc. of the 17th IEEE International Conference on Requirements Engineering (RE '09)*, pp. 79–88. IEEE CS, 2009.

[130] E. S. K. Yu. Modeling organizations for information systems requirements engineering. In *Proc. of the First IEEE International Symposium on Requirements Engineering*, pp. 34–41, 1993.

[131] Betty H. C. Cheng, Peter Sawyer, Nelly Bencomo, and Jon Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Proc. of the 12th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS '09)*, pp. 468–483. Springer, 2009.

[132] Alexei Lapouchnian, Yijun Yu, Sotirios Liaskos, and John Mylopoulos. Requirements-driven design of autonomic application software. In *Proc. of the 2006 conference of the Center for Advanced Studies on Collaborative research (CASCON '06)*, pp. 80–94, 2006.

[133] Renaud De Landtsheer, Emmanuel Letier, and Axel van Lamsweerde. Deriving tabular

event-based specifications from goal-oriented requirements models. In *Proc. of the 11th IEEE International Conference on Requirements Engineering (RE '03)*, pp. 200–210. IEEE CS, 2003.

[134] Yiqiao Wang and John Mylopoulos. Self-repair through reconfiguration: A requirements engineering approach. In *Proc. of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, pp. 257–268. IEEE CS, 2009.

[135] Bihuan Chen, Xin Peng, Yijun Yu, and Wenyun Zhao. Are your sites down? requirements-driven self-tuning for the survivability of web systems. In *Proc. of the 19th IEEE International Requirements Engineering Conference (RE '11)*, pp. 219–228, 2011.

[136] Erann Gat, R. Peter Bonnasso, Robin Murphy, and Aaai Press. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*, pp. 195–210. AAAI Press, 1998.

[137] John C. Georgas and Richard N. Taylor. Policy-based self-adaptive architectures: a feasibility study in the robotics domain. In *Proc. of the 2008 international workshop on Software engineering for adaptive and self-managing systems (SEAMS '08)*, pp. 105–112. ACM, 2008.

[138] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From goals to components: a combined approach to self-management. In *In Proc. of the International Workshop on Software Engineering for Adaptive and Self-managing Systems (SEAMS '08)*, pp. 1–8, Leipzig, Germany, 2008. ACM.

[139] Howard Foster, Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Towards self-management in service-oriented computing with modes. In *3rd International Workshop on Engineering Service Oriented Applications: Analysis, Design and Composition - WESOA07*, pp. 338–350, Vienna, Austria, Sep 2007. Springer LNCS.

[140] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, Vol. 37, No. 10, pp. 46–54, Oct. 2004.

[141] R. Asadollahi, M. Salehie, and L. Tahvildari. Starmx: A framework for developing self-managing java-based systems. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on*, pp. 58–67, May 2009.

[142] Ian Gorton, Yan Liu, and Nihar Trivedi. An extensible, lightweight architecture for adaptive J2EE applications. In *SEM '06: Proceedings of the 6th international workshop on Software engineering and middleware*, pp. 47–54, New York, NY, USA, 2006. ACM.

[143] Sylvain Sicard, Fabienne Boyer, and Noel De Palma. Using components for architecture-

based management: the self-repair case. In *Proc. of the 30th international conference on Software engineering (ICSE '08)*, pp. 101–110, Leipzig, Germany, 2008. ACM.

[144] Sara Bouchenak, Fabienne Boyer, Sacha Krakowiak, Daniel Hagimont, Adrian Mos, Stefani Jean-Bernard, Noel de Palma, and Vivien Quema. Architecture-based autonomous repair management: An application to j2ee clusters. In *Proc. of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS '05)*, pp. 13–24. IEEE Computer Society, 2005.

[145] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, N. de Palma, V. Quema, and J.-B. Stefani. Architecture-based autonomous repair management: Application to j2ee clusters. In *Proc. of the 2nd International Conference on Autonomic Computing (ICAC'05)*, pp. 369–370, Seattle, Washington, June 2005. IEEE Computer Society.

[146] Sun Microsystems. Java Management Extensions (JMX) technology. http://java.sun.com/javase/technologies/core/ mntr-mgmt/javamanagement/.

[147] Mirko Morandini, Loris Penserini, and Anna Perini. Towards goal-oriented development of self-adaptive systems. In *Proc. of the International Workshop on Software Engineering for Adaptive and Self-managing Systems (SEAMS'08)*, pp. 9–16, 2008.

[148] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering (ICSE '06)*, pp. 371–380. ACM, 2006.

[149] Ji Zhang and Betty H. C. Cheng. Specifying adaptation semantics. In *Proceedings of the 2005 workshop on Architecting dependable systems (WADS '05)*, pp. 1–7. ACM, 2005.

[150] Ina Schaefer and Arnd Poetzsch-Heffter. Slicing for model reduction in adaptive embedded systems development. In *Proc. of the 2008 international workshop on Software engineering for adaptive and self-managing systems (SEAMS '08)*, pp. 25–32, Leipzig, Germany, 2008.

