卒業論文

RoIS Framework を用いた
異なるロボットシステム間において互換性のある
ロボットサービスの構築

総合システム工学科

16111026　田中 大揮

指導教員：猪平　栄一　講師

提出日：令和 2 年 2 月 13 日

目次

第1章　序論

1.1　本研究の背景

これまで様々なタスクを実行するロボットの研究がなされ，ロボティクスにおける技術はソフトウェア，ハードウェアともに発達してきた．現在では，具体的には，教育現場における議論支援ロボット [1] や商業施設や博物館，図書館における案内ロボット [2, 3, 4]，医療施設における資材運送ロボット [5] など，様々な分野においてロボットを活用するための研究開発が行われている．ロボットサービスを実用化されるにあたって，ロボットがこれらの多様な環境に対して適応する必要がある．そのために，開発者にはロボットを開発・運用する技術だけでなく，ロボットサービスとして利用する業種や現場の知識が求められる．しかし，全ての専門分野の知識を網羅した研究者や技術者は存在し得ない．そのため，様々な分野の研究者や技術者がロボットサービスの開発に参入することで，ロボットを多様な環境に適応させる必要がある．

1.2　RoIS Framework

概要　RoIS Framework (Robotic Interaction Service Framework) は，2010 年に OMG (Object Management Group) によって策定された，ロボットサービスのアプリケーションから HRI (Human Robot Interaction) 機能を使うためのインタフェースを標準化するための枠組みである [6, 7]．HRI 機能とは，ロボットが人と相互作用をするための機能で，顔認識や音声認識，音声合成による発話，ジェスチャーなどがある．これらの HRI 機能は，様々な研究施設や企業によって開発されており，機能実現のためのハードウェア・ソフトウェアの実装やそれらを利用するためのインタフェースが統一されていない．HRI 機能を利用するためのインタフェースは統一されていないと，それらを利用するサービスのアプリケーションも HRI 機能の実装に合わせて再実装しなければならないという問題がある．

コンポーネント名	機能
System Information	システムの状態，ロボットの位置などの情報を提供する
Person Detection	人物の数を検出する
Person Localization	人物の位置を検出する
Person Identification	人物が誰であるのかを識別する
Face Detection	人の顔の数を検出する
Face Localization	人の顔の位置を検出する
Sound Detection	音源の数を検出する
Sound Localization	音源の位置を検出する
Speech Recognition	音声認識を行う
Gesture Recognition	ジェスチャーの認識を行う
Speech Synthesis	音声合成による発話を行う
Reaction	指定されたリアクションをとる
Navigation	指定された場所まで自律移動を行う
Follow	指定されたオブジェクトを追従する
Move	指定された距離，曲率で移動する

表 1.2.1: HRI コンポーネントの一覧

RoIS Framwwork は，HRI 機能を利用するためのインタフェースや，HRI 機能を利用してサービスを構築するためのサービスアプリケーションのインタフェースを規格化することで HRI 機能の実装による差異を吸収し，その問題を解決する手法として提案された．RoIS Framework を利用することで，サービスの内容を記述するプログラムであるサービスアプリケーションを，HRI 機能やロボットシステムの実装に依存することなく実装できる．これにより，異なるロボットシステム間におけるサービスアプリケーションの互換性が得られ，サービスアプリケーションを再利用できる．また，サービスアプリケーションを再利用できることでロボットシステムの実装とサービスアプリケーションの実装の分離が可能となり，それぞれを専門の技術者や研究者が担当することでロボットサービス開発を効率化できる．

RoIS Framework では，サービスロボットで用いられる基本的機能として，15 種類の HRI コンポーネントのインタフェースが規格化されている．RoIS Framework で規格化されている HRI コンポーネントの一覧を表 1.2.1 に示す．これ以外でユーザ定義の HRI コンポーネントを設定する仕組みも用意されている．

pyRoIS　pyRoIS [8] とは，九州工業大学工学部総合システム工学科で開発されている，RoIS Framework に則ったロボットサービスシステムを構築するための Python ライブラリであ

る．既に PyPI (Python Package Index) にて公開済みであり，オープンソース化されている．pyRoIS では RoIS Framework 利用して HRI 機能を利用したり，ロボット側からサービスアプリケーションに実行結果を通知したりするためのインタフェースが提供されており，これを利用することで開発を効率化できる．pyRoIS の実装の詳細については [9] を参照されたい．

1.3 シミュレーションの活用

ロボットのソフトウェア開発や実験において，しばしばシミュレーションソフトウェアが用いられる．シミュレーションを用いる利点としては，実機のロボットを使用しないためハードウェアの開発をする必要がないという点がある．すなわち，実機のロボットを開発するだけの作業と金銭的コストが不要となる．さらに，ロボットを管理したりメンテナンスしたりといった継続的に発生するコストもない．

欠点としては，実験時にロボットの各機能のプログラムに加えてそれらを再現するシミュレーションのプログラムを動作させる必要があるため，計算機にかかる負荷が増すことが考えられる．これらはスペックの高い計算機を用いることで容易に対処できる．

本研究においても，上で述べたシミュレーションの利点に着目し，ロボットの動作実験やロボットサービスの再現，動的な環境の変化に対するサービスシステムの再現性の確認のためにシミュレーションを利用した．例えば，実験環境におけるロボットの台数の変更についてはシミュレーション環境を利用すれば，シミュレーション環境の起動時にパラメータを変えるだけで済む．しかしこれを実機で行おうとすると，ロボットを開発するための金銭的・作業量的コストが台数倍に膨れ上がるだけでなく，継続的にメンテナンスをするための人員が必要となる．その他，タスク進行における障害発生の再現は，シミュレーションソフトウェアからロボットをマウス操作して倒すことで簡単に再現でき，実験場所のモデルはコマンドやパラメータを操作するだけで変更できる．これらについても，実機による実験を行おうとするとロボットの部品や実験環境にある器物の破損に繋がったり，ロボットの運搬にコストが発生したりと，コストの面で問題となる．

このように実機のロボットでは難しい，実験条件の変更や障害の再現をシミュレーションであれば容易に行える．

1.4 本研究の目的

上で述べたように， RoIS Framework はサービスアプリケーションの異なるロボットシステム間における互換性を実現するための規格である．現に，音声認識や自律移動などの一通りの HRI 機能を利用するためのインタフェースが RoIS Framework 仕様書には記述されている．しかし，規格こそされているが，現状 RoIS Framework を利用したロボットサービスが公開・普及されておらず，RoIS Framework を利用してロボットサービスを構築する手法が確立されてない．また，機能の実装レベルすなわちコンポーネントレベルの違いであれば，機能ごとに共通のインタフェースに従うことで互換性を実現できることが既にわかっているが，ロボットの台数や，それに伴う連携等の有無の違いなど，コンポーネントレベルよりも高いレベルに波及するシステムの違いにおいても互換性を保たれるかが確認されていない．また，ロボットが複数台になることで連携の要素が加わり，サービスアプリケーションからの同じ要求に対するロボットの動作を変えるための手法が確立されておらず，これらについて，実際にロボットサービスを実装し検証する必要であると考えられる．

本研究では，ロボットサービスアプリケーションの台数の異なるロボットシステムにおける互換性の実現に向けて，具体的なロボットサービスをシミュレーションを利用しながら構築する．それを通して，台数の異なる場合でも互換性を実現する手法を提案し，有効性の検証を行う．

1.5 本論文の構成

本論文は以下の章から構成される．

第 1 章「序論」では研究の背景および目的について述べた．

第 2 章「互換性実現のための方針」では，RoIS Framework が目的とする機能と本研究におけるロボットサービスの開発方針について述べる．

第 3 章「ロボットサービスのシミュレーションと互換性の評価」では実際にロボットサービスを開発した方法と複数の実験条件におけるサービスアプリケーションの互換性の検証結果について述べる．

第 4 章「考察」では、台数の異なるロボットシステムに対して互換性を保つことのできるロボットサービスの実装方法と RoIS Framework に残された課題について考察する．

第 5 章「結論」では，以上の内容を踏まえて結論を述べる．

第2章　互換性実現のための方針

2.1　まえがき

本章では，RoIS Framework の背景にある概念の説明と関連研究との違い，RoIS Framework の構造について説明した後に，それらを踏まえて，シミュレーション環境およびロボットサービスを構築するにあたっての開発方針について述べる．

2.2　ロボットの運動レベル

ロボットの運動は，大きく分けて 3 つのレイヤに分類される．それぞれレイヤの高い順に movement, action, behavior と呼ばれる．RoIS Framework の構造や関連研究との差別化について述べる際にこれらの概念を用いて説明する必要があるため，各運動レベルについて，ロボットが「二足歩行で目的地まで歩く」というタスクを例に説明する．本節における説明は浅田 稔, 國吉 康夫のロボットインテリジェンス [10] を参考にした．

movement レベル　movement はロボットの運動レベルの中で最も低いレイヤで，モータを回転させるなどのアクチュエータを操作するレベルの運動を指す．二足歩行で目的地まで歩くタスクで例えると，膝関節のモータを 2π [rad/s] の角速度で 0.5 秒間だけ回転させるといった運動が movement に該当する．

action レベル　action は，movement の 1 つ上のレイヤで，いくつかの movement を組み合わせて形成される人間が認識できる意味を持った運動を指す．二足歩行で目的地まで歩くタスクで例えると，姿勢を保つ，両足を交互に前に出して前進するなどといった運動がそれぞれ action に該当する．また，RoIS Framework で規格されている HRI コンポーネントは action レベルの運動の規格と言える．

behavior レベル　behavior は action のさらに 1 つ上のレイヤであり，いくつかの action を組み合わせて形成される，ある目的を達成するための一連の運動を指す．例えば，二足歩行で目的地まで歩くというタスク自体が behavior と言え，障害物に遭遇するなどの環境変化や転倒するなどの例外の発生にも対処しながら目的を達成する運動のことを behavior と呼ぶ．生産ライン方式の工場で稼働するような，決められた動作のみを行うロボットではなく，人にサービスを提供したり人と共同で作業したりするロボットには behavior レベルの，臨機応変な動作が求められる．

2.3 関連研究

2.3.1 ROS

RoIS Framework と同じ,「ロボットソフトウェアのフレームワーク」として ROS [11] が挙げられる．ROS とは，ロボット制御のためのソフトウェアフレームワークで，現在多くのロボット開発現場で利用されている．ROS も RoIS Framework と同じく，ロボットソフトウェアにおける標準化という役割を持つ．しかし，両者は標準化するレイヤという点で異なる．

ROS は,「速度の指令値」や「センサ情報」などといったロボットの制御レベルの情報伝達を標準化している．これに対して RoIS Framework は,「音声認識」や「自律移動」の指令およびそれらの実行結果などといったタスクレベルの情報伝達を標準化している．すなわち，2.2 で述べた内容を踏まえると，ROS は movement や action などといった比較的低いレイヤの運動を，RoIS Framework では 比較的高いレイヤである behavior レベルの運動を実現するためのフレームワークと言える．3.3.2 で述べる ROS の Navigation Stack[12] は自律移動のタスクレベルの標準化の 1 つといえるが，RoIS Framework の方が HRI によるサービスにおけるタスクを網羅的に扱っている．

RoIS Framework と ROS それぞれが標準化している運動のレベルを表 2.3.1 に，タスクを表 2.3.2 に示す．

	運動レベル		
フレームワーク	movement	action	behabior
RoIS Framework	×	◯	◎
ROS	◯	◯	△

表 2.3.1: RoIS Framework と ROS の各運動レベルへの対応

	標準化の対応	
コンポーネント名	RoIS Framework	ROS
System Information	◯	×
Person Detection	◯	×
Person Localization	◯	×
Person Identification	◯	×
Face Detection	◯	×
Face Localization	◯	×
Sound Detection	◯	×
Sound Localization	◯	×
Speech Recognition	◯	×
Gesture Recognition	◯	×
Speech Synthesis	◯	×
Reaction	◯	×
Navigation	◯	◯
Follow	◯	×
Move	◯	◯

表 2.3.2: RoIS Framewok と ROS の各機能に対する標準化の有無

2.3.2 SIGVerse

SIGVerse[13] とは，本研究と同じく，HRI によるロボットサービスのために構築されたシミュレーションシステムである．SIGVerse の特徴は，VR を用いて人をシミュレーション環境に投影することで人とロボットのインタラクションをシミュレートできるという点にある．これを利用することで，実機のロボットを利用したり専用の実験環境を用意したりすることなくロボットの動作に対するリアルな人間のリアクションを学習用データとして蓄積したり実験や検証に利用したりできる．

RoIS Framework との違いとしては，SIGVerse では人とロボットのインタラクションに焦点を当てているのに対し，RoIS Framework では実装や挙動の異なるロボットシステム間におけるロボットサービスのソフトウェア的な互換性に焦点を当てている点にある．

2.4 RoIS Framework の構造

RoIS Framework は，サービスアプリケーション，HRI エンジン，HRI コンポーネントの 3 つに分けられている．以下に，それぞれについての説明する．また，これ以降の説明において,「タスク」とは「ユーザの位置を特定する」,「目的地に向かって自律移動する」,「音声認識を行う」などといった，ロボットがある目的をもって起こす action のことを指し,「サービス」とは，タスクの中でもユーザの抱える問題を解決するなどの，人に直接影響を与えるものを指すこととする．

HRI コンポーネント　HRI コンポーネントとは，HRI 機能を提供するソフトウェアあるいはハードウェアであり，それをサービスアプリケーションから利用するためのインタフェースが設計・規格化されている．RoIS Framework で規格化されている HRI コンポーネントの一覧は表 1.2.1 に示した通りである．

HRI エンジン　HRI エンジンは，1 つ以上の HRI コンポーネントあるいは HRI エンジンを統括するプログラムであり，ロボットあるいはその集団を 1 つの単位として構成される．HRI エンジンに階層構造を持たせることで複数台のロボットの操作を可能としている．HRI エンジンは後述するサービスアプリケーションと HRI コンポーネントの間で情報の仲介を

する役目を持つ.

サービスアプリケーション サービスアプリケーションは，RoIS Framework の中で最もレイヤの高い部分で，タスクの進行をコントロールする役目を持つ．サービスアプリケーションは HRI エンジンと介して HRI コンポーネントに動作を要求するほか，HRI コンポーネント の実行結果をもとに次の要求をするなどの機能をもつ．つまり，サービスアプリケーションは action の組み合わせや処理手順から 1 つの behavior を記述するためのプログラムと言える.

ここで，RoIS Framework のシステムの全体像を図 2.4.1 に示す．本稿では，図 2.4.1 中の灰色の破線より下の層のシステムを「ロボットシステム」と呼ぶこととする.

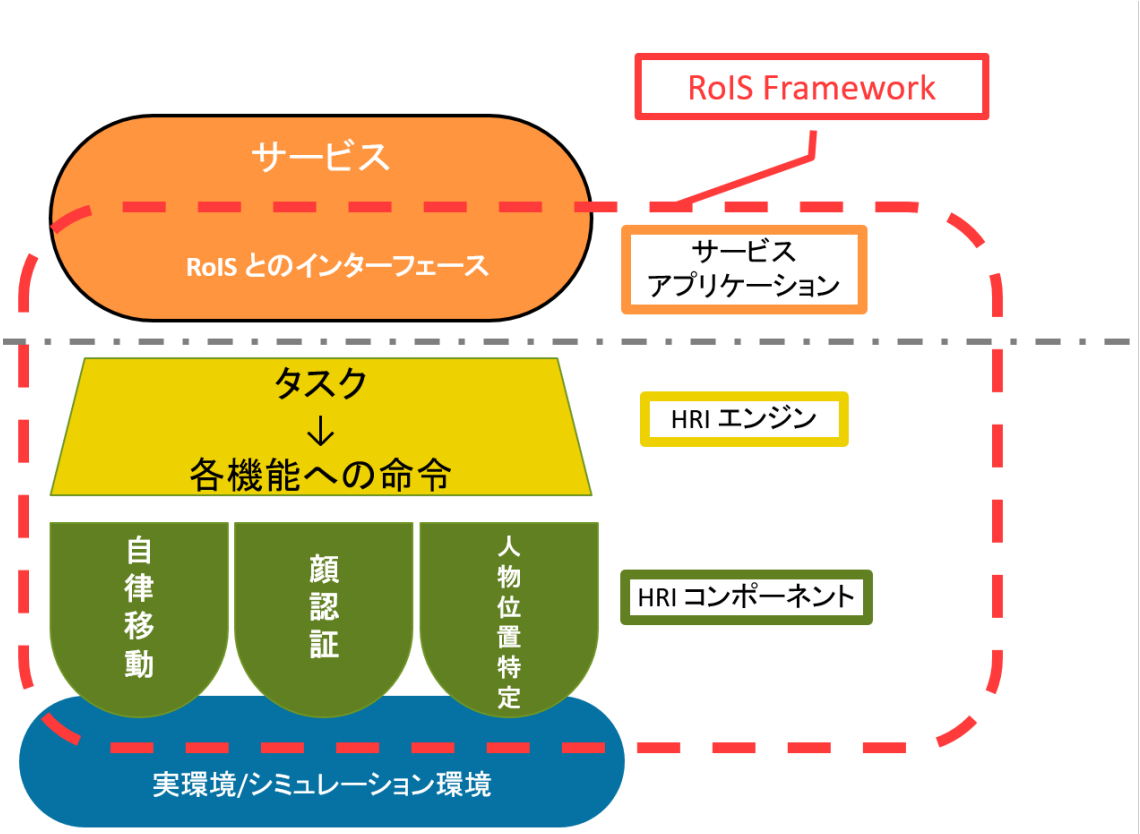

図 2.4.1: RoIS Framework のシステムの全体像

HRI コンポーネントと HRI エンジンの間や HRI エンジンとサービスアプリケーションの間においては，RoIS Framework で規格化されたインタフェースに則って通信を行う．ここで，HRI エンジンと HRI コンポーネント間の互換性については仕様書で記載されてお

らず，あくまでサービスアプリケーションとそれ以下のレイヤの間で互換性が保たれていればよいとされている．コンポーネントそれぞれのインタフェースやイベント等の，RoIS Framework の詳細な仕組みについては [14] を参照されたい．

2.5 仕様要求

ロボットサービスの構築に RoIS Framework を利用する最大の有用性はサービスアプリケーション-ロボットシステム間での互換性が保たれることにある．互換性を保つためには，サービスアプリケーションの実装に変更を加えることなくロボットの台数や動作環境の変化に対応できる必要がある．

また，実用的なロボットサービスへと拡張しても互換性が保たれることを示すために，ロボットサービスが実際に利用されることを想定して以下の 2 つの仕様を設定した．1 つ目は，サービス提供の中で柔軟な対応をするために，動的なタスクや環境の変化に対応しながらタスクを進行させるという仕様である．2 つ目は，タスク進行中にロボットが動けなくなったり，無効な入力を受けたりなどの障害について，人が手を加えることなくシステムが自動的に対処できるという仕様である．

2.6 開発ポリシー

2.5 で述べた，互換性実現のための仕様要求を満たすために，以下のような開発ポリシーのもと開発を進めた．

まずサービスアプリケーションは人物を発見する，その次にその人の位置を特定するなどといったタスクの「処理手順」のみを持つこととした．これによりサービスアプリケーションが地図やロボットの台数等を保持する必要がなくなり，ロボットとの依存を排除できた．

また，ロボットシステム固有の処理や機能については，サービスアプリケーションより下の階層で処理するようにすることとした．これによって，ロボットを変更した際のサービスアプリケーション内部への影響を排除できた．

RoIS Framework を利用したシステムの全体的な開発ポリシーとしては，サービスアプリケーションはタスクを進行するためのタスクの処理手順のみを持ち，何台ロボットがあるの

か，どのロボットが動いているかなどの情報をやり取りしない．逆に，HRI コンポーネントはインタフェース介して HRI 機能を提供するだけで，受けた指示が何のための動作なのか，ロボットサービスのどの段階にあたるのかの情報を持たない．そして，エンジンは HRI 機能単位 (コンポーネントレベル) のインタフェースの差異を吸収し，サービスアプリケーションとコンポーネントの間で橋渡しをする部分である．

なお，RoIS Framework が目指すのは，異なるロボットシステムを同一のサービスアプリケーションで動作させるための規格を定めることにあり，そのためにロボットの機能やロボットシステム，サービスアプリケーションのそれぞれの間で「要求」とそれに対する「応答」といったインタフェースを標準化を行っている．各機能の動作の定義や動作互換については考えず，機能が要求に対してどのような動作をするかについては RoIS Framework の管轄外である．

2.7 シミュレーションを利用した検証

本研究では，ロボットサービスの中で，ロボットの機能が実際に動作する部分である，HRI コンポーネントの動作を確認するために，シミュレーションを利用した．これによって，RoIS Framework のサービスアプリケーションからの自律移動の要求を Navigation コンポーネントまで伝達し，その要求通りに実際にロボットが自律移動する様子を確認できる．また，シミュレーションで確認できるのは HRI コンポーネントの動作のみであり，サービスアプリケーションの実装に影響が及ばない．そのため，サービスアプリケーションの互換性は，同じサービスアプリケーションを動作させながら，それ以下のロボットシステムを変更することで確かめられる．サービスの再現性の確認では，ロボットをマウス操作で倒したり，パラメータ調整でロボットの配置を変えるなどの操作を容易に行える．また，ロボットシステムとシミュレーション環境の構築には，後述する ROS と Gazebo を使用しているため，本研究で実装したロボットサービスシステムを実機に転用することが可能であり，今後の研究にも応用できると言える．

2.8 まとめ

本章では，まず RoIS Framework 開発の背景にあるロボットの運動レベルの概要を説明をした．次に，関連研究との比較を行い，本研究の新規性や研究の意義を示した．次に RoIS Framework の構造とそこに発生する仕様要求，仕様要求を満たすための開発ポリシーについて述べた．また，シミュレーションで検証をおこなう理由についても述べた．