[151] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, pp. 352–357, 1984.

[152] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Slicing of uml models using model transformations. In *Proc. of the 13th international conference on Model driven engineering languages and systems (MODELS '10)*, MODELS'10, pp. 228–242, Berlin, Heidelberg, 2010. Springer-Verlag.

[153] Object Management Group. Unified Modeling Language (UML). `http://www.uml.org/`.

[154] CSK. VDM tools – The VDM++ Language Manual. `http://www.vdmtools.jp/uploads/manuals/langmanpp_a4E.pdf`.

[155] Cliff B Jones. *Systematic Software Development Using VDM (Second Edition)*. Prentice Hall, 1990.

[156] Gerard J. Holzmann. The model checker SPIN. *IEEE transactions on Software Engineering*, Vol. 23, No. 5, pp. 279–295, 1997.

[157] Hassan Gomaa. Towards feature-based evolutionary software modeling. In *Preliminary Proc. of International Workshop on Models and Evolution (ME'11)*, pp. 64–73, 2011.

[158] Sheng Huang, Zhong Jie Li, Jun Zhu, Yanghua Xiao, and Wei Wang. A novel approach to regression test selection for j2ee applications. In *Proc. of IEEE 27th International Conference on Software Maintenance (ICSM'11)*, pp. 13–22. IEEE, 2011.

[159] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. express: Guided path exploration for efficient regression test generation. In *Proc. of 2011 International Symposium on Software Testing and Analysis (ISSTA'11)*, pp. 1–11. ACM, 2011.

[160] Prahladavaradan Sampath, Silky Arora, and S. Ramesh. Evolving specifications formally. In *Proc. of 19th IEEE International Requirements Engineering Conference (RE'11)*, pp. 5–14. IEEE, 2011.

研究業績

種類別	題名、　発表・発行掲載誌名、　発表・発行年月、　連名者（申請者含む）
1.論文	**【学術誌原著論文】**
(1)	○　ゴール指向要求記述の整形に基づいたソフトウェアシステム進化手法，情報処理学会論文誌，Vol.53，No.10，pp.2328-2344，2012 年，中川博之，大須賀昭彦，本位田真一
(2)	○　ビヘイビア記述に基づく自己適応システム実装フレームワークの提案，人工知能学会論文誌，Vol.26，No.1，pp.1-12，2011 年，中川博之，大須賀昭彦,本位田真一
(3)	自律システム実現に向けたアーキテクチャの構築，人工知能学会論文誌,Vol.26,No.1,pp.107-115，2011 年，西村一彦，中川博之，田原康之，大須賀昭彦
(4)	○ プロセス間競合を考慮した自己適応システムの形式仕様構築，情報処理学会論文誌，Vol.51，No.9，pp.1751-1764，2010 年，中川博之，大須賀昭彦，本位田真一
(5)	○　ゴール指向要求分析を用いた self-adaptive システムの構築，情報処理学会論文誌，Vol.50，No.10，pp.2500-2513，2009 年，中川博之，大須賀昭彦，本位田真一
(6)	○ モデル変換に基づく要求記述を利用した形式仕様記述の構築，情報処理学会論文誌，Vol.49，No.7，pp.2304-2318，2008 年，中川博之，田口研治，本位田真一
(7)	○　要求の重要度を用いたマルチエージェントシステムの分析モデル検証，電子情報通信学会論文誌，Vol.J90-D，No.9，pp.2281-2292，2007 年，中川博之，吉岡信和，本位田真一
(8)	○　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 年，中川博之，吉岡信和，本位田真一
2.講演	**【国際会議・シンポジウム】**（査読あり）
(1)	○　Towards Dynamic Evolution of Self-adaptive Systems Based on Dynamic Updating of Control Loops, in Proc. of 6th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2012), pp.59-68, 2012.9, Nakagawa, H., Ohsuga, A., Honiden, S.
(2)	Goal-oriented Approach to Creating Class Diagrams with OCL Constraints, in Proc. of the 27th ACM Symposium On Applied Computing (SAC2012), pp.1051-1056, 2012.3, Sombat, C., Honda, K., Nakagawa, H., Tahara, Y., Ohsuga, A.
(3)	Dynamic Reconfiguration in Self-adaptive Systems Considering Non-functional Properties, in Proc. of the 27th ACM Symposium On Applied Computing (SAC2012), pp.1144-1150, 2012.3, Horikoshi, H., Nakagawa, H., Tahara, Y., Ohsuga, A.