第3章　ロボットサービスの
シミュレーションと互換性の評価

3.1　まえがき

本章では，初めにロボットサービスの互換性を検証するために構築した具体例としてのロボットサービスについて述べる．次に，上で述べたロボットサービスと，それをシミュレートするシステムの実装について述べる，最後に，ロボットシステムを変化させたときの，実装したサービスアプリケーションの互換性について，実験・評価する．

3.1.1　目的

RoIS Framework はロボットサービスにおいて，同一のサービスアプリケーションでハードウェアやソフトウェアの実装が異なるロボットを操作するためのソフトウェアの規格であり，サービスアプリケーションとロボットシステム間における互換性を保つように設計されている．そのため，理論上はロボットの台数やロボット間の連携の有無によるシステムの違いがあっても同一のサービスアプリケーションを用いてタスクを遂行できる．しかし，RoIS Framework の仕様書では，それを実現する手法について明示されておらず．実際に互換性が保てるのかが不明確である．

そこで，同一のサービスアプリケーションをロボットの台数の異なるロボットシステムで実際に動作させ，タスクを達成できるかの検証を行う．

3.1.2　ロボットサービスの内容

本研究では，ロボットサービスの具体例として「道案内サービス」を実装することにした．これには，ロボットが単体であるときと複数であるときでの挙動に変化を持たせやすい，

RoIS Framework でインタフェースが規格されている機能のみを使って実装できる，実際に利用する際のタスクの処理手順や想定されるエラーが明確であるなどといった理由がある．

サービスの大まかな内容は以下の通りである．ロボットが余裕をもって移動できる環境に，1 台あるいは 3 台のロボットが配置されているという状態で，

1. ロボットが画像認識等でユーザの位置を特定する．
2. ロボットがユーザの位置まで移動する．
3. ロボットがユーザに目的地を尋ねるなどして情報を得る．
4. ロボットが先導するようなかたちで，ユーザを目的地まで案内する．

タスク進行中に障害が発生した場合の対処は，以下の通りである．

- 2 でロボットがスタックした場合は，他の手の空いているロボットにタスクを委託する．ここで，手の空いたロボットがいなかった場合はタスク続行不可能とサービスアプリケーションが判断する．
- 3 で音声認識に失敗あるいは無効な目的地を指定された場合は，再び目的地を尋ねる．
- 4 でロボットがスタックした場合は，目的地への到達が不可能であるとし，タスク続行不可能とサービスアプリケーションが判断する．

このロボットサービスを実現するにあたって，必要と考えられる HRI コンポーネントは，System Information(自己位置の取得), Navigation(自律移動), Person Localization(ユーザの位置の取得), Speech Recognition(音声認識) である．しかし，この道案内サービスにおける機能を全て実装しようとすると，画像認識による人の検出や機械学習による音声認識など，本研究の目的から逸脱した技術分野まで網羅する必要があり，作業量が過多となってしまう．そこで，本研究では，実装するロボットの機能を絞り，それ以外の機能は疑似的に実装することでこの問題に対処した．例えば，Speech Recognition では，音声の処理・解析などはせずに，音声認識後に結果として得られる文字列を出力するようにして，音声認識の機能を再現した．この文字列はターミナルから手入力で設定できるようにして，文字列の内容や時間間隔を動的に変更できるようにした．

機能	中身
System Information	ROS の acml パッケージを利用
Navigation	ROS の Navigation Stack に沿って実装
Person Localization	ダミー (人のモデルの自己位置推定結果を利用)
Speech Recognition	ダミー（ROS トピックから文字列を取得）

表 3.1.1: 各コンポーネントの実装方法

本研究で構築するロボットサービスにおける，必要な機能 (HRI コンポーネント) とその実現のためのアプローチを，表 3.1.1 に示す．

3.1.3 台数によるロボットの振舞いの違い

台数の異なるロボットシステムを実装するにあたって，単に台数だけでなく，要求に対するロボットの振舞いにも違いを持たせた．

単一ロボットの場合 ユーザに近づく，インタラクションを行う，道案内を行うなどのタスクを，1 台のロボットで遂行する．環境のどこにユーザがいたとしても，そのロボットが移動して対応するため，到着に時間がかかることがある．

複数ロボットの場合 ユーザに近づくタスクでは，そのユーザに最も近い場所にいるロボットにタスクを割り当てることで作業効率を上げることができる．また，タスク進行中に障害が発生した場合に，他のロボットにタスクを委託することでタスクを続行できる．このように，複数台のロボットを利用したほうが高効率でロバストなロボットサービスを構築できると言える．

3.2 システム構成

3.2.1 全体像

システムの全体像を図 3.2.1 に示す．大まかに分けて，サービスアプリケーション・RoIS Framework・ロボットシステムの 3 つで構成されている．サービスアプリケーションと RoIS Framework は Python で実装した．ロボットシステムは Python と，後述する ROS と

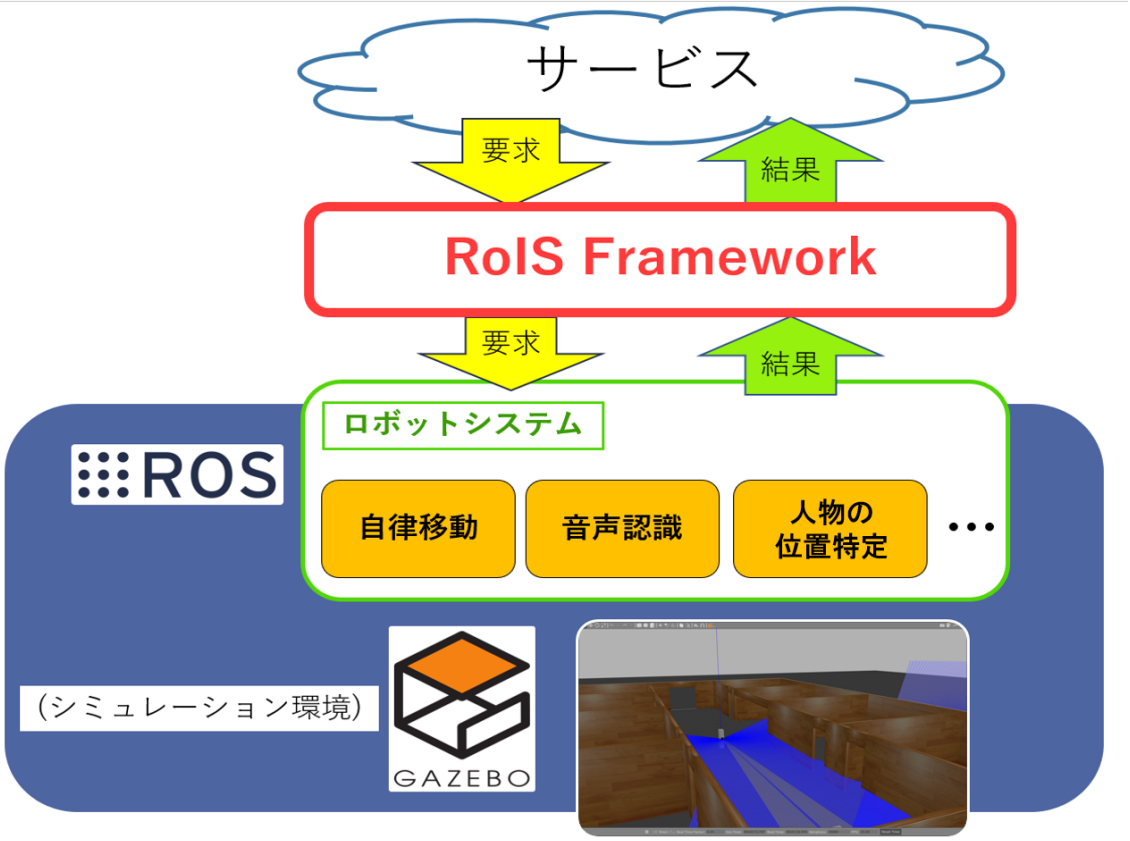

図 3.2.1: システムの全体像

Gazebo を利用して実装した．動作検証の簡易化のため，図 3.2.1 の下方の，よりレイヤの低い層から順に実装した．

3.2.2 ROS

ROS とは，ロボットを制御するアプリケーションを作成するためのオープンソースのソフトウェアフレームワークである．ROS は分散処理方式で設計されており,「パッケージ」という単位でプログラムを配布できるため，ソフトウェアの再利用性が高い．また，ハードウェア抽象化やデバイスドライバ，ライブラリ，データの可視化ツール (Rviz)，メッセージ通信，パッケージ管理などのためのツールが提供されており，ロボットプログラミングが容易となる．

本研究では，自己位置推定や自律移動，ユーザの位置の特定など，ロボットの機能の大部分を ROS を用いて実装した．自己位置推定や自律移動などの多くの機能は既存のパッケージを再利用した．なお，本稿でそれらのアルゴリズムについては言及しない．

以下に ROS の用語について簡単な説明を記す．用語の説明については，銭 飛氏の著書「ROS プログラミング [15]」を参考にした．

ノード　ROS システムにおけるノードとは，処理機能の基本単位で，処理プロセスを表す．複数のノードがそれぞれ他のノードとメッセージをやりとりすることで，プロセス間のデータ通信を可能とする．ノードは，ROS のクライアントライブラリを利用することで，C++ や Python などといった言語で実装できる．

パッケージ　パッケージとは，ROS におけるモジュールの基本単位である．パッケージにはノードのソースファイルや設定ファイル，launch ファイル（起動スクリプト）等が含まれており，パッケージを単位として再利用される．

トピック通信　トピックとは，ROS システムにおいてノード間で通信するメッセージの実体を指す．ノードがトピックを生成することをパブリッシュ（配布），トピックを受け取ることをサブスクライブ（購読）をいう．各トピックには，データ型と名前の 2 つのメタデータが付与されており，このデータをもとにノード間のメッセージ通信を実現する．

例えば，cv_camera [16] というパッケージの cv_camera_node というノードは，カメラデバイスから画像データを取得し，sensor_msgs/Image という型の ĩmage_raw という名前のトピックをパブリッシュする．また，各ノードがパブリッシュ/サブスクライブするト

ピックの名前はノードの起動時に変更できる．

Rviz　ROS では Rviz というデータ可視化ツールが提供されている．Rviz は，センサからのデータや，ロボットのモデル，各ジョイントの座標，自律移動の際の経路などの様々なデータを可視化できるツールで，本研究においてもロボットのモデルの作成や自律移動機能の実装をするにあたって Rviz を用いて動作確認やデバッグをおこなった．

3.2.3　Gazebo

Gazebo[17] とは，ロボットシミュレーションのためのオープンソースの 3D シミュレータである．Gazebo は，ODE や Dart などといった物理エンジンを選択して利用できる他，カメラや LRF (Laser Range Finder) などといったセンシングのシミュレーションが可能，ROS 用のプラグインが用意されており，ROS との親和性が高いというなどといった特徴がある．本研究において，ロボットシミュレータに Gazebo を選択した理由は，ROS との親和性が高く，開発を効率化できると考えられたからである．ROS と Gazebo を連携させたロボットの動作のシミュレーションには，既に確立された手法が存在するため，実験環境を整える時間を削減できた．さらに，ROS を利用して実装されたロボットシステムは，Gazebo 上で実験するためのプログラムと実機で動作させるためのプログラムを共通化できるため，ROS を利用していないものと比べてプログラムの再利用性が高い．

3.3　シミュレーション環境の構築

開発における作業内容を，その手順に沿って述べる．

3.3.1　モデルの作成

ロボットサービスをシミュレートするのに必要な屋内のレイアウトとロボットのモデルを作成した．以下に屋内のレイアウトとロボットをモデリングするにあたっての，それぞれのコンセプトと作成方法について述べる．

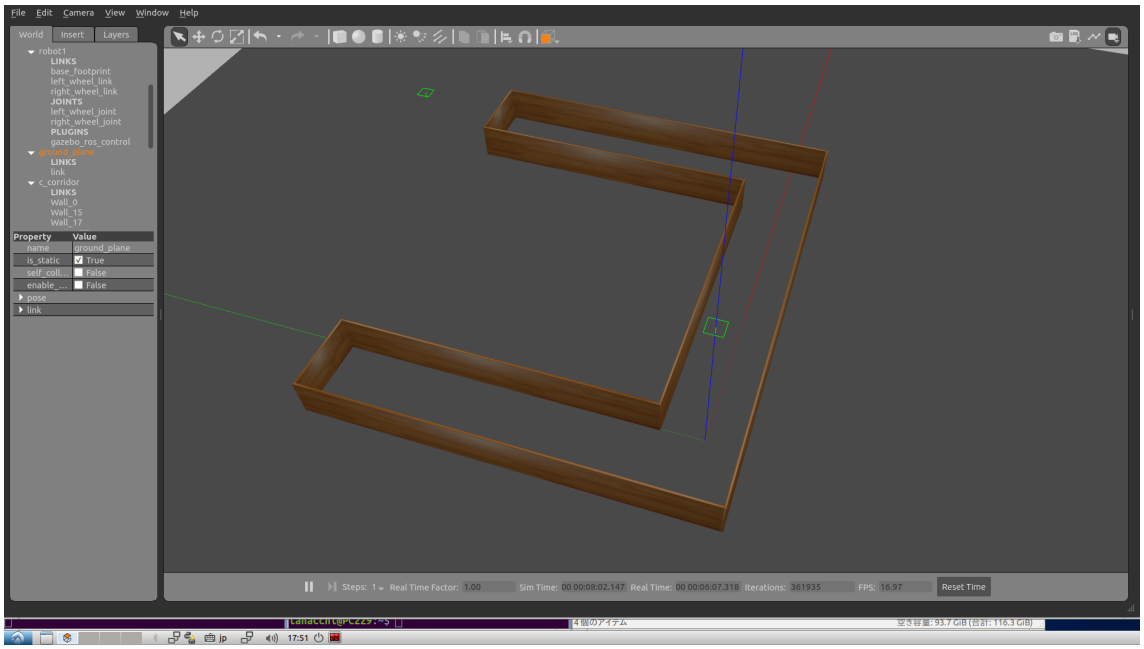

図 3.3.1: コの字型の環境

3.3.1.1 屋内のレイアウトのモデル

コンセプト 屋内のレイアウトのモデルは，シミュレーションによるロボットサービスの再現性を確認するために 2 種類作成した．

1 つ目は，図 3.3.1 に示すコの字型の構造のモデルである．このモデルでは、ロボットが余裕をもって移動できるシンプルな構造にすることで実験や動作確認を行いやすくすることと，3 つの区画を作ることでロボットやユーザの配置を変えた時の動作に変化を生じさせることを狙っている．

2 つ目は，図 3.3.2 に示す廊下と 6 つの部屋に分かれた，より複雑な環境である．これは九州工業大学戸畑キャンパス教育研究 3 号棟 4 階の廊下や部屋を参考にしたモデルで，現実に存在する環境でのロボットの動作を検証をするために作成した．

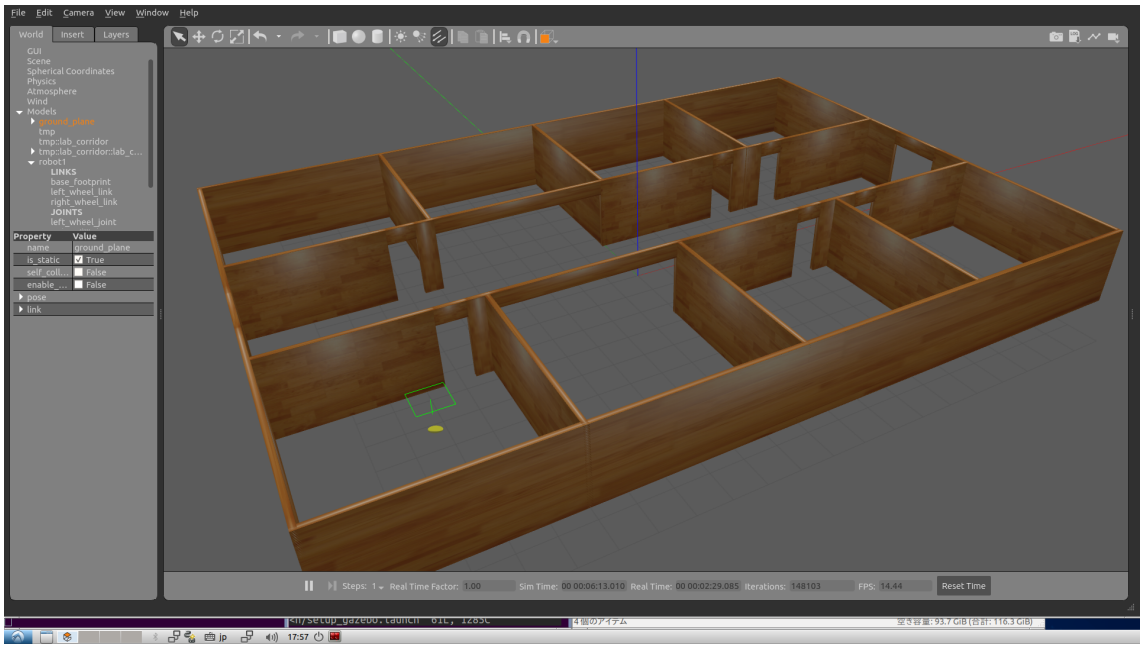

図 3.3.2: 実際の屋内を参考にした環境

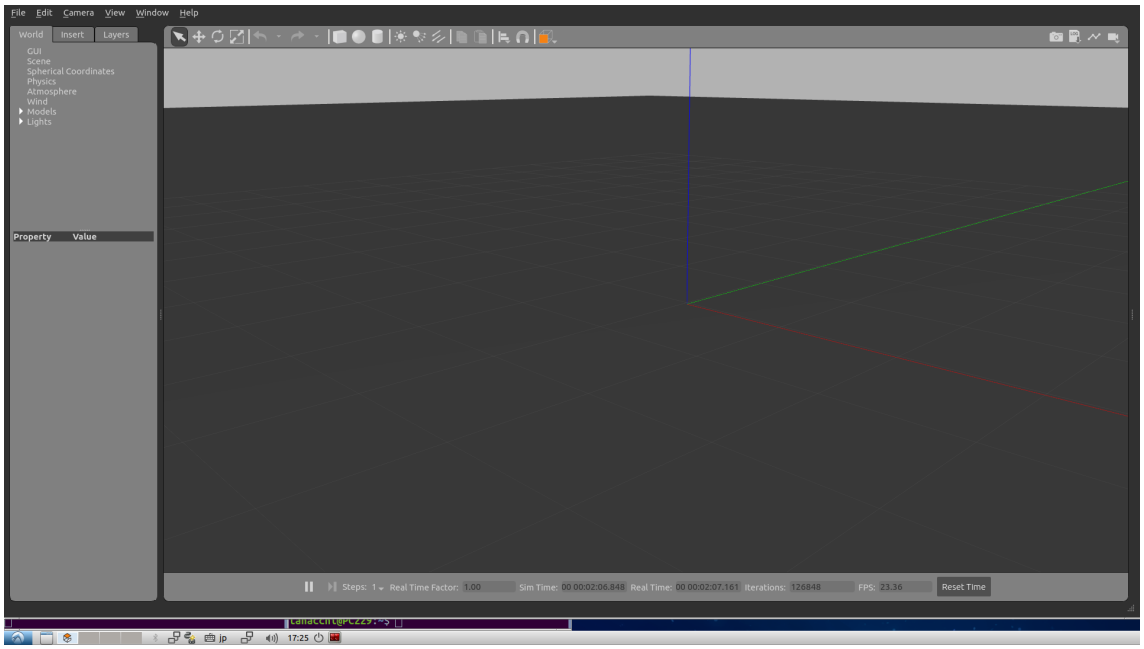

図 3.3.3: Gazebo の起動画面

作成方法　コマンドプロンプトで

```
$ gazebo
```

あるいは

```
$ roslaunch gazebo_ros empty_world.launch
```

を実行して Gazebo を起動する．

右上のタブから Edit ⇒ Building Editor を選択し，Building Editor を起動する．

上側の画面に壁や扉などを設置していくことで環境のパーツを組み立てながら，下画面でプレビューを確認できる．マウス操作で上画面で壁や扉の位置などを調整するだけでなく，長さや厚さを数値で指定したり外観を変えたりできる．屋内のモデルを作成する様子を図 3.3.4 および図 3.3.5 に示す．

編集が終わったら右上のタブから File ⇒ Save を選択してモデルを保存する．

以上の作業で実際に作成した環境のモデルの概略図を図 3.3.6 および図 3.3.7 に示す．

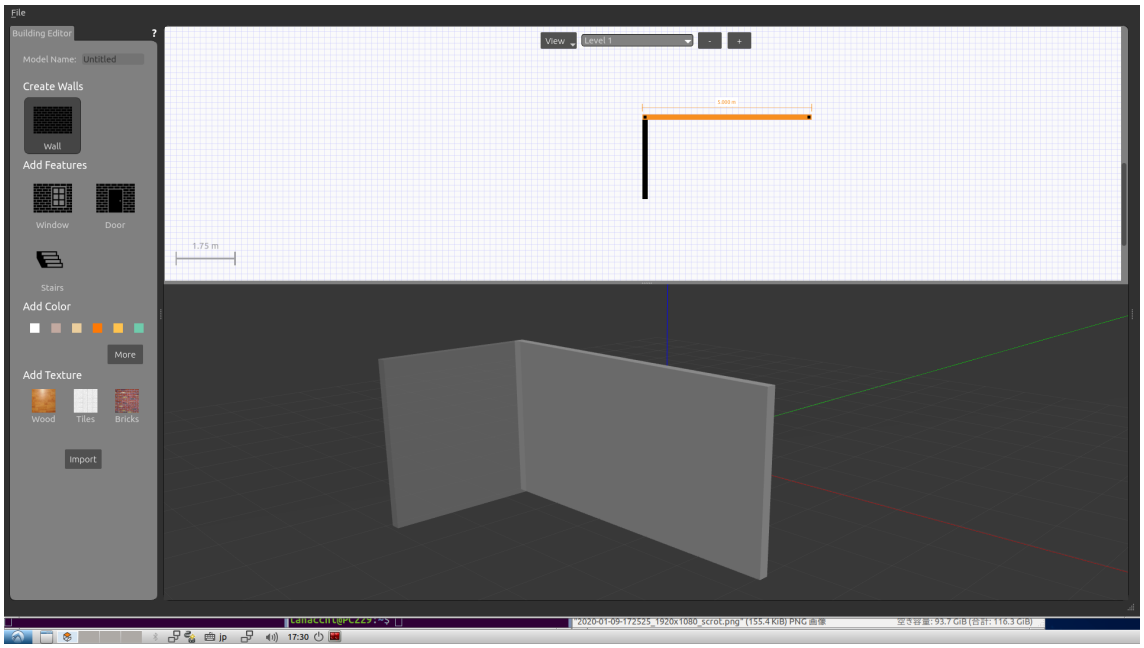

図 3.3.4: モデル作成の様子 (1)

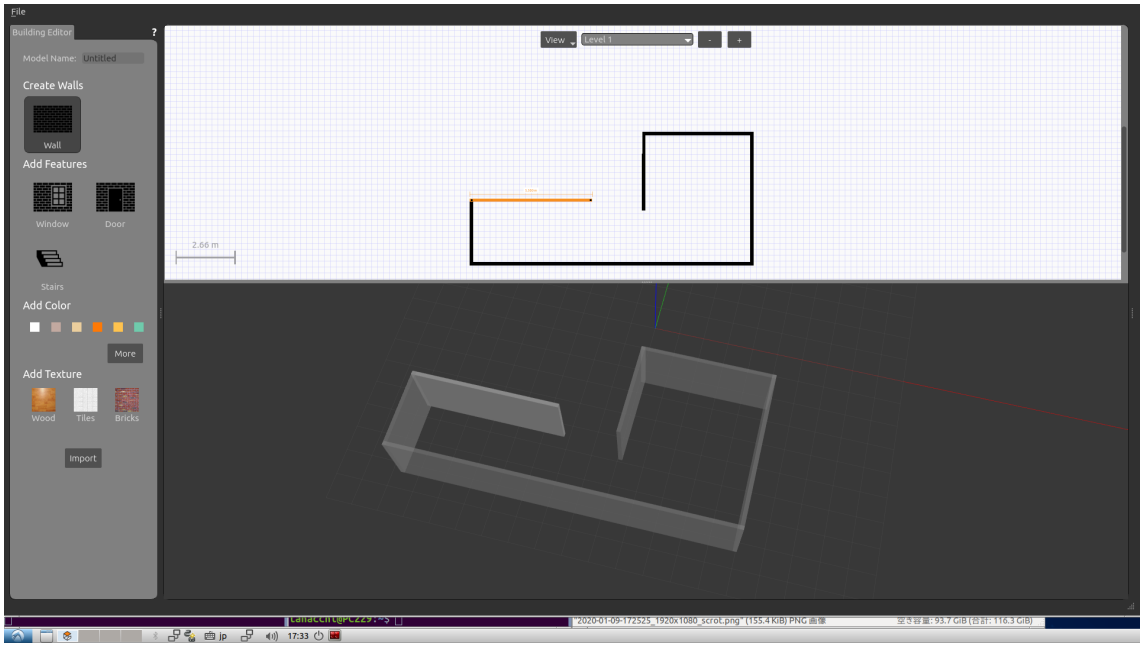

図 3.3.5: モデル作成の様子 (2)

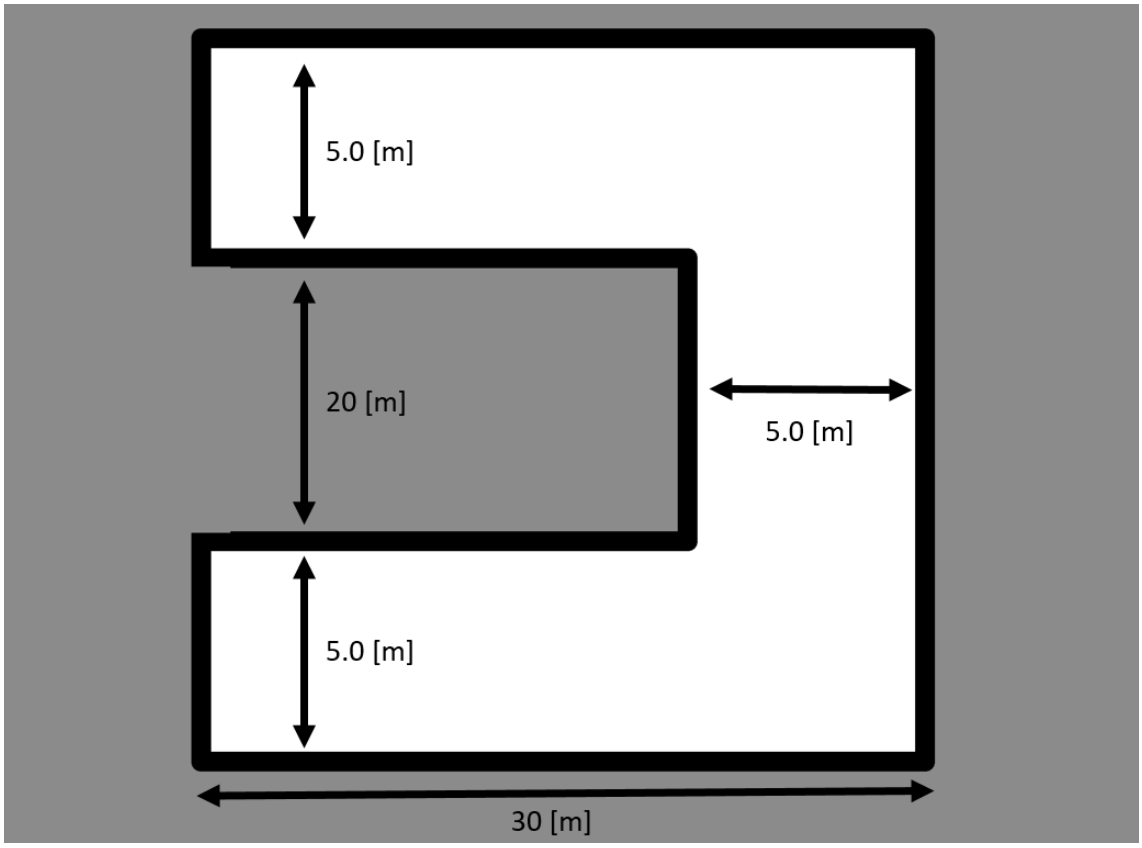

図 3.3.6: コの字型の環境の概略図

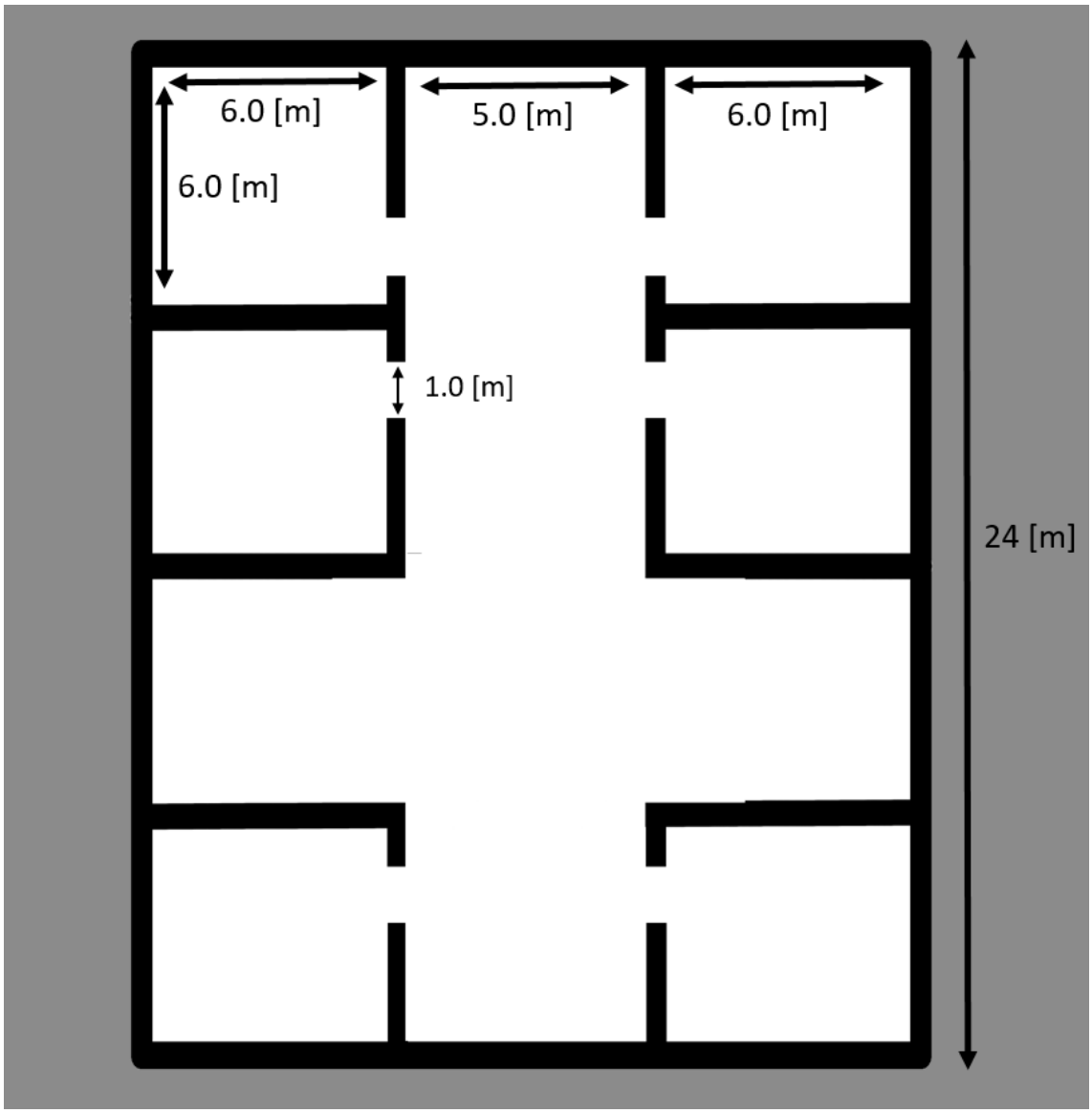

図 3.3.7: 実際の屋内を参考にした環境の概略図

3.3.1.2 ロボットのモデル

コンセプト ロボットのハードウェアのモデルは，実験中に計算機にかかる負荷を考慮して，本研究の道案内サービスにて必要となる最小限の機能だけを持たせることとした．また，サイズを含めた外観は avatar-in (ANA ホールディングス) で開発されたアバターロボット「newme」[18] を参考にした．

ロボットの駆動系は，前方の 2 つの車輪による独立二輪駆動とし，補助として後方にキャスタを設けた．センサは二次元測域センサ (以下 LRF) をロボットの前方に 1 つだけ設置した．また後述する URDF ファイルに，パーツの属性やパラメータを記述することで，LRF としての機能を持たせることができた．ロボットの外観は，人が立った状態でサービスを受けることを考慮して高さ約 1m，幅と奥行きはそれぞれ約 0.3m である (車輪幅含む)．実際に作成したモデルを図 3.3.8 および図 3.3.9 に示す．図 3.3.8 は，ロボットを Gazebo のシミュレーション環境に投影した画面である．図の中央の縦長の直方体がロボットの胴体部分で，下部に付いている赤と緑の円柱が車輪，胴体の中央付近にある青の立方体は LRF のモデルである．この LRF の可視領域は正面を 0 [rad] として $-\frac{3}{4}\pi \leq \theta \leq +\frac{3}{4}\pi$ である．この図に見られるの青い平面は，LRF が検知している平面上の障害物の存在しない領域を可視化したものである．また，木目状の部分は屋内のモデルの壁に相当する部分である．図 3.3.9 はロボットのモデルを Rviz 上で表示した画面である．図 3.3.8 と配色が異なるが，ロボットの位置と向きはおおよそ同じである．中央の赤い直方体がロボットの胴体で，下方の 2 つの円柱が車輪である．車輪と同じ高さの後方部分に，少し見えにくいが球状のキャスタも確認できる．ロボットの胴体の中央付近にある立方体は LRF のモデルである．各パーツの中央に 3 本の棒が確認できるが，これは各パーツの座標系を表したもので，赤が x 軸，緑が y 軸，青が z 軸を表している．

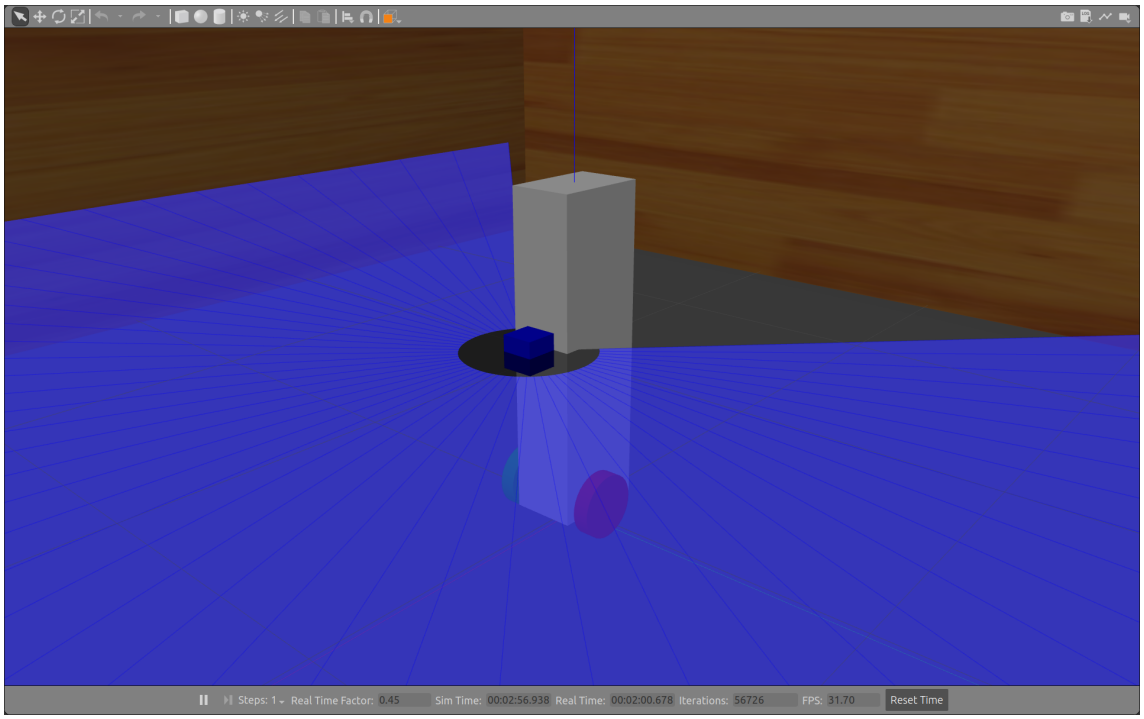

図 3.3.8: ロボットのモデル (Gazebo)

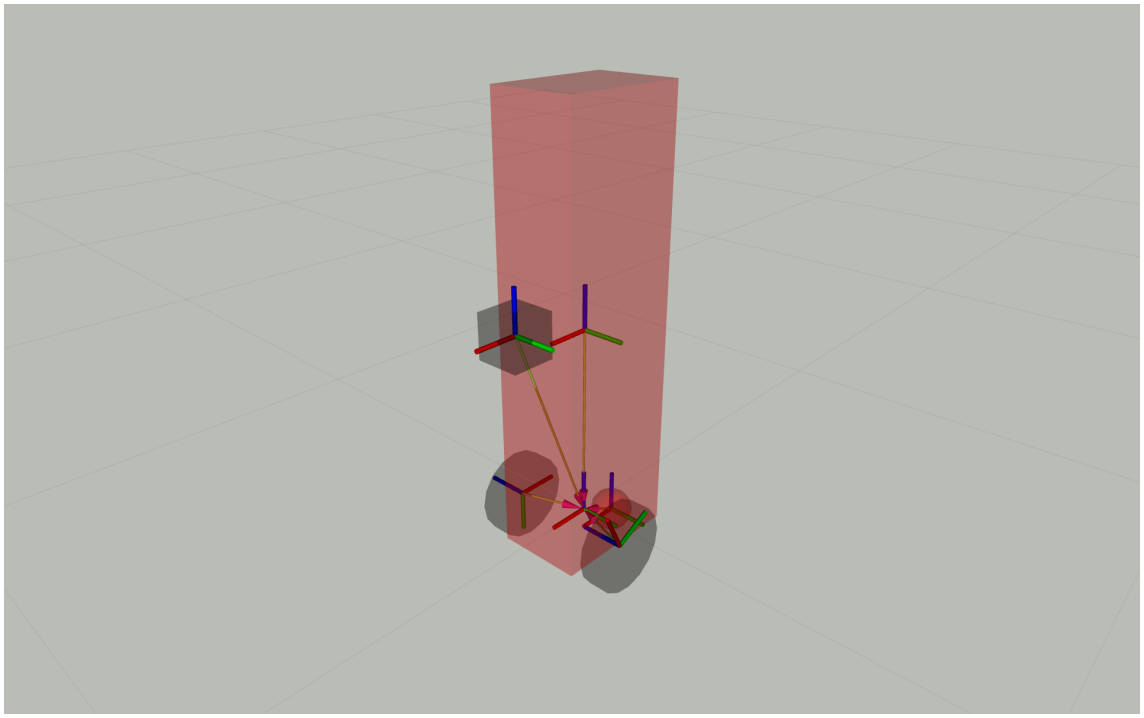

図 3.3.9: ロボットのモデル (Rviz)

パーツ名	説明	形状
base_footprint	座標系の原点	-
base_link	ボディ	直方体
lrf_link	LRF	立方体
left_wheel	左車輪	円柱
right_wheel	右車輪	円柱
caster_link	キャスタ	球

表 3.3.1: ロボットのモデルのパーツの役割と形状

作成方法

URDF　ロボットの構造を記述するために URDF ファイルを作成した．URDF[19] とは Unified Robot Description Format の略で，XML でロボットモデルを記述するフォーマットである．URDF には各パーツ (リンク) とそれらを繋ぐ接合部 (ジョイント) についてパラメータを記述する．具体的には，リンクのパラメータにはサイズや摩擦係数，センサであればその種類や周波数を記述する．ジョイントのパラメータには固定や回転といった属性や接続されるリンクなどといった設定を記述する．作成中のモデルは，

```
$ roslaunch urdf_tutorial display.launch model:=<URDF ファイルへのパス>
```

で Rviz によってモデルのパーツの位置関係を確認できる．Gazebo を利用するのも 1 つの方法ではあるが，処理が重たく起動にも時間がかかるため外観を確認したいだけであれば Rviz を利用するのがよい．

また，URDF の記述には Xacro を利用できる．Xacro [20] とは XML Macro の略であり，Xacro コマンド利用することで URDF ファイル内に変数などを埋め込むことができる．本研究においても，Xacro を用いてロボットの寸法に変数を使用し，モデルの調整において効率化できた．

以上の作業で実際に作成したロボットのモデルの外観を図 3.3.10 に，各パーツの説明と形状，寸法，位置関係を，表 3.3.1, 表 3.3.2 および表 3.3.3 に示す．

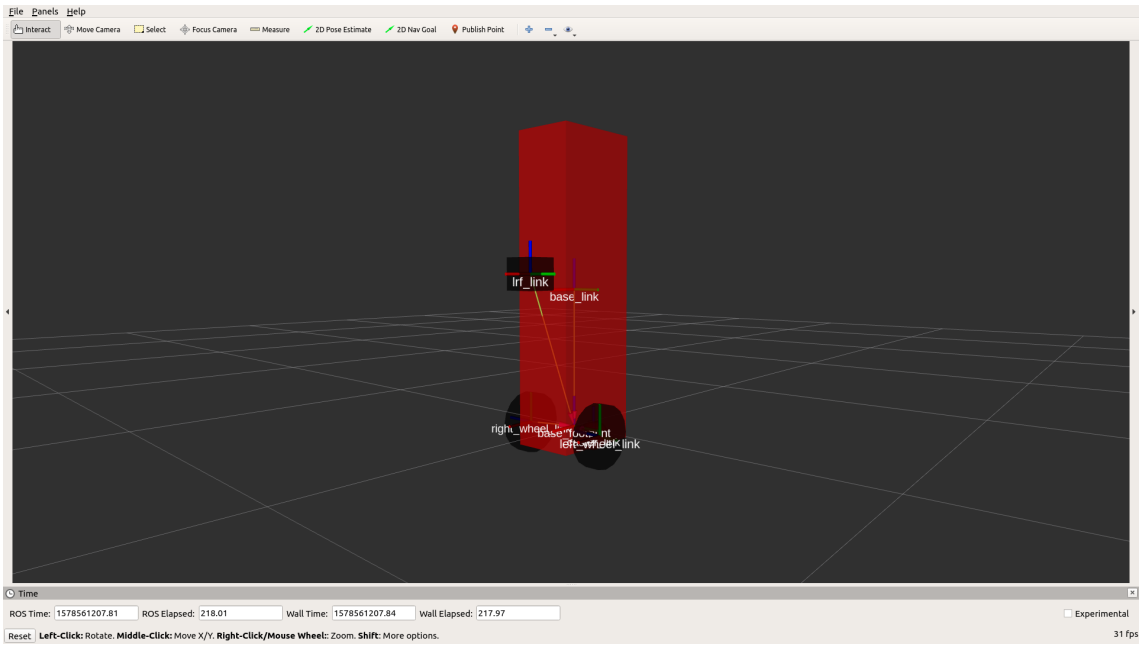

図 3.3.10: ロボットのモデルの確認：Rviz を利用してモデルを構成するパーツの位置関係を確認できる．

	寸法 [m]				
パーツ名	幅	奥行	高さ	半径	厚さ
base_footprint	-	-	-	-	-
base_link	0.20	0.30	1.00	-	-
lrf_link	0.10	0.10	0.10	-	-
left_wheel	-	-	-	0.10	0.05
right_wheel	-	-	-	0.10	0.05
caster_link	-	-	-	0.05	-