研究業績

種類別	題名、　発表・発行掲載誌名、　発表・発行年月、　連名者（申請者含む）
(4)	○ gocc: A Configuration Compiler for Self-adaptive Systems Using Goal-oriented Requirements Description, in Proc. of the ACM/IEEE 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), co-located with ICSE 2011, pp.40-49, 2011.5, Nakagawa, H., Ohsuga, A., Honiden,S.
(5)	○ IMPULSE: a Design Framework for Multi-Agent Systems Based on Model Transformation, in Proc. of the 26th ACM Symposium On Applied Computing (SAC2011), pp.1416-1423, 2011.3, Nakagawa, H., Ohsuga, A., Honiden, S.
(6)	○ A Framework for Validating Task Assignment in Multi-agent Systems using Requirements Importance, in Proc. of the 13th International Conference on Principles and Practice of Multi-Agent Systems (PRIMA2010) (Early Innovation Track), pp.443-458, 2010.11, Nakagawa, H., Ohsuga, A., Honiden, S.
(7)	○ Cooperative Behaviors Description for Self-* Systems Implementation, in Proc. of the 8th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS2010), pp.69-74 (ショートペーパー), 2010.4, Nakagawa, H., Ohsuga, A., Honiden, S.
(8)	○ Formal Specification Generator for KAOS, in Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE2007) (Tool Demonstrations Track), pp.531-532 (ツールデモンストレーション), 2007.11, Nakagawa, H., Kenji Taguchi, Honiden, S.
(9)	○ Analysis of Multi-Agent Systems based on KAOS Modeling, in Proc. of the 28th International Conference on Software Engineering (ICSE2006) (Emerging Results Track), pp.926-929 (ポスター発表), 2006.5, Nakagawa, H., Karube, T., Honiden,S.
	【国際ワークショップ】（査読あり）
(10)	○ Towards Effective Use of Requirements Description in Self-adaptive System Development, in Proc. of the International Workshop on Modern Science and Technology 2010 (IWMST2010), pp.100-105 (アブストラクト査読), 2010.9, Nakagawa, H., Ohsuga, A., Honiden, S.
(11)	○ Constructing Self-adaptive Systems Using KAOS Model, in Proc. of the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2008), pp.132-137, 2008.10, Nakagawa, H., Ohsuga, A., Honiden,S.
	【国内ワークショップ・シンポジウム】（査読あり）
(12)	○ KAOS モデルを利用した self-adaptive システムの構築,ソフトウェア工学の基礎ワークショップ(FOSE2008), pp.21-30, 2008.11, 中川博之, 大須賀昭彦, 本位田真一

研究業績

種類別	題名、　発表・発行掲載誌名、　発表・発行年月、　連名者（申請者含む）
(13)	○　コンポーネントモデルを用いた JADE ビヘイビア実装手法の提案, 合同エージェントワークショップ＆シンポジウム（JAWS2008), pp.1-8, 2008.10, 中川博之, 大須賀昭彦, 本位田真一
(14)	○　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, ソフトウェア工学の基礎ワークショップ(FOSE2006), pp.131-136, 2006.11, 中川博之, 本位田真一
(15)	○　要求の重要度を用いたロール・組織構造の同定", 合同エージェントワークショップ＆シンポジウム(JAWS2006), pp.1-8, 2006.10, 中川博之, 本位田真一
(16)	○　KAOS を用いたマルチエージェントシステムの分析支援, 合同エージェントワークショップ＆シンポジウム(JAWS2005), pp.253-260, 2005.11, 中川博之, 苅部卓哉, 本位田真一
3.その他	**（その他学術誌原著論文）** 下記の他、8 件
(1)	服飾オントロジーを用いた EC サイトにおけるユーザデザイン嗜好の推定と評価, 情報処理学会論文誌, Vol.53, No.11（2012 年 11 月掲載決定), 全泰賢, 川村隆浩, 中川博之, 田原康之, 大須賀昭彦
(2)	センチメント分析とトピック抽出によるマイクロブログからの評判傾向抽出, 電子情報通信学会論文誌, Vol.J94-D, No.11, pp.1762-1772, 2011 年, 橋本和幸, 中川博之, 田原康之, 大須賀昭彦
(3)	コンテンツ投稿型 SNS における未知性と意外性を考慮した推薦エージェントの提案, 電子情報通信学会論文誌, Vol.J94-D, No.11, pp.1800-1811, 2011 年, 住元宗一朗, 中川博之, 田原康之, 大須賀昭彦
(4)	ユビキタスコンピューティングにおけるアプリケーション開発手法に関する研究動向, 日本ソフトウェア科学会　学会誌『コンピュータソフトウェア』, Vol.25, No.4, pp.121-132, 2008 年, 鄭顕志, 中川博之, 川俣洋次郎, 吉岡信和, 深澤良彰, 本位田真一
	（その他国際会議・シンポジウム） 下記の他、12 件
(5)	Support for Video Hosting Service Users using Folksonomy and Social Annotation, in Proc. of 2012 IEEE/WIC/ACM International Conference on Web Intelligence (WIC 2012), pp.472-479, Ishino, K., Orihara, R., Nakagawa, H., Tahara, Y., Ohsuga, A.
(6)	**（その他国内ワークショップ・シンポジウム）** 下記の他、37 件 自律システム実現に向けたアーキテクチャの構築, 合同エージェントワークショップ＆シンポジウム(JAWS2009), pp.524-531（優秀論文賞受賞）, 2009.10, 西村一彦, 中川博之, 中山健, 田原康之, 大須賀昭彦