表 3.3.2: ロボットのモデルのパーツの寸法

	位置座標 [m]		
パーツ名	x	y	z
base_footprint	0.00	0.00	0.00
base_link	0.00	0.00	0.00
lrf_link	0.20	0.00	0.50
left_wheel	0.05	-0.15	0.00
right_wheel	0.05	0.15	0.00
caster_link	-0.10	0.00	-0.05

表 3.3.3: ロボットのモデルの各パーツの中心座標の位置関係

ROS と Gazebo の連携　ROS と Gazebo の連携のために ros_control[21] というパッケージの集合を利用する．これを利用して，例えば速度指令値 (ROS トピック) からシミュレーション環境上のモデルのロボットの動作へと反映させる．設定する具体的なパラーメータは，力制御，速度制御，位置制御などの選択や操作するモデルのリンク，モデルの車輪の半径，最大移動速度などがある．

3.3.2　自律移動の確立

ROS には Navigation Stack というロボットの自律移動のための枠組みが存在する．Navigation Stack は，二次元の環境地図およびセンサ情報をもとに，障害物の分布や移動経路の計算を行い，自律移動を実現するための ROS パッケージの集合である．具体的な流れとしては，

1. SLAM アルゴリズムに基づいて環境地図を予め作成
2. 環境地図やセンサ情報などを用いてロボット自身が地図上のどこにおるのかを計算 (自己位置推定)
3. 自己位置推定結果と環境地図，センサ情報を用いて
   障害物の分布 (コストマップ) を計算
4. コストマップを用いて経路を計算 (パスプランニング) し，それをもとに車輪を制御

となる．本研究では，この Navigation Stack に沿ってロボットの自律移動を確立した．

環境地図の作成　ROS パッケージの gmapping を利用して，環境地図を作成した．gmapping[22] パッケージの slam_gmapping ノードは，SLAM アルゴリズムを利用して，LRF からのデータとオドメトリ情報をもとに二次元の環境地図を作成する．ここでのオドメトリ情報とは，車輪の半径と回転角から計算した位置情報を指す．作成した環境地図は自律移動の際の (大域的な) 経路計画を行うのに利用される．実際に作成した環境地図を図 3.3.11 および図 3.3.12 に示す．障害物がない部分を白，障害物がある部分を黒，未知の部分を灰色として表現している．

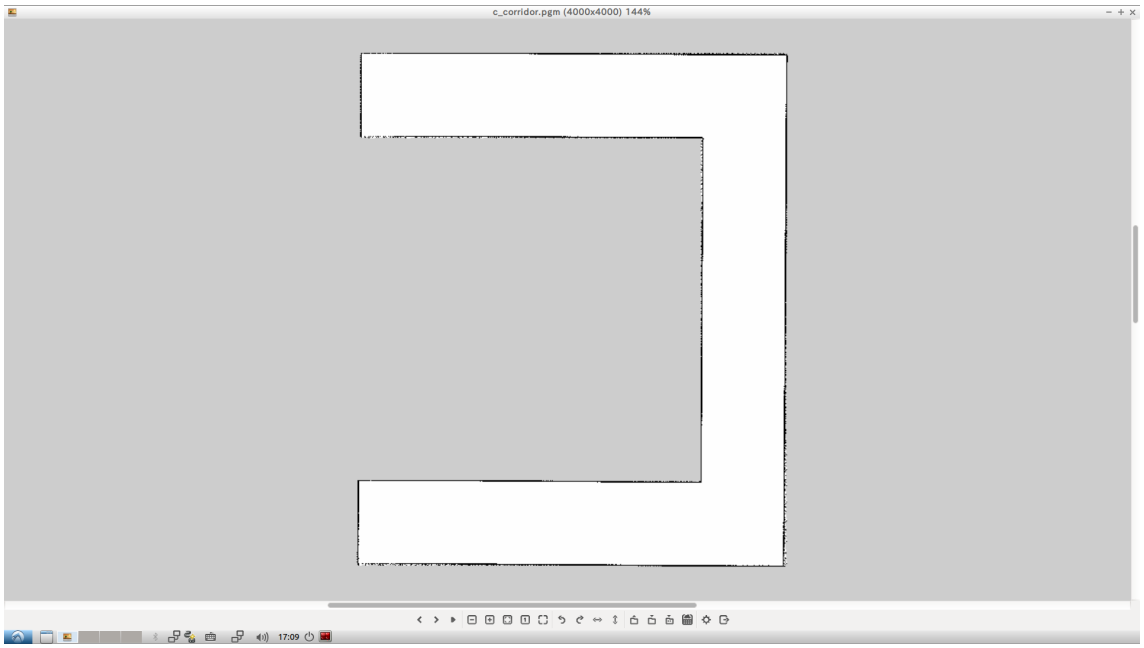

図 3.3.11: 環境地図 (コの字型)

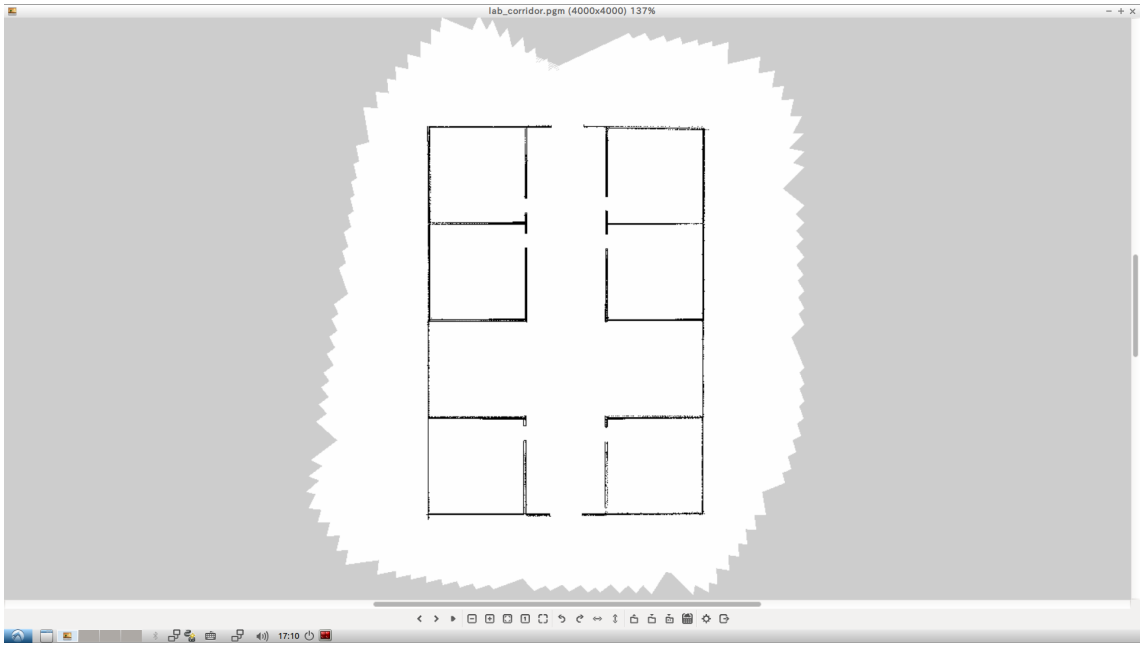

図 3.3.12: 環境地図 (3 号棟 4 階)

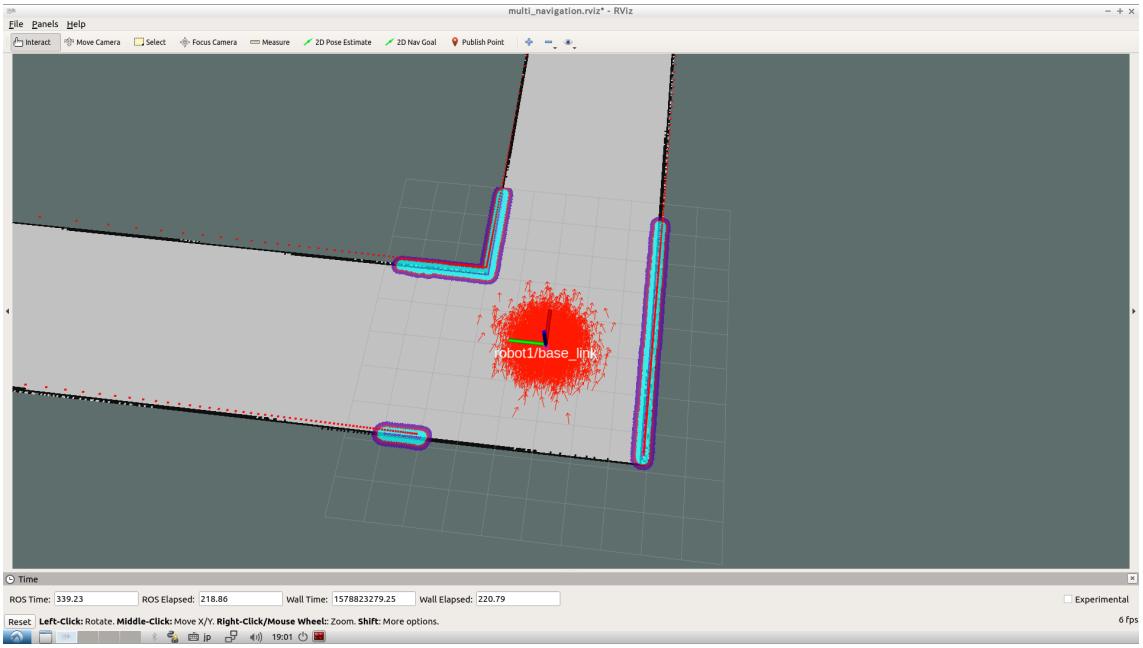

図 3.3.13: amcl による自己位置推定

amcl　ロボットの自己位置推定には amcl[23] という ROS パッケージを用いた．amcl とは，adaptive Monte Carlo localization の略で，amcl パッケージの amcl ノードは，オドメトリ情報と LRF からのデータ，環境地図などの情報をもとに，最尤推定を用いて地図上でのロボットの位置を推定する．amcl ノードによる自己位置推定の様子を図 3.3.13 に示す．図 3.3.13 では，コの字型の環境内で，図 3.3.11 に示した環境地図 とセンサデータをもとに自己位置推定を行っている．中央付近に見られる赤い小さな矢印の群は，ロボットの姿勢を推定するためのパーティクルである．

move_base　コストマップの計算，パスプランニングには move_base[24] という ROS パッケージを用いた．move_base パッケージの move_base ノードはセンサ情報，環境地図，amcl ノードにより得られた自己位置結果をもとに，コストマップを計算をする．その後，そのコストマップと目的地となる座標をもとにパスプランニングを行う．コストマップと移動経路はそれぞれ大域 (グローバル) と局所 (ローカル) の 2 種類について計算する．大域的なコストマップは環境地図をもとに計算され，大域的な経路は現在地点からゴールまでの経路である．局所的なコストマップではセンサからの情報を用いてロボット近傍の障害物を認識し，局所的な経路はそれをもとに障害物を避けるような経路として生成される．大域的な経路と局所的な経路の 2 つをもとに最終的な経路を生成する．このとき，大域的な経路

と局所的な経路のどちらを優先するかはパラメータ調整によって設定できる．move_base ノードのパラメータ調整は Kaiyu Zheng 氏の ROS Navigation Tuning Guide [25] を参考にした．

また，move_base は ROS の actionlib のサーバーをベースに実装されている．actionlib [26] は，ROS システムにおける，実行中の目標値の設定やフィードバック，実行結果の取得を実現する機能である．そのため，actionlib のクライアントを介して，自律移動の目標地点の設定や実行状況の監視を行うことができる．

3.3.2.1 複数台を自律移動させる方法

ROS システムにおけるノード名，トピック名はプログラム起動時に設定できる．また，このときに名前に階層構造を持たせることができる [27]．例えば，move_base ノードはデフォルトで /move_base という名前で実行されるが，これを /robot1/move_base という名前にして「robot1 用の move_base ノード」として扱うことができる．また，move_base ノードから出力される速度の指令値は，デフォルトでは /cmd_vel というトピック名でパブリッシュされるが，これを /robot1/cmd_vel とすることでロボット固有の速度指令値として扱うことができる．

本研究の実装においては，モデルの投影やセンシングなどシミュレーションに必要となるデータと自己位置推定，パスプランニングなど自律移動で用いるノードの名前を/robotN/* の形式で設定することで，複数台のロボットによる自律移動のシミュレーションを実現した．

3.3.3 Navigation コンポーネントの実装

Navigation コンポーネントは，RoIS Framework の HRI コンポーネントの中で，自律移動の機能に該当するプログラムである．Navigation コンポーネントサーバー内に actionlib クライアントを内包させ，それを用いて move_base ノードを操作する仕組みを実装した．これにより，RoIS Framework のサービスアプリケーションからの要求に応じて自律移動させたり，自律移動の結果を返答したりできる．

本研究では，ロボット 1 台ごとに起動する Navigation コンポーネントと別に，1 つの Navigation コンポーネントで全てのロボットの自律移動を操作する仮想の Navigation コン

ポーネントを実装した．本稿では， 単一ロボット用の Navigation コンポーネントと区別するため，Virtual Navigation コンポーネントと呼ぶ．Virtual Navigation コンポーネントは RoIS Framework で複数台のロボットを扱うための手法の 1 案として実装したもので，目的地に最も近いロボットを選択するなどのロジックをコンポーネント内に押し込められるという効果がある．詳細については考察で述べる．

3.3.4 System Information コンポーネントの実装

System Information コンポーネントはロボットそれぞれの現在地を取得するために実装された．ROS の amcl ノードからパブリッシュされた最新のロボットの位置を System Information コンポーネントのサーバー内で保持し，サービスアプリケーションや HRI エンジンから現在地取得の要求には，それを結果として応答するように実装している．上の 3.3.3 で述べた仮想コンポーネントを使用せずに「目的地に最も近いロボットに自律移動させる」という機能を実装する場合は，HRI エンジンが各ロボットの System Information コンポーネントに現在地取得を要求し，その結果をもとに目的地に最も近いロボットを HRI エンジン内で計算し，そのロボットに対して自律移動を要求する必要がある．しかし，この方法は HRI エンジンの自律移動に対する動作が固定されてしまい，HRI エンジンのモジュラリティが損うという欠点がある．本研究では，複数台のロボットの自律移動を，仮想コンポーネントを用いて制御する手法を採用したため，System Information コンポーネントは実装こそされたが，使用されていない．

3.3.5 ユーザのモデルの作成

ロボットサービスのシミュレーションを行う上で，実際の動作のデモンストレーションとロボットとのインタラクションの再現のためにユーザのモデルを作成した．外形は 3.3.1.2 で作成したロボットと同じモデルを使用し，区別のしやすさのため，赤を基調とする外観にした．実験時には，自律移動でなくキーボードからの操作でユーザのモデルを移動させるようにし，そのために ROS パッケージの teleop_twist_keyboard [28] を利用した．また，後述する Person Localization コンポーネントの実装のため，LRF と ROS の amcl ノードによる自己位置推定はロボットのモデルと同じく動作させるようにした．

3.3.6 Person Localization コンポーネントの実装

ユーザの位置を特定をする，Person Localization コンポーネントは，実際に商用利用される際は，カメラを用いた画像認識によるものが想定させるが，本研究では，実装の簡略化のために ROS の amcl ノードを利用して実装した．ユーザのモデルの位置を推定するための amcl ノードを動作させ，そこからパブリッシュされる amcl_pose トピックをサブスクライブするノードを Person Localization コンポーネントのサーバ内で動作させることで自己位置推定結果を取得するようにした．なお，Person Localization コンポーネントのインタフェースでは，人物識別した上で位置の特定を行うことも想定されているが，本研究では人物認識の機能は扱わないこととした．

3.3.7 Speech Recognition コンポーネントの実装

音声認識を行うための Speech Recognition コンポーネントは，本研究で実装するロボットサービスにおいて，ユーザから目的地を取得する目的で使用される．本研究では 3.1.2 で述べたように，この目的を達成することを第一に考えて，実際には音声認識を行わない，ダミーの Speech Recognition コンポーネントを実装した．内容としては，Speech Recognition コンポーネント内で std_msgs/String 型 の detected_text というトピックをサブスクライブするノードを起動させ，音声認識機能を動作させる際は rostopic コマンドを用いてコマンドラインからそのトピックをパブリッシュすることで Speech Recognition コンポーネント内に文字列データを渡すようにした．具体的には下のようなコマンドを入力する．

```
$ rostopic pub /robot1/detected_text std_msgs/String "data: 'point_a'"
```

3.3.8 サービスアプリケーションの実装

サービスアプリケーションは，RoIS Framework を介して ROS にタスク遂行のための要求を出すプログラムである．サービスアプリケーションは Python で実装した．RoIS のインタフェースに従って，HRI エンジンのサーバーにリクエストを送信することで，ロボット側に要求を伝達する．サービスアプリケーションはロボットシステムに依存してはならないため，ロボットサービスを実現するためのタスクの処理手順のみを保持するように実装し

た．すなわち，どのロボットが稼働中であるかや，ハードウェアや HRI コンポーネントの実装部分，利用しているミドルウェアや OS などに依存する要素を排除した．サービスアプリケーションは，ロボット側へタスクを要求し，ロボット側からタスクの実行結果を取得しながらタスクを進行させる．本研究におけるサービスアプリケーションの実装は，プロトタイプと改良版の 2 段階に分け，エラーハンドリングや動的な環境変化に対応するを加えるなど，段階的にロボットサービスの実用性を上げながら開発を行った．なお，互換性の検証で使用するのは改良版のサービスアプリケーションのみである．

プロトタイプ　プロトタイプのサービスアプリケーションでは，ロボットの機能は自律移動のみで，ユーザの位置と目的地は固定，さらに障害に対応する機能を付けないなど，限定的なものとなっている．これは，先にごく簡単な機能のみを持った道案内サービスのアプリケーションを試験的に実装し，RoIS Framework を利用したロボットサービスがシミュレーション環境上で実際に動作するのかを確認するために作成されたものである．

道案内サービスのプロトタイプのアクティビティ図を図 3.3.14 に示す．まずサービスアプリケーションがロボットに対し，目的地 (ユーザの位置) への自律移動を要求し，Navigation コンポーネントが自律移動の処理を行う．ロボットが到着したら，Navigation コンポーネントがイベントを発行し．サービスアプリケーションに通達する．それを受け取ったサービスアプリケーションは，次の目的地 (ユーザの目的地) への自律移動を要求し，同様の流れで Navigation コンポーネントが処理を行い，到着したらイベントを発行する．

改良版　実際にデパートや病院などの施設で提供できるサービスを実現するためには，タスク進行における障害やユーザからの無効な入力に対しても対応する必要がある．例えば，自律移動中に障害物などによって行く手を阻まれて，スタックしてしまったときにタスクの続行が不可能であることを検知したり他のロボットにタスクの続行を引き渡したりといった対処が必要である．あるいは，ユーザとの対話における音声認識で，期待した結果を得られなかったときに聞き返すなどとった対処が必要である．そういったエラーハンドリングは，サービスアプリケーションが担当する．また，プロトタイプではユーザの位置とユーザの目的地が固定であったが，ユーザの位置は Person Localization コンポーネント，ユーザの目的地は Speech Recognition コンポーネントを用いて取得するように改良し，動的な変化に

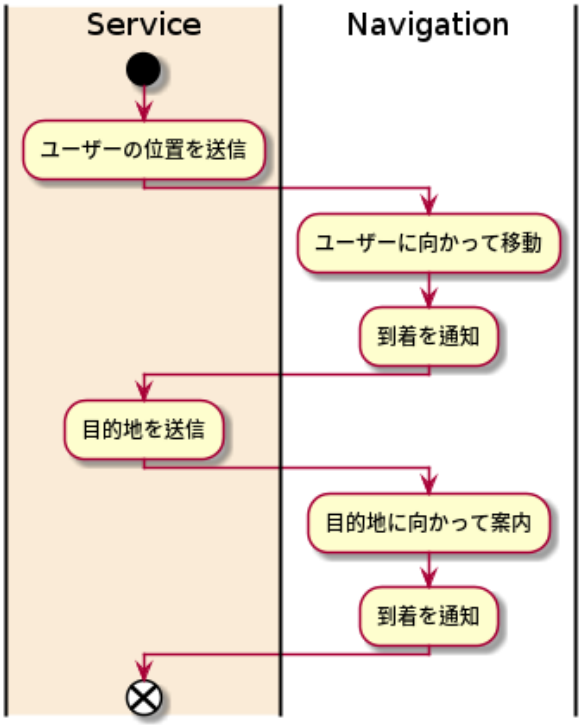

図 3.3.14: サービスシステム (プロトタイプ) のアクティビティ図

順応するサービスシステムを構築することとした．

エラーが発生することを考えないサービスアプリケーションでは，タスクの進行がシーケンシャルになるため，タスク管理の実装はそう難しくない．しかし，エラーハンドリングの機能を付与すると，タスクの実行結果によって分岐やループが発生し，実装が急激に複雑になる．そこで，サービスアプリケーションの中でタスク管理をする部分をステートマシンを利用して実装することで，実装の複雑さを軽減し，保守性を向上させた．改良版のサービスアプリケーションの状態遷移図を図 3.3.15 に示す．サービスを開始すると始めにユーザの位置の取得を行い，取得次第その場所へ自律移動を行う．この自律移動が成功したら，ユーザに目的地を尋ね，失敗した場合はもう一度ユーザの位置の取得に戻る．これはユーザの位置情報が無効であった可能性を加味するためである．また，このときスタックした，あるいは動作中であるなどで，システム内の全てのロボットが利用できない場合は，サービスアプリケーションがタスクの破棄を伝えるメッセージを出力しシステムを終了するようにしている (この動作をサービスシステムの異常終了と呼ぶ)．ユーザに目的地を尋ね，返答が来たらユーザが指定した目的地までの自律移動のフェーズに移る．ここで，目的地の指定が無効で

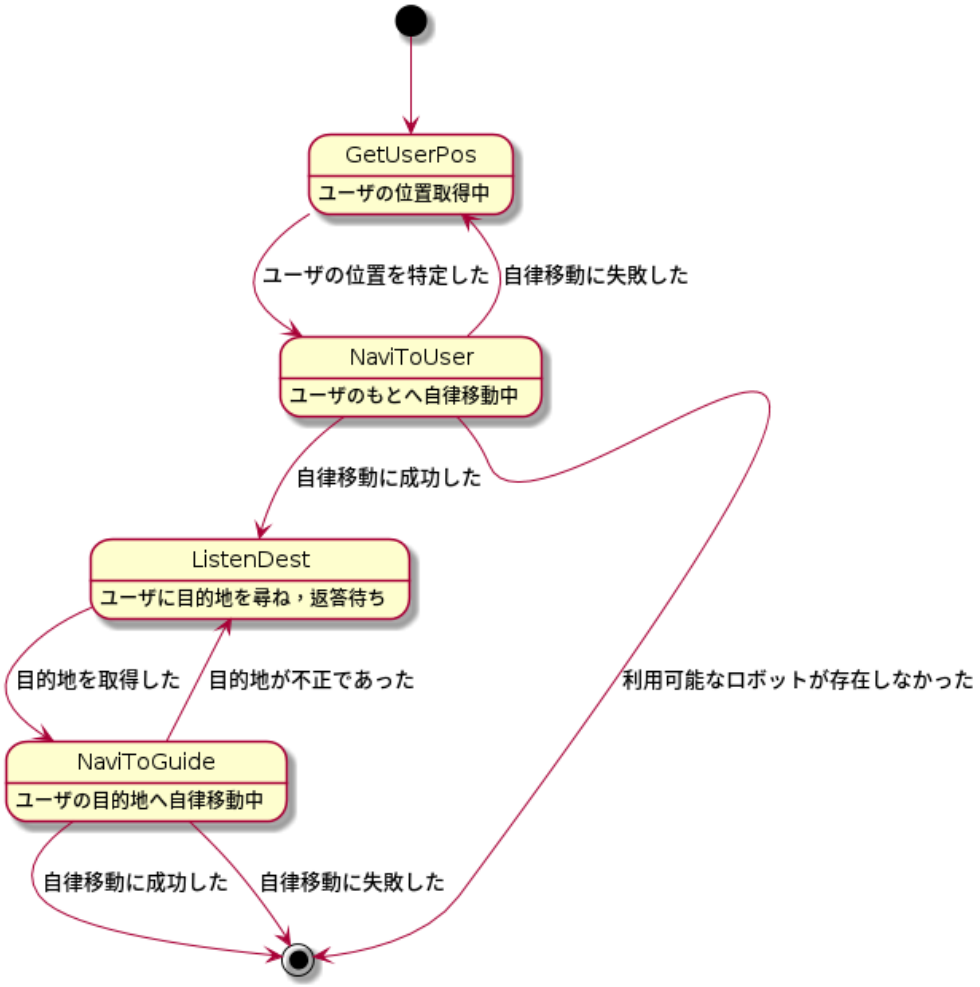

図 3.3.15: サービスアプリケーション (改良版) の状態遷移図

あった場合は，再び目的を尋ねるフェーズに戻る．目的地が正しく入力されていた場合は，その場所への自律移動を開始する．

単一ロボットのシステムによる改良版の道案内サービスのアクティビティ図を図3.3.16 に示す．まず，サービスアプリケーションがユーザの目的地の取得を指示し，Person Localization コンポーネントからのイベントを待つ状態になる．Person Localization コンポーネントは，ユーザの位置を特定したら，それをサービスアプリケーションへ通知する．次に，サービスアプリケーションはその結果をもとに，ユーザの位置への自律移動を要求する．要求を受けた Navigation コンポーネントは，先に利用可能なロボットが存在するかを確認し，無かった場合はそのことをサービスアプリケーションに通知して，サービスアプリケーション側でサービスシステムを異常終了させる．利用可能なロボットが存在した場合は，Navigation コンポーネントの処理で自律移動を開始する．自律移動に成功した，あるいは失敗したらイベントを発行し，それをサービスアプリケーションに通達する．それを受け取ったサービスアプリケーションは，失敗であればユーザの位置取得の段階まで戻り，成功であればユーザの目的地の取得タスクに移る．サービスアプリケーションは目的地の取得のために音声認識を要求し，Speech Recognition コンポーネントからのイベントを待つ状態になる．ただし，3.1.2 で述べたように，本研究で実装した Speech Recognition コンポーネントは，実際には音声認識を行わず，認識結果となる文字列を標準入力から受け取るように実装されていることに注意されたい．ここで入力される文字列は，事前に定義した実験環境上の地点の名前である．Speech Recognition コンポーネントが音声認識イベントをサービスアプリケーションに通達した後，それをもとにサービスアプリケーションは，ユーザの目的地への自律移動を要求する．このとき，Navigation コンポーネント内部で音声認識結果の文字列から対応する座標に変換するのだが，存在しない地点名が指定されていた場合は無効な入力を受けたとしてサービスアプリケーションに通知し，それを受けたサービスアプリケーションは，目的地の取得のフェーズに戻る．音声認識結果の文字列が目的地の名前として正当であった場合，その地点までの自律移動を開始する．あとは，ユーザのもとへ向かう自律移動と同様に，Navigation コンポーネントが動作を終了したタイミングでイベントをサービスアプリケーションへ通達する．

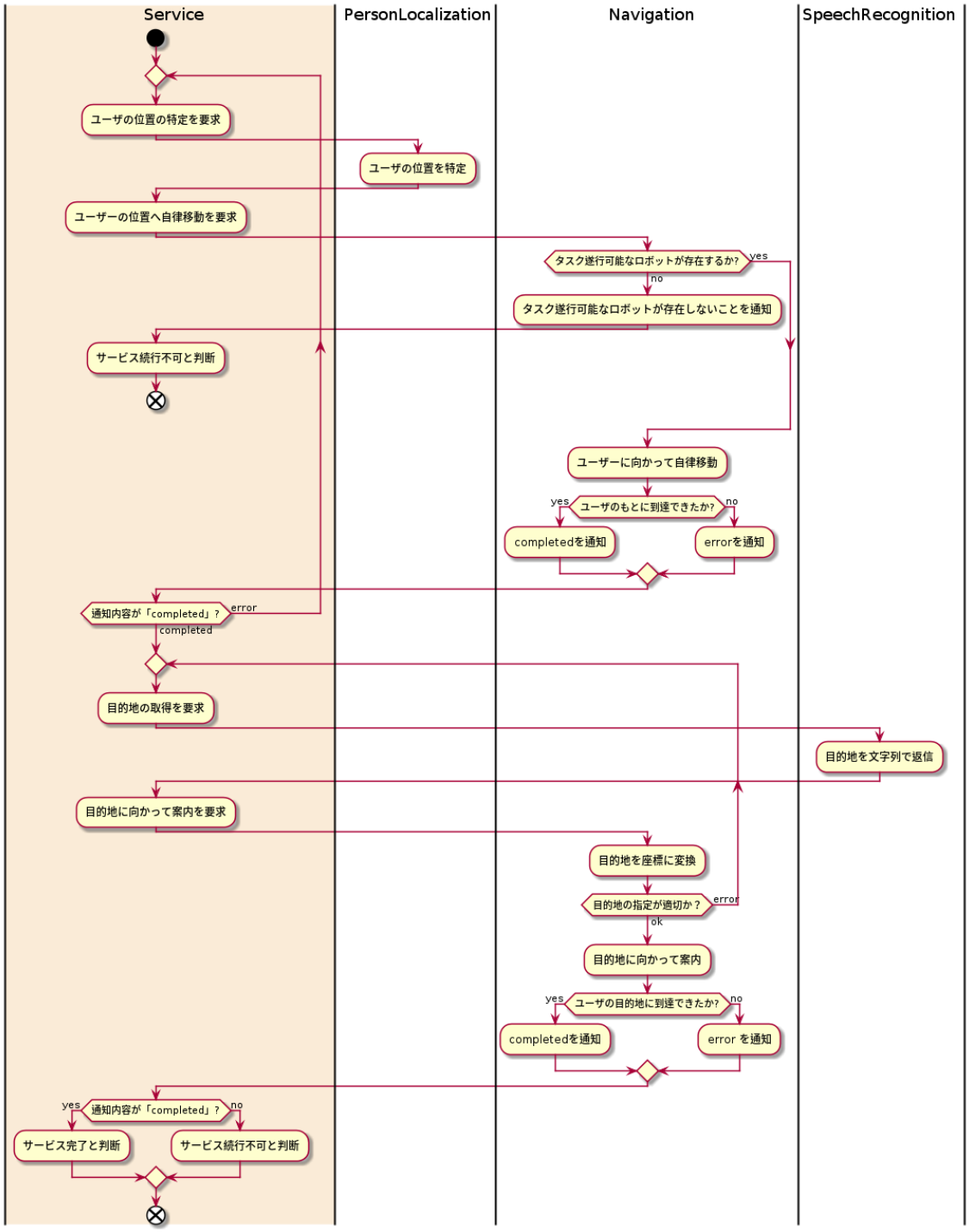

図 3.3.16: 単一ロボットによるサービスシステム (改良版) のアクティビティ図

複数ロボットのシステムによる改良版の道案内サービスのアクティビティ図を図 3.3.17 に示す．2 つの図はロボットシステムが単一ロボットの場合と複数ロボットの場合での違いで，サービスアプリケーションは同一の動作をするが，ロボットが複数になることで連携の要素が加わり，サービス提供の流れに違いが存在する．具体的には，ユーザのもとへ向かう自律移動を開始する前にスタックしているロボットを自身の初期位置に帰還させたり，利用可能なロボットそれぞれの位置情報から目的地に最も近いロボットにタスクを担当させたりといった機能が追加される．

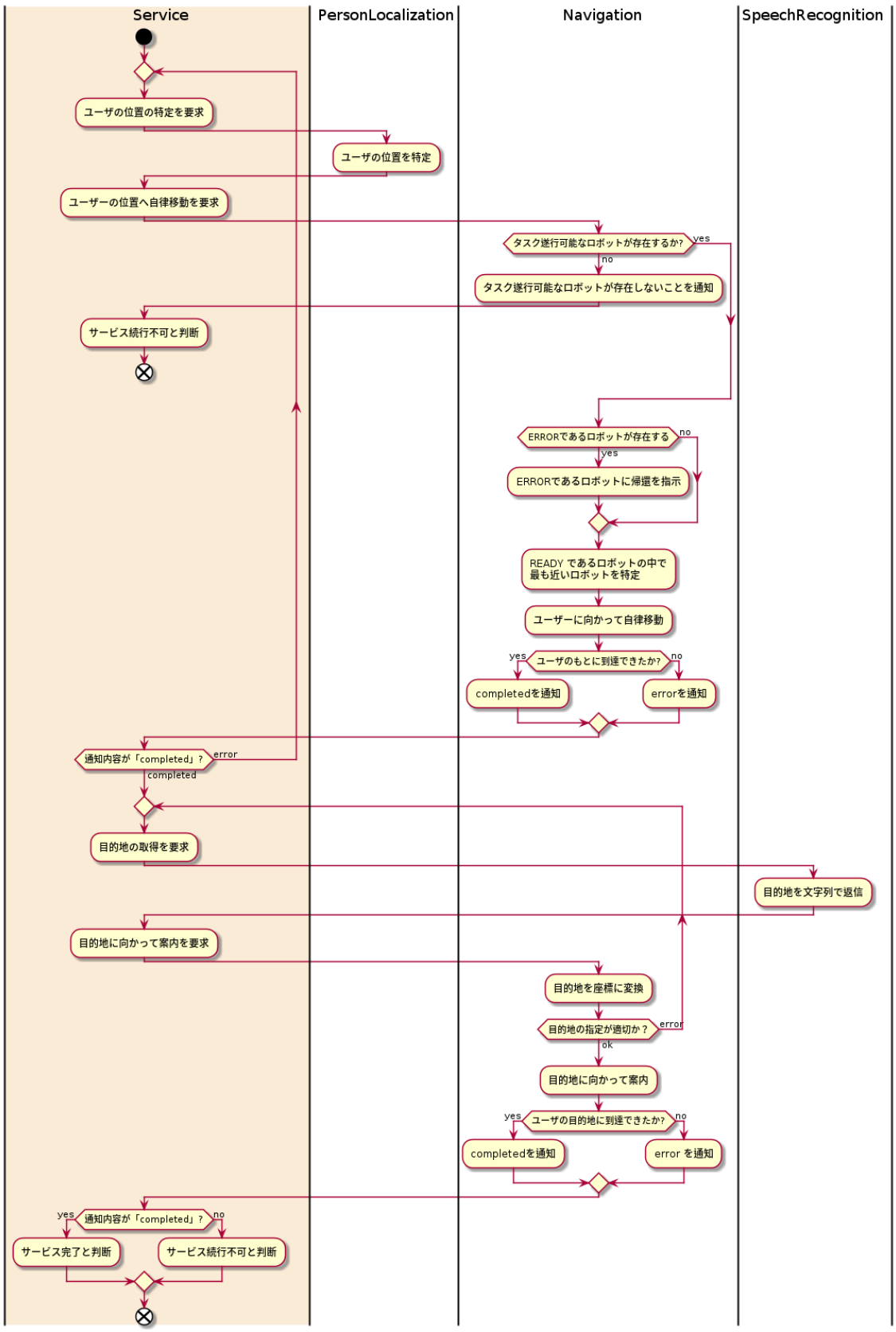

図 3.3.17: 複数ロボットによるサービスシステム (改良版) のアクティビティ図

CPU	Intel (R) Core (TM) i7 950 (3.07 GHz)
メモリ	12 GB
グラフィックカード	NVIDIA GK107 (GeForce GT 640)
ストレージ	ATA TOSHIBA THNSNH (128GB)
OS	Ubuntu 18.04.3 LTS
ROS	melodic

表 3.4.1: 実験における動作環境

3.4 実験と評価

以上の実装を経て，RoIS Framework を用いた道案内用のサービスアプリケーションおよびロボットの台数と，それによって連携の有無が異なる 2 種類のロボットシステムを構築した．本節では，RoIS Framework を用いたの異なるロボットシステム間における同一サービスアプリケーションの互換性について評価するために，実験条件を変えながら同一のサービスアプリケーションを 2 種類のロボットシステムで動作させて，ロボットサービスを遂行できるかを検証する．具体的な実験条件は以下の 6 パターンである．

1. 理想的な状態
2. ユーザの位置変更
3. ロボットの配置変更
4. 屋内のレイアウトの変更
5. 自律移動中の障害発生
6. 無効な目的地の入力

実験における動作環境を表 3.4.1 に示す．

実験の手順を以下に示す.

単一ロボットの場合の実験手順

1. Gazebo シミュレータを起動する．ここで屋内のレイアウトとロボットのモデルをシミュレーション環境にスポーン (spawn) する．同時にモデル操作のための ROS ノード (robot_state_publisher など) も起動する.

2. ロボットの自律移動に必要な ROS ノードを起動する．具体的には，amcl ノード (自己位置推定)，move_base ノード (パスプランニング) がある.

3. 単一ロボット用のロボットシステムを起動する．具体的には Navigation コンポーネント，System Information コンポーネント，Speech Recognition コンポーネント，Person Localization コンポーネントとそれらを統括する HRI エンジンを起動する.

4. ユーザのモデルをシミュレーション環境にスポーンする．同時にモデル操作のための ROS ノード と amcl ノードを起動する.

5. サービスアプリケーションを起動する.

複数ロボットの場合の実験手順

1. 単一ロボットの場合と同様に，シミュレーション環境を起動する．複数ロボットの場合は 3 台のロボットモデルのスポーンし，モデルごとにモデル操作のための ROS ノードも起動する.

2. ロボットの自律移動に必要な ROS ノードを起動する．amcl ノード，move_base ノードはロボットそれぞれについて起動する.

3. 複数ロボット用のロボットシステムを起動する．複数ロボットの場合は，3 台のロボットそれぞれについて Navigation コンポーネント，System Information コンポーネント，Speech Recognition コンポーネントとそれらを統括する HRI エンジン (サブエンジン) を起動し，それに加え Person Localization コンポーネントと 3 台分のサブエンジンを統括する HRI エンジン (メインエンジン) を起動する.

出力文字列	色	説明
Starting service..	白	サービスアプリケーション正常に起動した.
GetUserPos started.	青	ユーザの位置取得中.
person localized successfully.	緑	ユーザの位置取得に成功した.
Destination is [x, y, z] (list)	白	目的地の取得結果. (座標による指定)
Destination is XXX (str)	白	目的地の取得結果. (場所の名前による指定)
NaviToUser started.	青	ユーザのもとへ自律移動中.
robot reached goal successfully.	緑	ロボットが目的地に到着した.
robot failed to reach goal.	赤	ロボットが目的地に到着できなかった.
There is no robot that is available.	赤	自律移動を開始できるロボットが存在しなかった.
ListenDest started.	青	ユーザに目的地を尋ね，返答待ち.
speech recognized successfully.	緑	音声認識による目的地取得に成功した.
NaviToGuide started.	青	ユーザの目的地まで移動中.
Bad destination detected. Trying to ask again...	黄	無効な目的地の入力を検出し， 再びユーザに目的地を尋ねようとしている.
Route Guidance Completed !	青	道案内サービスの正常終了
Route Guidance Aborted...	赤	道案内サービスの異常終了

表 3.4.2: サービスアプリケーションの出力文字列と意味

4. 単一ロボットの場合と同様にユーザのモデルをシミュレーション環境にスポーンする.

5. サービスアプリケーションを起動する.

これらの手順は 6 パターン全ての実験条件において共通である．また，単一ロボットの場合と複数ロボットの場合で起動するサービスアプリケーションは全く同じものである．また，単一ロボットの場合は robot1 のみが，複数ロボットの場合は robot1, robot2, robot3 シミュレーション環境にスポーンされる.

サービスアプリケーションはタスクの進行を標準出力するように実装されており，これを確認することでタスクの進行状況を確認できる．表 3.4.2 に，サービスアプリケーションが出力する文字列とその意味を示す.

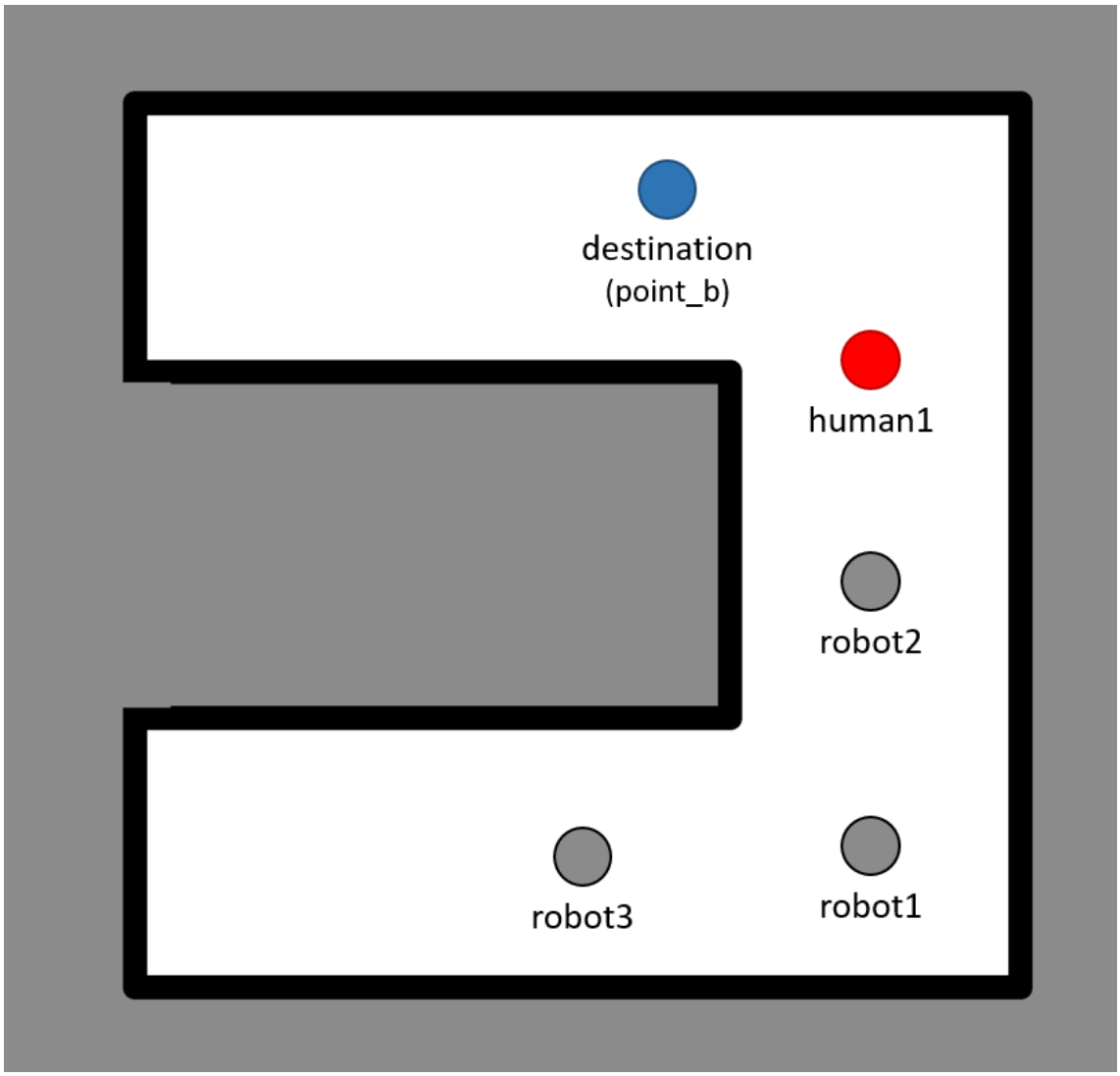

図 3.4.1: 実験条件 1 の概略図

3.4.1 実験条件 1:理想的な状態

はじめに，実装の過程での動作確認に用いたロボットおよびユーザの配置，屋内のレイアウトを用いて検証を行った．この状態は既に動作が確認できているため，このロボットシステムにおける理想的な状態であると言える．実験条件 1 のユーザの位置，ロボットの配置および屋内のレイアウトの概略図を図 3.4.1 に示す．

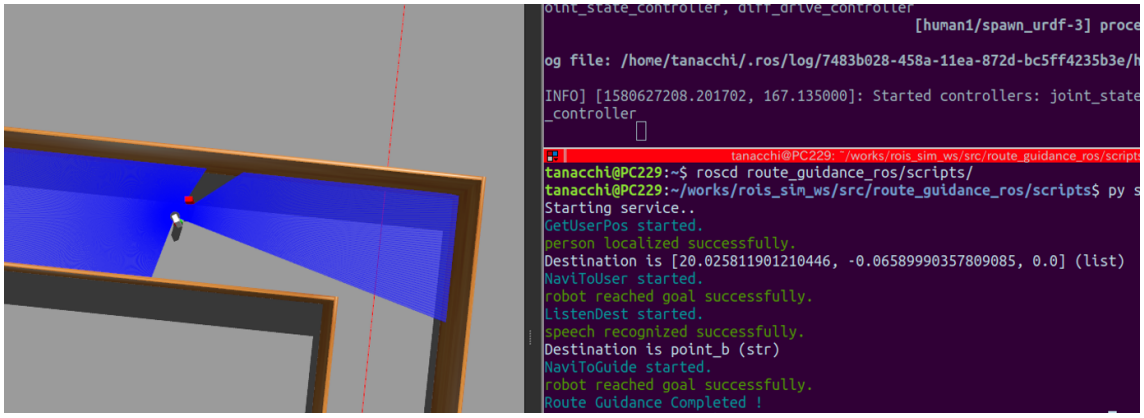

図 3.4.2: 実験条件 1 での実験の様子 (単体ロボット)

```
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$ py service_app_lv2.py
Starting service..
GetUserPos started.
person localized successfully.
Destination is [20.025811901210446, -0.06589990357809085, 0.0] (list)
NaviToUser started.
robot reached goal successfully.
ListenDest started.
speech recognized successfully.
Destination is point_b (str)
NaviToGuide started.
robot reached goal successfully.
Route Guidance Completed !
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$
```

図 3.4.3: 実験条件 1 でのサービスの実行結果 (単体ロボット)

単一ロボットの場合　実験条件 1 における，単一ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.2，サービスアプリケーションの実行結果を図 3.4.3 に示す．どちらも，全てのタスクが問題なく遂行されたことを示している．

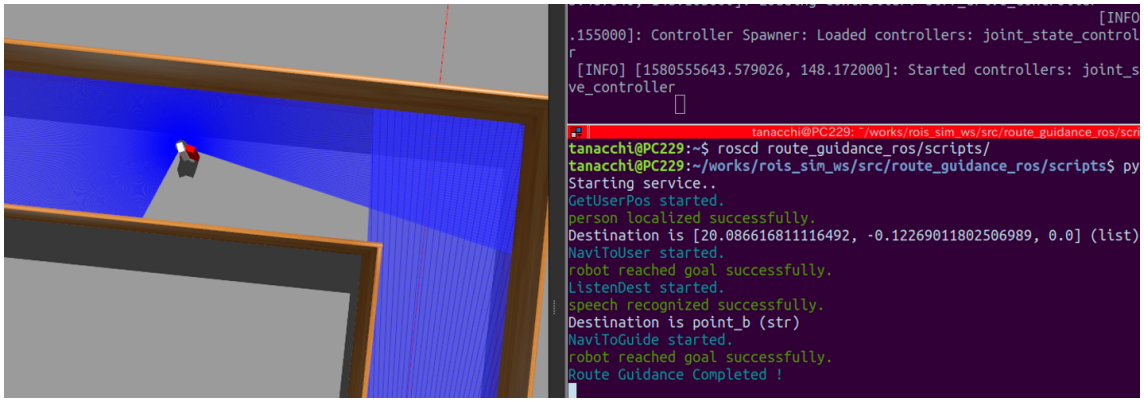

図 3.4.4: 実験条件 1 での実験の様子 (複数ロボット)

```
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$ py service_app_lv2.py
Starting service..
GetUserPos started.
person localized successfully.
Destination is [20.086616811116492, -0.12269011802506989, 0.0] (list)
NaviToUser started.
robot reached goal successfully.
ListenDest started.
speech recognized successfully.
Destination is point_b (str)
NaviToGuide started.
robot reached goal successfully.
Route Guidance Completed !
```

図 3.4.5: 実験条件 1 でのサービスの実行結果 (複数ロボット)

複数ロボットの場合　実験条件 1 における，複数ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.4，サービスアプリケーションの実行結果を図 3.4.5 に示す．どちらも，全てのタスクが問題なく遂行されたことを示している．

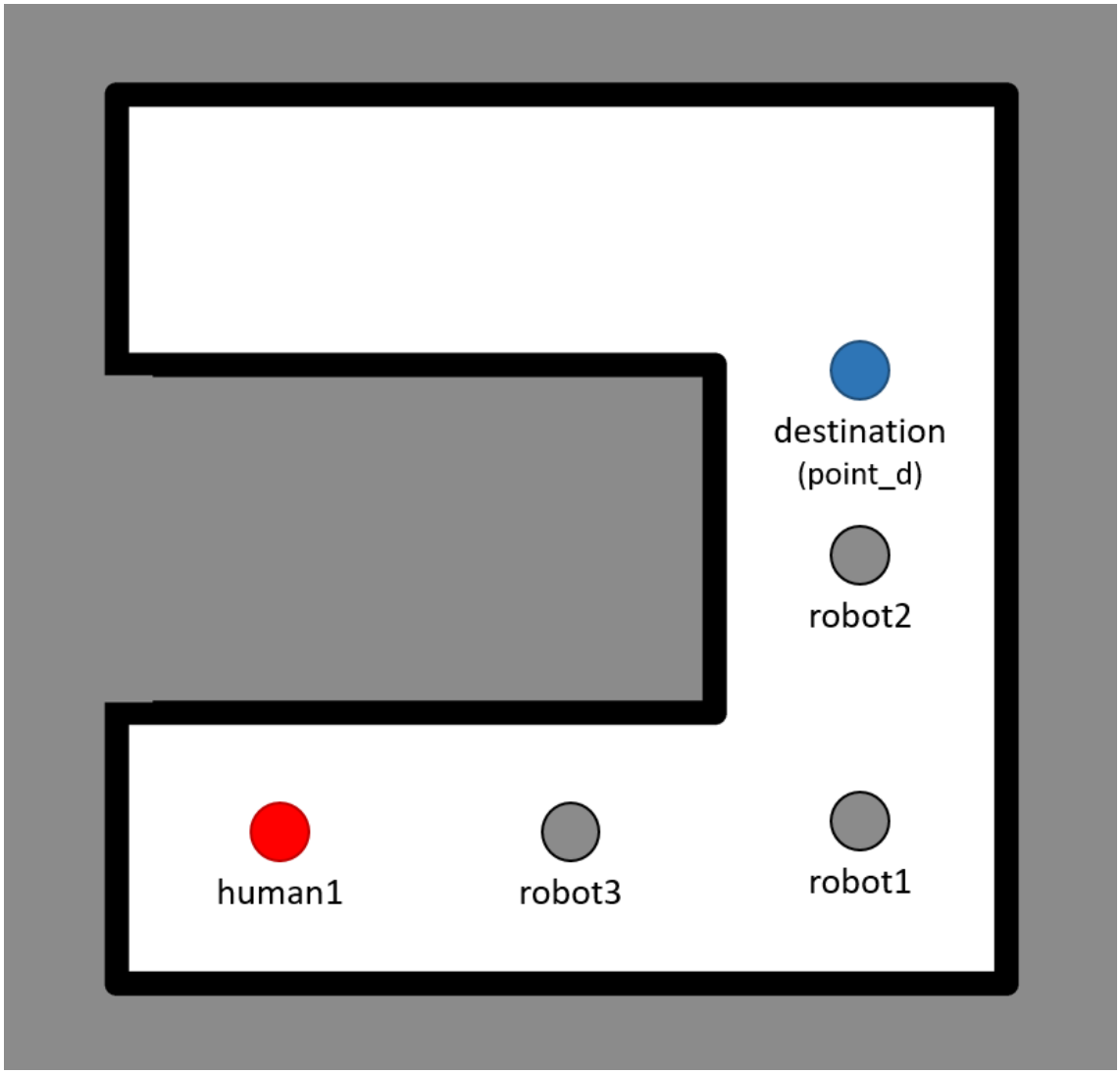

図 3.4.6: 実験条件 2 の概略図

結果として，どちらも問題なくサービスを提供できることが確認された．また，複数台のロボットを用いた方がより早く道案内を行えることが確認された．

3.4.2 実験条件 2:ユーザの位置を変える

次に，ユーザの位置を変えて検証を行った．実験条件 2 のユーザの位置，ロボットの配置および屋内のレイアウトの概略図を図 3.4.6 に示す．

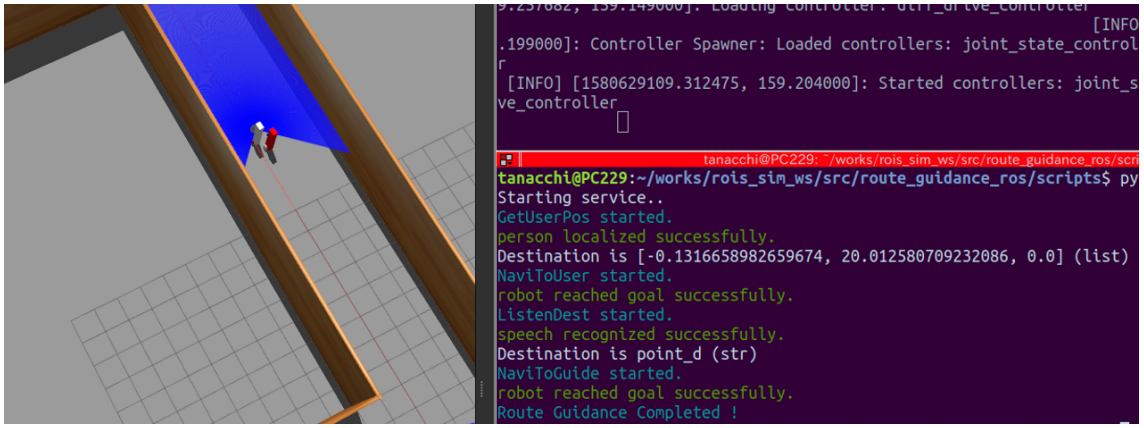

図 3.4.7: 実験条件 2 での実験の様子 (単体ロボット)

```
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$ py service_app_lv2.py
Starting service..
GetUserPos started.
person localized successfully.
Destination is [-0.1316658982659674, 20.012580709232086, 0.0] (list)
NaviToUser started.
robot reached goal successfully.
ListenDest started.
speech recognized successfully.
Destination is point_d (str)
NaviToGuide started.
robot reached goal successfully.
Route Guidance Completed !
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$
```

図 3.4.8: 実験条件 2 でのサービスの実行結果 (単体ロボット)

単一ロボットの場合　実験条件 2 における，単一ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.7，サービスアプリケーションの実行結果を図 3.4.8 に示す．どちらも，全てのタスクが問題なく遂行されたことを示している．

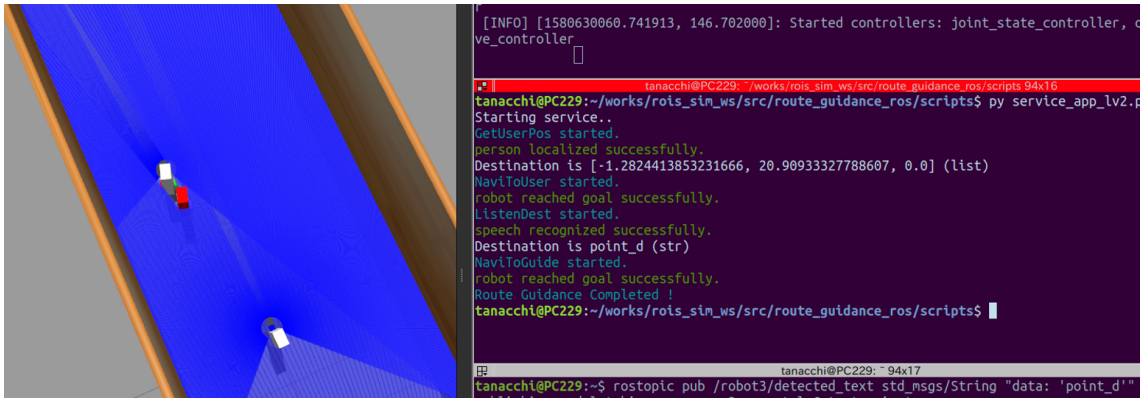

図 3.4.9: 実験条件 2 での実験の様子 (複数ロボット)

```
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$ py service_app_lv2.py
Starting service..
GetUserPos started.
person localized successfully.
Destination is [-1.2824413853231666, 20.90933327788607, 0.0] (list)
NaviToUser started.
robot reached goal successfully.
ListenDest started.
speech recognized successfully.
Destination is point_d (str)
NaviToGuide started.
robot reached goal successfully.
Route Guidance Completed !
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$
```

図 3.4.10: 実験条件 2 でのサービスの実行結果 (複数ロボット)

複数ロボットの場合　実験条件 2 における，複数ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.9，サービスアプリケーションの実行結果を図 3.4.10 に示す．どちらも，全てのタスクが問題なく遂行されたことを示している．

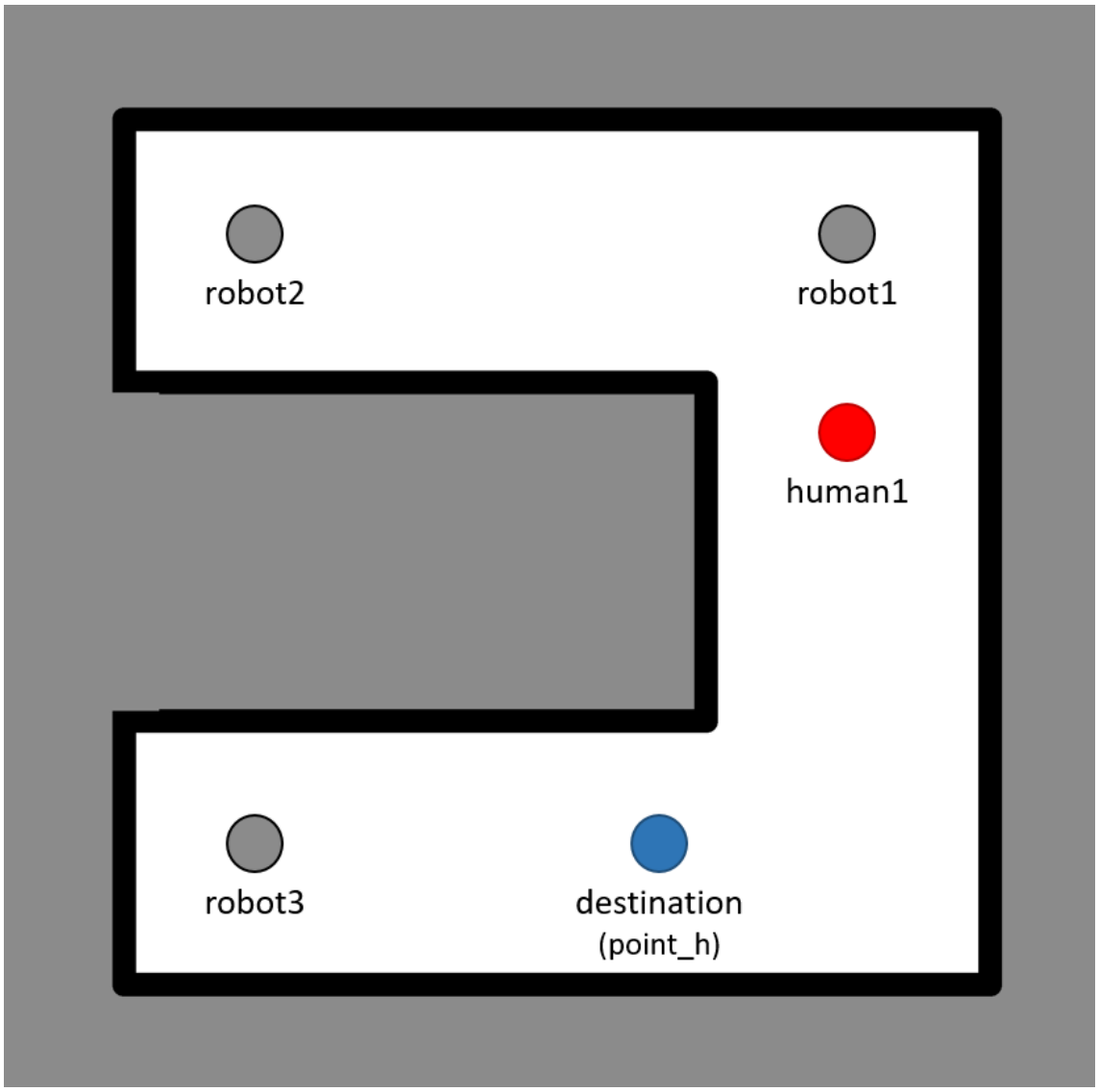

図 3.4.11: 実験条件 3 の概略図

結果として，どちらも問題なくサービスを遂行できることがわかった．

3.4.3 実験条件 3:ロボットの配置を変える

次に，ロボットの配置を変えて検証を行った．実験条件 3 のユーザの位置，ロボットの配置および屋内のレイアウトの概略図を図 3.4.11 に示す．

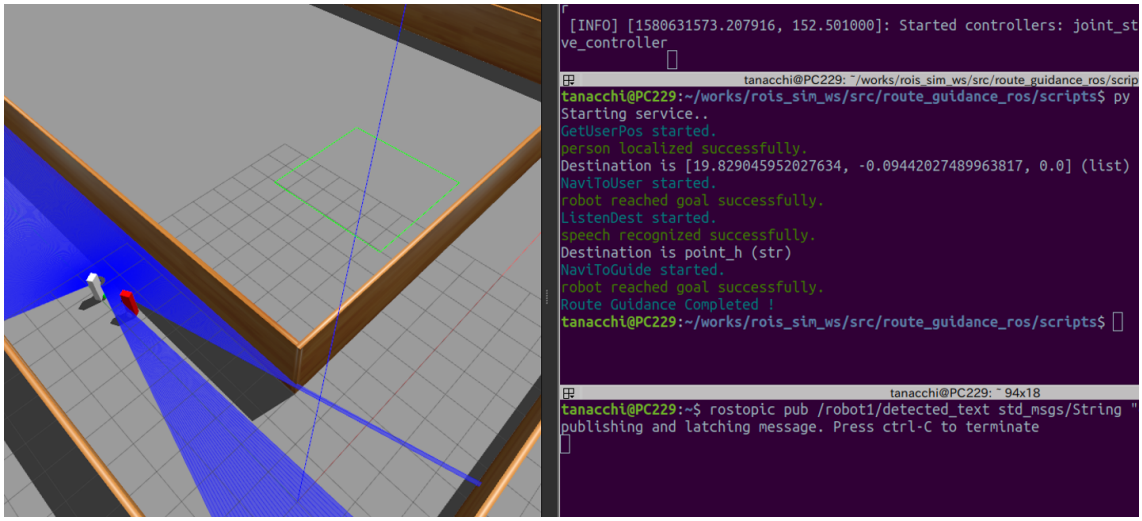

図 3.4.12: 実験条件 3 での実験の様子 (単体ロボット)

```
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$ py service_app_lv2.py
Starting service..
GetUserPos started.
person localized successfully.
Destination is [19.829045952027634, -0.09442027489963817, 0.0] (list)
NaviToUser started.
robot reached goal successfully.
ListenDest started.
speech recognized successfully.
Destination is point_h (str)
NaviToGuide started.
robot reached goal successfully.
Route Guidance Completed !
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$
```

図 3.4.13: 実験条件 3 でのサービスの実行結果 (単体ロボット)

単一ロボットの場合　実験条件 3 における，単一ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.12，サービスアプリケーションの実行結果を図 3.4.13 に示す．どちらも，全てのタスクが問題なく遂行されたことを示している．

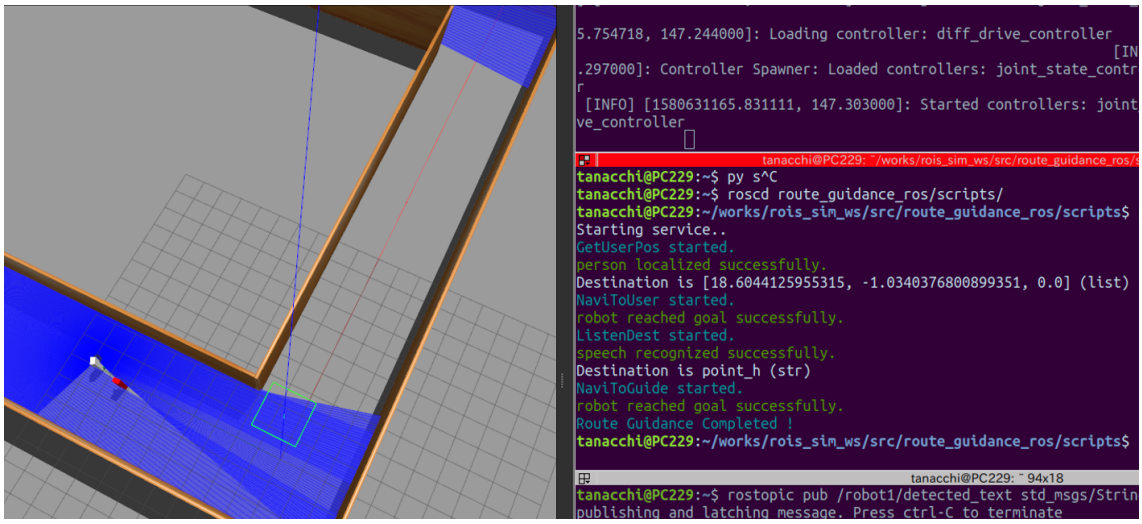

図 3.4.14: 実験条件 3 での実験の様子 (複数ロボット)

```
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$ py service_app_lv2.py
Starting service..
GetUserPos started.
person localized successfully.
Destination is [18.6044125955315, -1.0340376800899351, 0.0] (list)
NaviToUser started.
robot reached goal successfully.
ListenDest started.
speech recognized successfully.
Destination is point_h (str)
NaviToGuide started.
robot reached goal successfully.
Route Guidance Completed !
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$
```

図 3.4.15: 実験条件 3 でのサービスの実行結果 (複数ロボット)

複数ロボットの場合　実験条件 3 における，複数ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.14，サービスアプリケーションの実行結果を図 3.4.15 に示す．どちらも，全てのタスクが問題なく遂行されたことを示している．

結果として，どちらも問題なくサービスを遂行できることがわかった．今回は複数台の場合と単体の場合で同じ位置にいたロボットが対応したため，サービスを完了するのにかかった時間はほとんど変わらなかった．

3.4.4 実験条件 4:屋内のレイアウトを変える

次に，屋内のレイアウトを，実際の屋内を参考にしたモデルに変更し検証を行った．実験条件 4 のユーザの位置，ロボットの配置および屋内のレイアウトの概略図を図 3.4.16 に示す．

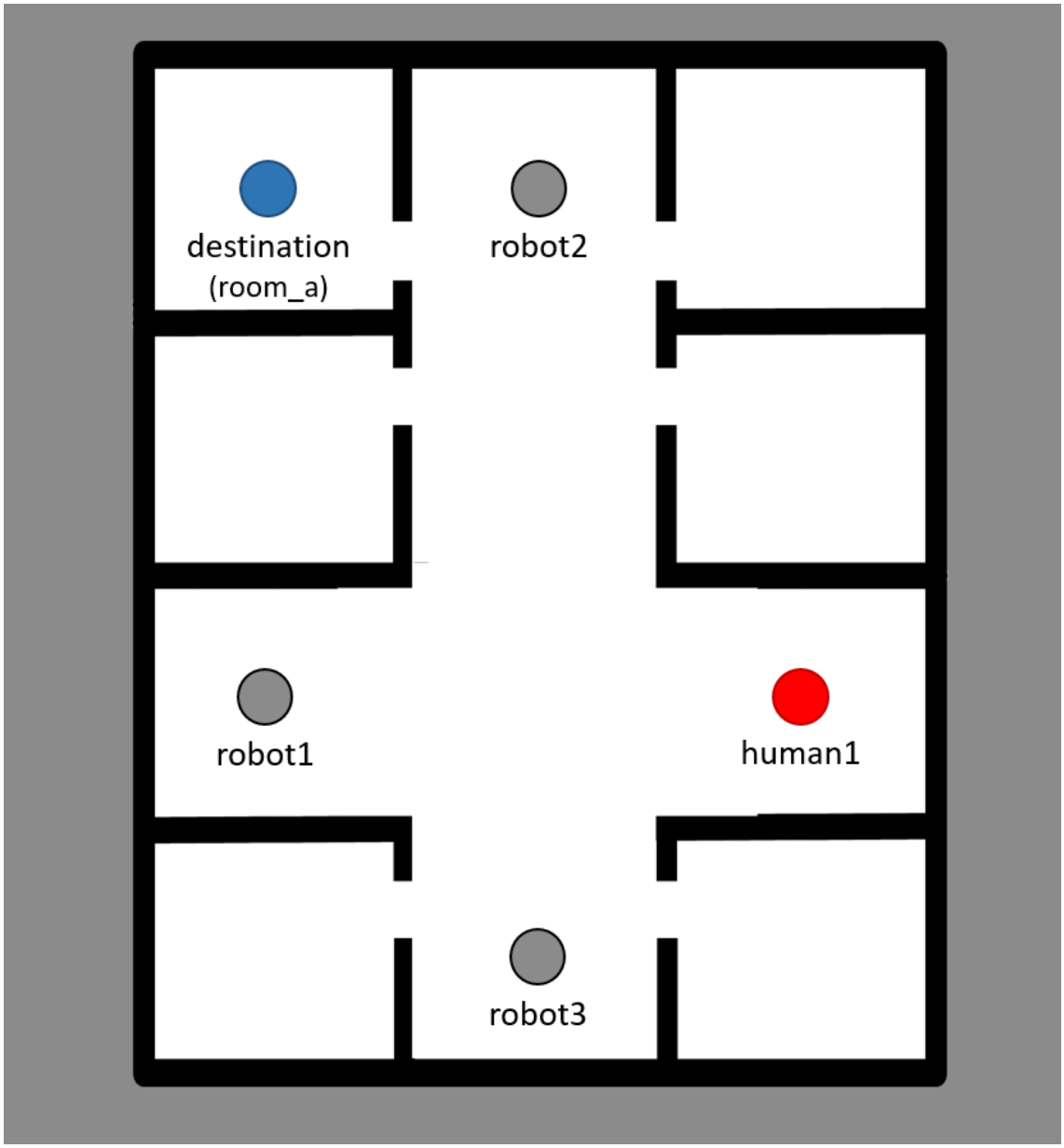

図 3.4.16: 実験条件 4 の概略図

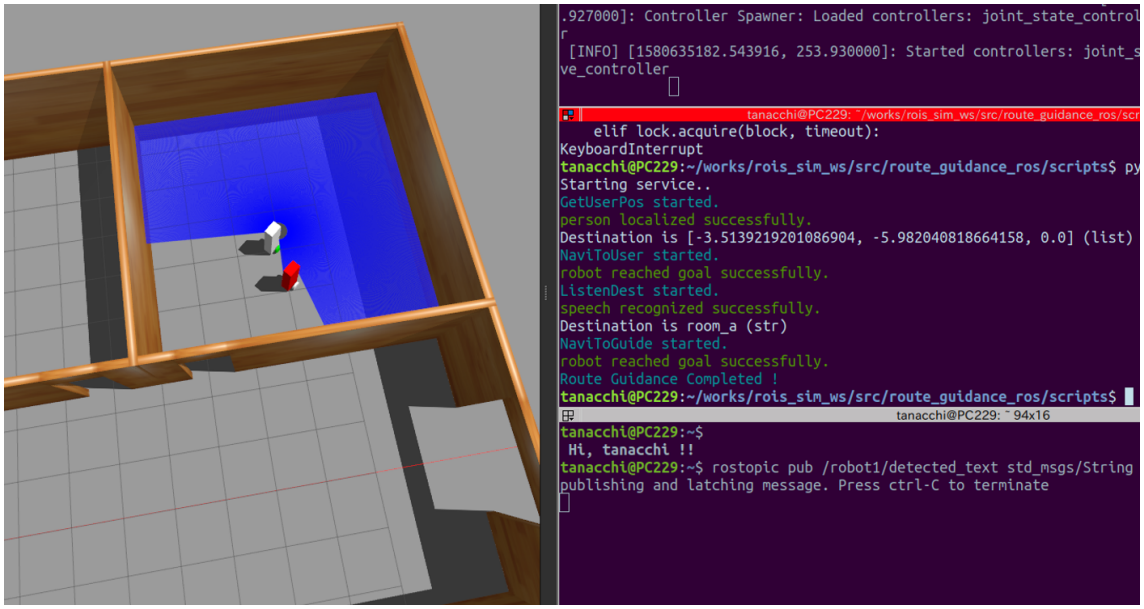

図 3.4.17: 実験条件 4 での実験の様子 (単体ロボット)

```
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$ py service_app_lv2.py
Starting service..
GetUserPos started.
person localized successfully.
Destination is [-3.5139219201086904, -5.982040818664158, 0.0] (list)
NaviToUser started.
robot reached goal successfully.
ListenDest started.
speech recognized successfully.
Destination is room_a (str)
NaviToGuide started.
robot reached goal successfully.
Route Guidance Completed !
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$
```

図 3.4.18: 実験条件 4 でのサービスの実行結果 (単体ロボット)

単一ロボットの場合　実験条件 4 における，単一ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.17，サービスアプリケーションの実行結果を図 3.4.18 に示す．どちらも，全てのタスクが問題なく遂行されたことを示している．

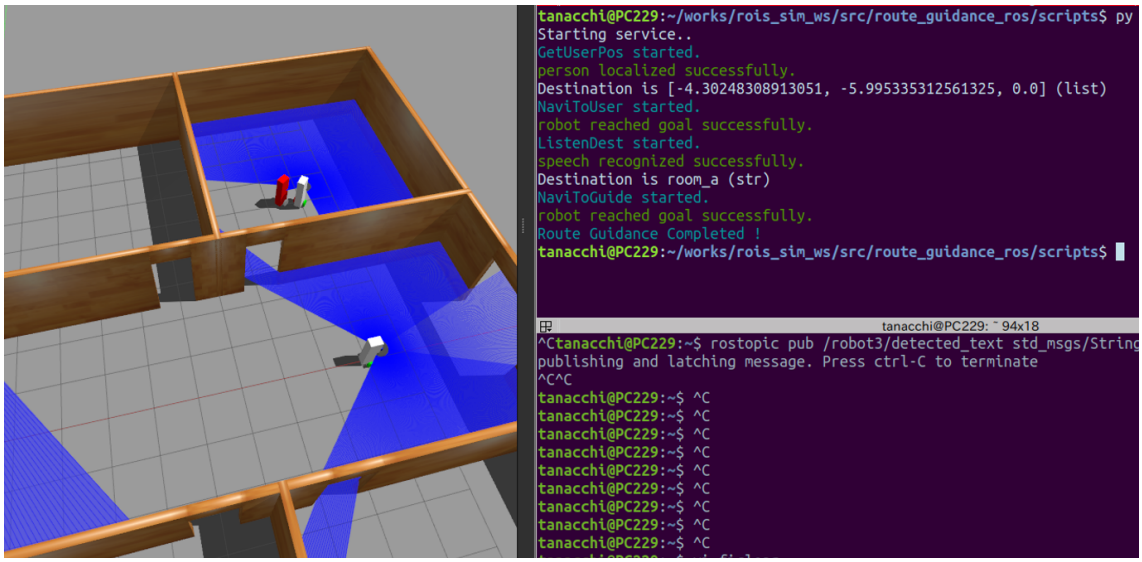

図 3.4.19: 実験条件 4 での実験の様子 (複数ロボット)

```
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$ py service_app_lv2.py
Starting service..
GetUserPos started.
person localized successfully.
Destination is [-4.30248308913051, -5.995335312561325, 0.0] (list)
NaviToUser started.
robot reached goal successfully.
ListenDest started.
speech recognized successfully.
Destination is room_a (str)
NaviToGuide started.
robot reached goal successfully.
Route Guidance Completed !
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$
```

図 3.4.20: 実験条件 4 でのサービスの実行結果 (複数ロボット)

複数ロボットの場合　実験条件 4 における，複数ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.19，サービスアプリケーションの実行結果を図 3.4.20 に示す．どちらも，全てのタスクが問題なく遂行されたことを示している．

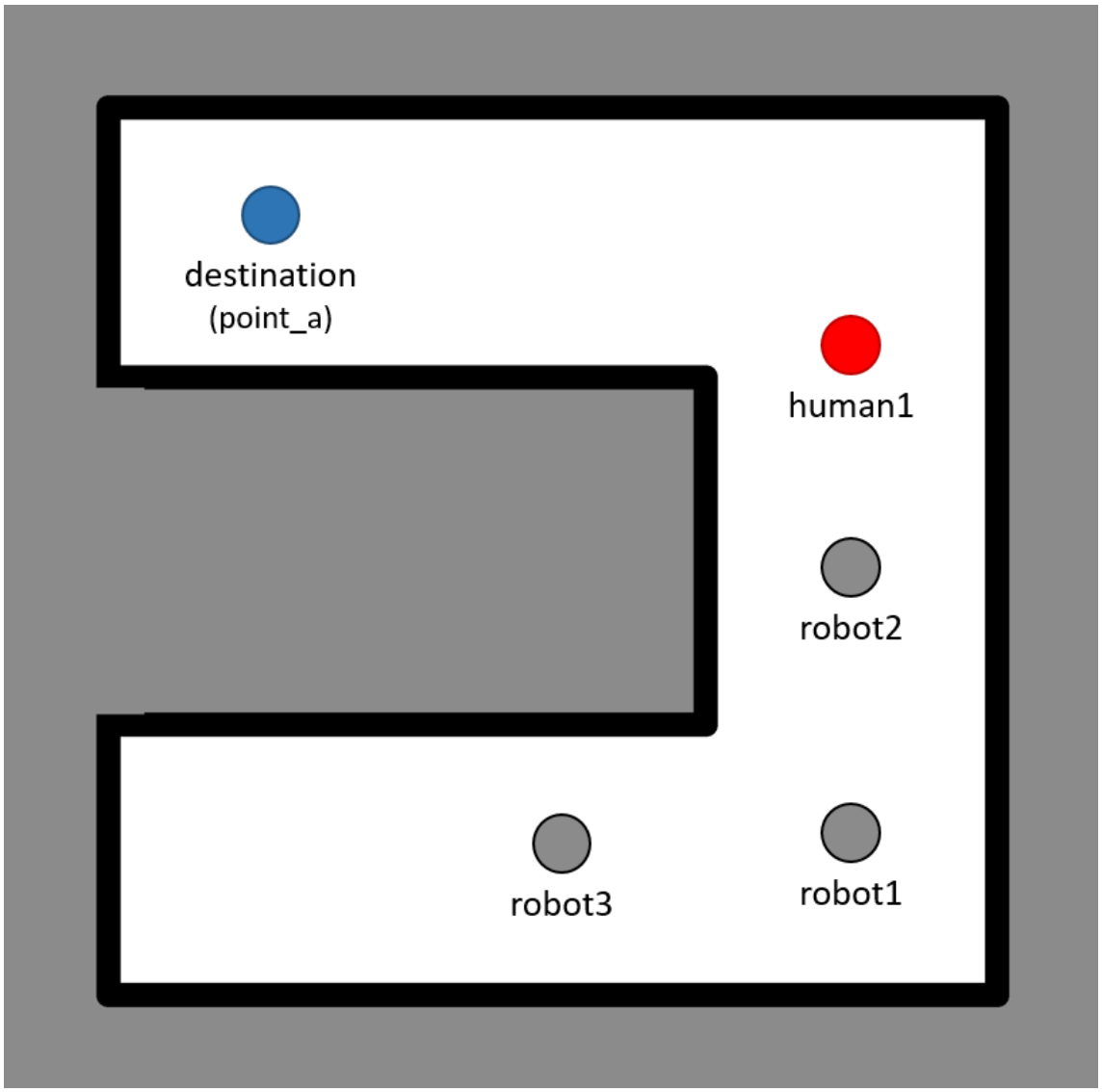

図 3.4.21: 実験条件 5 の概略図

結果として，どちらも問題なくサービスを遂行できることがわかった．

3.4.5 実験条件 5:自律移動中に障害を発生させる

次に，実験におけるロボットおよびユーザの位置と屋内のレイアウトを 3.4.1 と同じ状態にした．その上で，ユーザのもとへ自律移動をしている最中にロボットを倒し，タスク続行不可能にさせて対応できるかを検証した．実験条件 5 のユーザの位置，ロボットの配置および屋内のレイアウトの概略図を図 3.4.21 に示す．

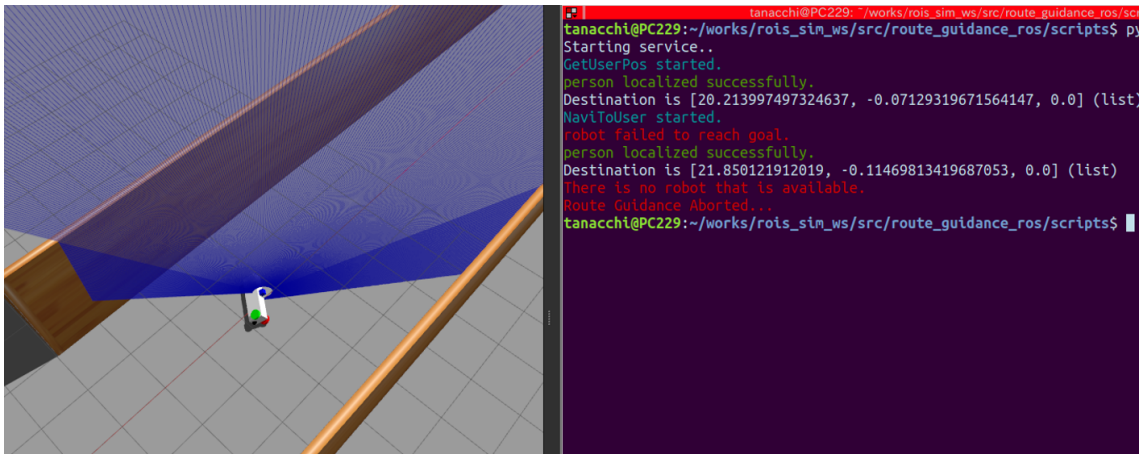

図 3.4.22: 実験条件 5 での実験の様子 (単体ロボット)

```
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$ py service_app_lv2.py
Starting service..
GetUserPos started.
person localized successfully.
Destination is [20.213997497324637, -0.07129319671564147, 0.0] (list)
NaviToUser started.
robot failed to reach goal.
person localized successfully.
Destination is [21.850121912019, -0.11469813419687053, 0.0] (list)
There is no robot that is available.
Route Guidance Aborted...
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$
```

図 3.4.23: 実験条件 5 でのサービスの実行結果 (単体ロボット)

単一ロボットの場合　実験条件 5 における，単一ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.22，サービスアプリケーションの実行結果を図 3.4.23 に示す．単一ロボットによるロボットシステムで自律移動中にタスク続行困難な状態になった際，サービスアプリケーションが再び自律移動の要求を出したところで「There is no robot that is available.」という文字列を出力した後，サービスを異常終了した．これは，ロボット 1 台では他のロボットにタスクを委託するなどの対処ができないため，サービスそのものが進行不可であると判断されたからである．

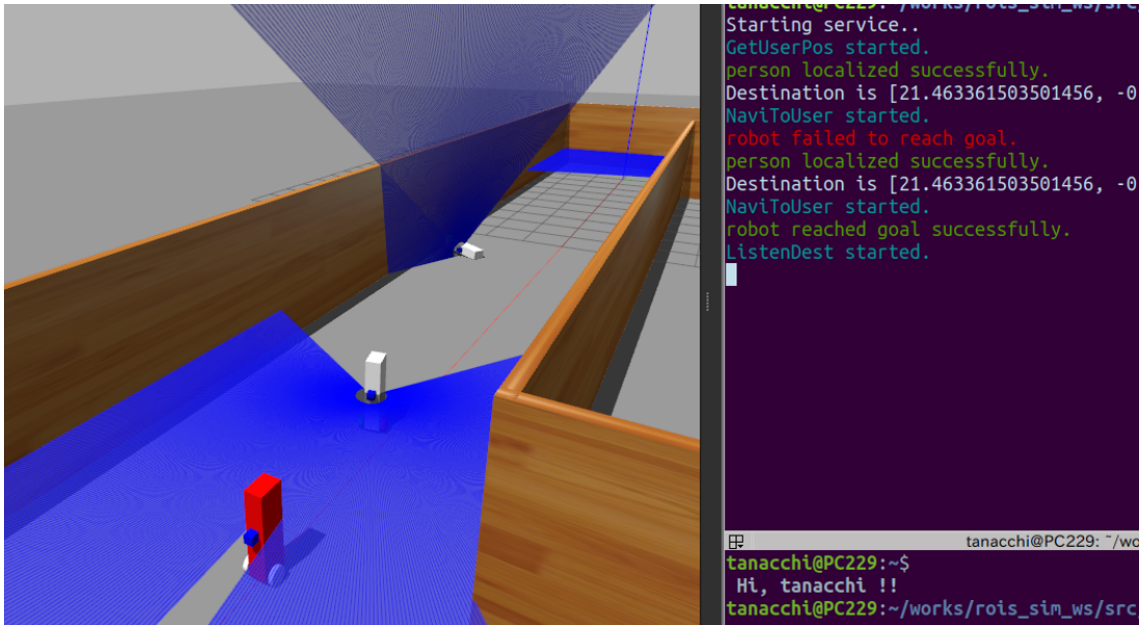

図 3.4.24: 実験条件 5 での実験の様子 (複数ロボット)

複数ロボットの場合　実験条件 5 における，複数ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.24，サービスアプリケーションの実行結果を図 3.4.25 に示す．複数ロボットによるロボットシステムで自律移動中にタスク続行困難な状態になった際，サービスアプリケーションが再び自律移動の要求を出したところで，別のロボットが自律移動を開始した．そこからはそのロボットがその後のタスクを担当し，サービスを完了した．

図 3.4.25: 実験条件 5 でのサービスの実行結果 (複数ロボット)

結果として，単一ロボットの場合は，1 台がタスクを続行できなくなった段階でサービスを遂行できなくなったため，サービスがそれを検知して続行不可能であると出力した．複数台の場合は，1 台がタスクを続行できなくなった場合，動作中でない他のロボットにタスクを委託することでサービスを続行した．このように，複数台のロボットの用いることでロバストなロボットシステムを実現できることも確認できた．

3.4.6 実験条件 6:無効な目的地の入力

次に，3.4.1 と同じロボット・ユーザの位置と屋内のレイアウトで，目的地の取得の際に無効な目的地を指定し，それに対処した上でサービスを遂行できるかの検証を行った．実験条件 6 のユーザの位置，ロボットの配置および屋内のレイアウトの概略図を図 3.4.26 に示す．

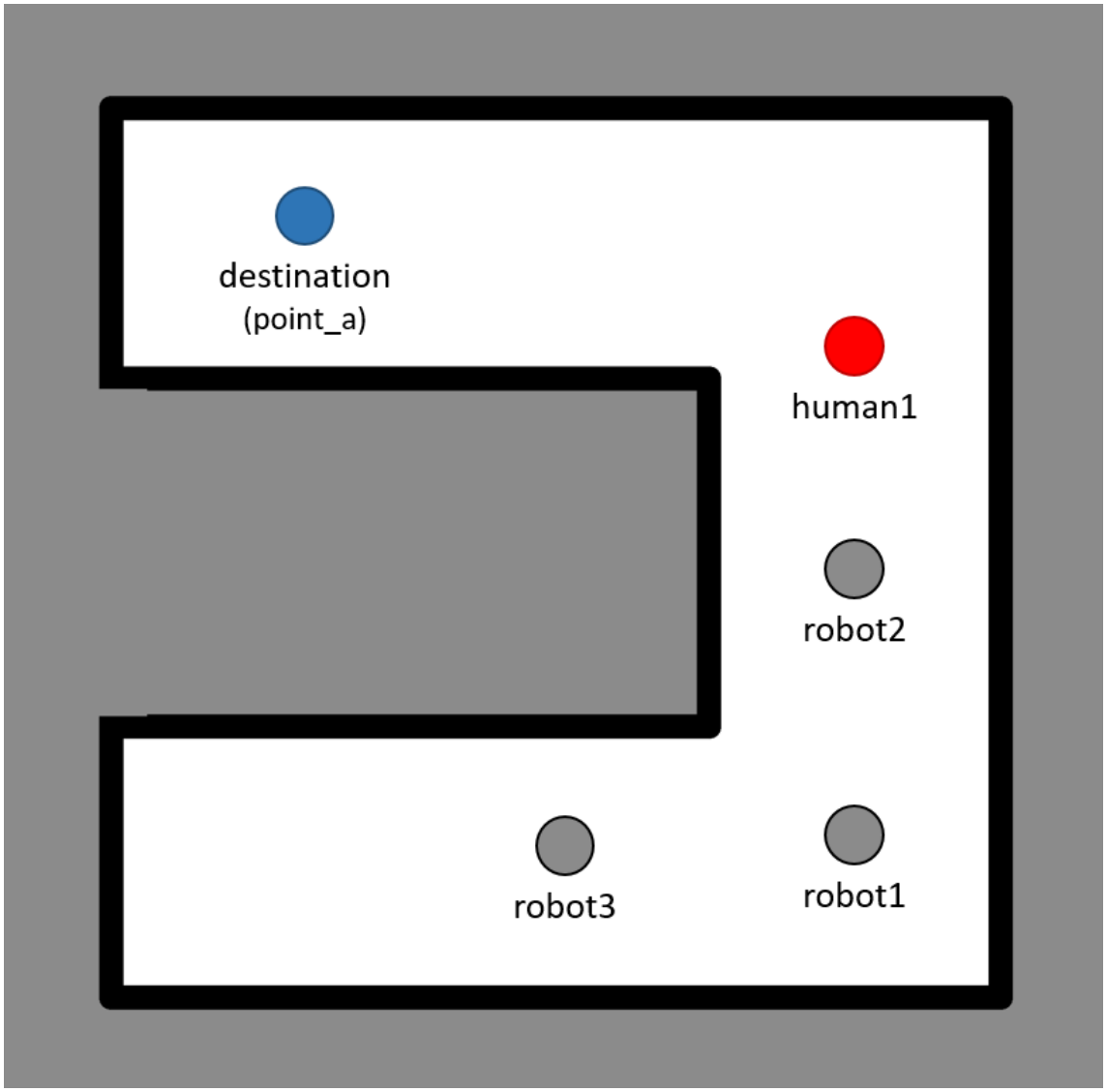

図 3.4.26: 実験条件 6 の概略図

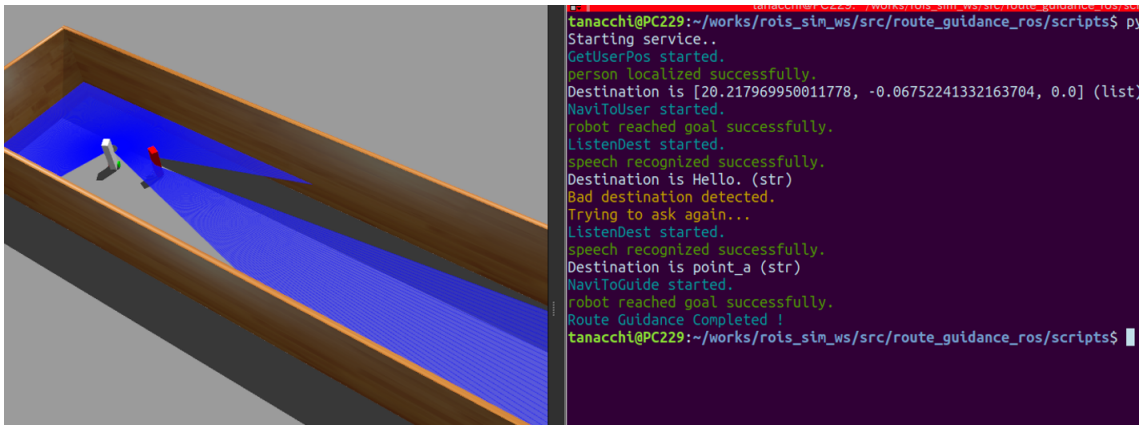

図 3.4.27: 実験条件 6 での実験の様子 (単体ロボット)

```
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$ py service_app_lv2.py
Starting service..
GetUserPos started.
person localized successfully.
Destination is [20.217969950011778, -0.06752241332163704, 0.0] (list)
NaviToUser started.
robot reached goal successfully.
ListenDest started.
speech recognized successfully.
Destination is Hello. (str)
Bad destination detected.
Trying to ask again...
ListenDest started.
speech recognized successfully.
Destination is point_a (str)
NaviToGuide started.
robot reached goal successfully.
Route Guidance Completed !
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$
```

図 3.4.28: 実験条件 6 でのサービスの実行結果 (単体ロボット)

単一ロボットの場合　実験条件 6 における，単一ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.27，サービスアプリケーションの実行結果を図 3.4.28 に示す．目的地の指定の際，まず「Hello.」という文字列を入力した．サービスアプリケーションがそれを目的地としてエンジンに自律移動の要求した後，無効な目的地がサービスアプリケーションに通知され，それを受けてサービスアプリケーションが再び目的地の取得を要求したことがわかる．次に正当な目的地を指定すると，正常にサービスを続行でき，最終的にサービスを完了できた．

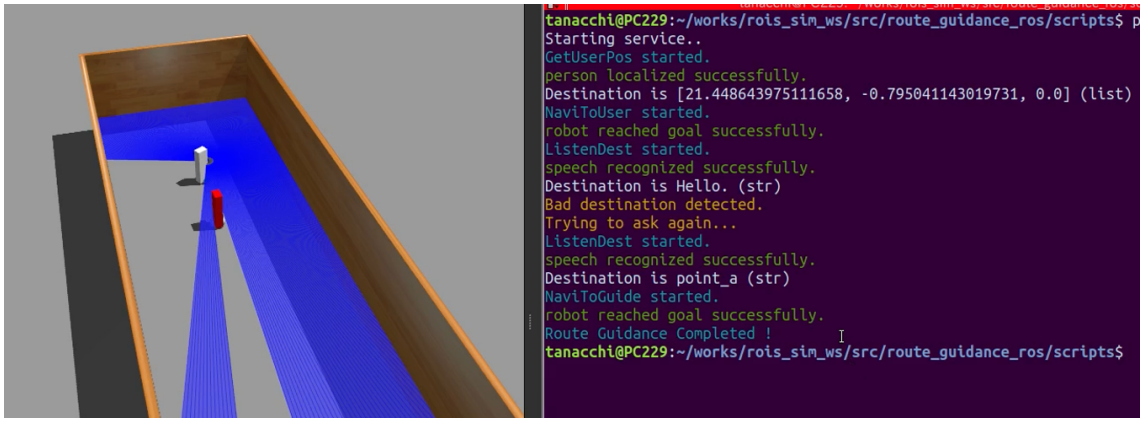

図 3.4.29: 実験条件 6 での実験の様子 (複数ロボット)

```
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$ py service_app_lv2.py
Starting service..
GetUserPos started.
person localized successfully.
Destination is [21.448643975111658, -0.795041143019731, 0.0] (list)
NaviToUser started.
robot reached goal successfully.
ListenDest started.
speech recognized successfully.
Destination is Hello. (str)
Bad destination detected.
Trying to ask again...
ListenDest started.
speech recognized successfully.
Destination is point_a (str)
NaviToGuide started.
robot reached goal successfully.
Route Guidance Completed !
tanacchi@PC229:~/works/rois_sim_ws/src/route_guidance_ros/scripts$
```

図 3.4.30: 実験条件 6 でのサービスの実行結果 (複数ロボット)

複数ロボットの場合　実験条件 6 における，複数ロボットシステムでのサービスシステムの動作終了時の様子を図 3.4.29，サービスアプリケーションの実行結果を図 3.4.30 に示す．単一ロボットの場合と同様に，ます目的地に「Hello.」という文字列を指定したが，無効な目的地としてサービスアプリケーションに通知され，それを受けてサービスアプリケーションが再び目的地の取得を要求したことがわかる．その後も同様に，正当な目的地を指定すると，正常にサービスを続行でき，最終的にサービスを完了できた．

結果として，どちらも無効な入力に対応した上でサービスを遂行できることがわかった．

全ての実験条件における結果を表 3.4.3 に示す．また，ユーザのもとへ向かう自律移動タスクの所要時間の比較表を表 3.4.4 に，実験条件 5 を除いた比較のためのグラフを図 3.4.31 に示す．

	全てのタスク遂行	
実験条件	単一ロボット	複数ロボット
1	○	○
2	○	○
3	○	○
4	○	○
5	×	○
6	○	○

表 3.4.3: 道案内サービスの動作結果

	所要時間 [s]			
実験条件	単一ロボット	複数ロボット	減少率	備考
1	25	17	32%	
2	30	24	17%	
3	17	20	-18%	同じロボットが動作
4	22	18	18%	
5	∞	112	-	1 台の場合タスクが異常終了 3 台の場合ロボットがスタックして エラーの検知を経て 別のロボットが動作したため値が大きい
6	25	20	20%	

表 3.4.4: ユーザのもとへ向かう自律移動の所要時間

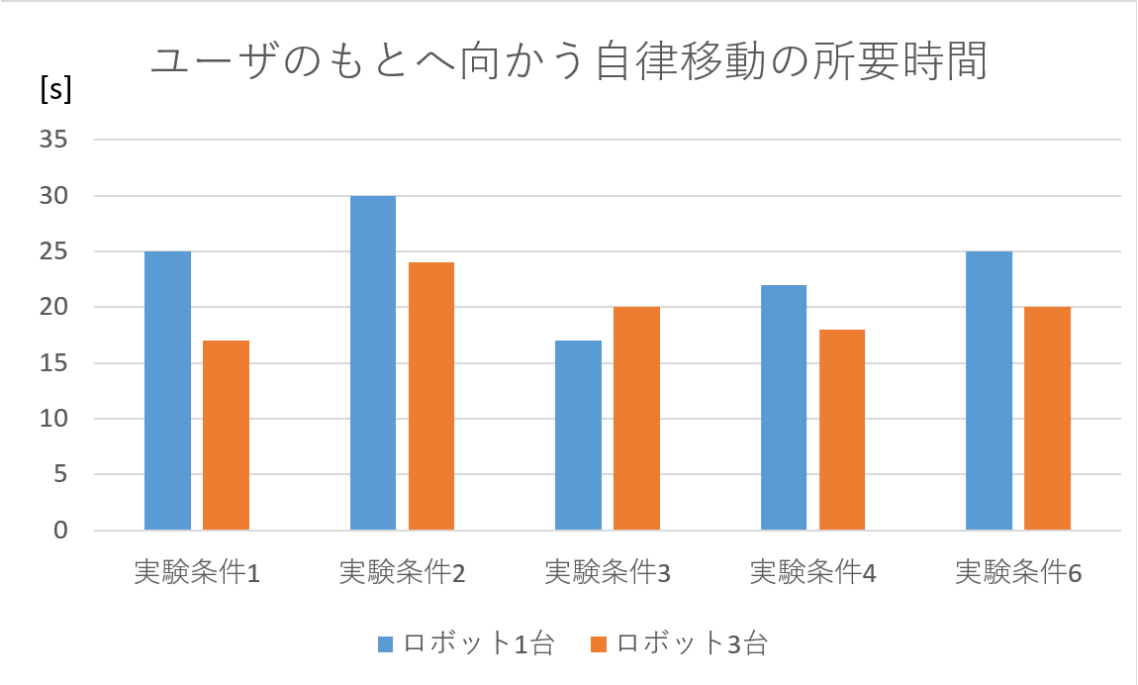

図 3.4.31: ユーザのもとへ向かう自律移動の所要時間のグラフ

3.4.7 評価

以上で行った 6 パターンの条件下において，サービスの実装を変更することなく台数の異なるロボットシステムの変化に対応できることがわかった．単一ロボットの場合の表 3.4.4 や図 3.4.31 は，3 台のロボットを用いたロボットサービスでは，開始から正常終了までの時間が単一ロボットの場合と比べて短くなる傾向があることを示しており，ロボット間の連携によってロボットサービスの効率化されることが確認された．また，表 3.4.3 の実験条件 5 (自律移動時の障害発生) の項目から，3 台のロボットを用いたロボットサービスだけが障害発生に対応してタスクを続行できたことから，ロボット間の連携によるロバスト性の向上が確認された．これは 2 種類のロボットシステムにおける「連携の有無」という差異を示している．

すなわち，RoIS Framework を用いることで，機能のレベルで吸収できない,「ロボット間の連携の有無」という違いのあるロボットシステム間でサービスアプリケーションの互換性を実現できることが示された．

3.5 まとめ

本章では，はじめに RoIS Framework を利用したロボットサービスの具体例として構築した道案内サービスの概要とシミュレーション環境を構築する方法について述べた．次に，それを異なる実験条件で動作させ，タスクを遂行できるかを検証・評価した．結果として，異なるロボットシステムで同一のサービスアプリケーションを動作させてタスクを遂行でき，RoIS Framework が目標とする互換性を実現できることが確認できた．

第4章　考察

本研究では RoIS Framework を利用したロボットサービスの具体例として，道案内サービスを台数の異なるロボットシステムで遂行する環境をシミュレーションを用いて構築した．そこから，RoIS Framework が目標とする，異なるロボットシステム間におけるサービスアプリケーションの互換性が実現できたかについて検証した．実験の結果，同一のロボットサービスアプリケーションを異なる 2 種類のロボットシステムで動作させられることが確認できた．また，ロボットの配置やユーザの位置，シミュレーション上の環境が変化した場合やエラーハンドリングや対話的要素が加わった場合などについても，同様にタスクを遂行できることが確認できた．本章では，本研究における実装および実験を通して得られた発見や今後の課題について述べる．

4.1　互換性の観点から

本研究で実装した道案内サービスにおける 2 種類のロボットシステムは，台数とそれによる連携の有無の違いがある．そのため同一のサービスアプリケーションで動作させるためには，HRI コンポーネントより高いレベルでインタフェースの差異を吸収する必要がある．しかし，RoIS Framework の仕様書では，HRI コンポーネント， HRI エンジンおよびサービスアプリケーションのインタフェースを規格しているが，その内部の実装手法については言及されていない．また，単に機能レベルの実装の違いであれば，規格に従って機能のインタフェースを設計すれば互換性を実現できるが，ロボットの連携の有無の違いといった，ロボットシステムの全体に波及するインタフェースの差異を如何にして吸収するかが確立されていない．

本研究で実装した道案内サービスを例に取って議論する．例えば，ユーザのもとへ向かう自律移動では，タスクの効率化のためユーザに最も近いロボットがタスクを担当するように実装した．しかし，目的地を取得した後に案内するための自律移動では，接客を担当したロ

ボットが担当する必要がある．このとき，先の自律移動と同じ動作をさせると，接客中のロボットより他のロボットの方が目的地に近かった場合，関係ないロボットが動作してしまうためサービスが成り立たなくなる．サービスを成り立たせるためには，ユーザの目的地に向かう自律移動を行う際に,「ユーザのもとへ向かったロボット」を指定してタスクを要求する必要がある．同様の理由でロボットの指定を行う必要があるのが，音声認識による目的地取得タスクである．ユーザに目的地を尋ねる際も，必要なのはユーザのもとへ向かったロボットからの音声認識結果であって，関係のないロボットが認識した音声はタスクの進行に関わってはならない．

また前提として，サービスアプリケーションの互換性の実現のため，サービスアプリケーション側は，ロボットが単体であるか複数あるかなどの変化に関係なく，同じ手順とインタフェースでタスクの進行を制御する必要がある．逆に，HRI コンポーネントは 1 つのロボットの機能として設計されているため，HRI コンポーネントのインタフェースに担当するロボットや動作モードを指定したりする類のパラメータは存在しない．この点が RoIS Framework を用いて複数台のロボットによる連携を実現する上での未定義な部分である．

結果として本研究では，具体的なロボットサービスの実装を通して，複数台のロボットによる連携を含みながらサービスアプリケーションの互換性を保つことのできるロボットシステムを構築できた．以下に，それを実現するための 2 通りの手法を提案する．

実装手法 1：HRI エンジンによるサブタスク割り当て　1 つ目は HRI エンジンの内部に，ロボットシステムのタスクの状態の一部を保持し，それによって動作を変更する手法である．これは，音声認識による目的地取得タスクを実行する際に，どのロボットからの音声認識結果を使用するのかを指定する場合に使用した．具体的には，ユーザのもとへ向かう自律移動のタスクが成功したことが HRI エンジンに通知されたときに，そのロボットの識別子を「次に目的地取得タスクを担当するロボット」として HRI エンジン内で保持するように実装した．また，自律移動タスクの「ユーザのもとへ向かうための自律移動」と「ユーザを案内するための自律移動」とで動作を変化させるために，コンポーネント内部に「どちらのモードで自律移動するのか」という情報を保持させ，自律移動が正常終了する度にモードを切り替えるようにして対処した．ここで，この手法はロボットシステム以下の階層に，タスクの進行に関係する情報を持たせているため，2.6 で述べた開発ポリシーに反しているよ

うに思えるかもしれない．しかし，サービスアプリケーション，HRI エンジン，HRI コンポーネントそれぞれが保持できる情報を，サービスアプリケーションはタスクの進行に必要な情報のみ，HRI エンジンはロボットシステムを統括するのに必要な情報のみ，HRI コンポーネントは機能レベルの動作を切り替えるのに必要な情報のみと制限することで，RoIS Framework を構成する要素の役割分担の明瞭さが保たれる．現に，音声認識をするロボットの指定をするための情報は HRI エンジンに，自律移動コンポーネントの動作モードを指定するための情報はコンポーネント内に保持するように設計している．

実装手法 2:仮想コンポーネントの利用　現状，RoIS Framework を利用して具体的なロボットサービスを実装した例が公開・普及されていないため，1 つ目に述べた実装手法が最適なものとは判断し難い．そこで，2 つ目の候補として仮想コンポーネントを利用した手法を提案する．これは，複数台分の同じ機能を 1 つのコンポーネントとしてまとめ，1 台のロボットの機能を扱うのと同じ要領でタスクの要求を行うものである．本研究では，複数台のロボットの自律移動タスクを管轄する機能を仮想コンポーネントとして実装した．複数台分の自律移動タスクを制御する Virtual Navigation コンポーネント (仮想 Navigation コンポーネント) は，単一ロボット用の Navigation コンポーネントと同じインタフェースを持つが，「ユーザのもとへ向かうための自律移動」の際に，ロボットそれぞれの現在地から目的地に最も近いロボットを探索し，そのロボットに対して自律移動をさせるという処理を行う．また，あるロボットが自律移動タスクを開始した際に，スタックしているロボットが存在すれば帰還させるという機能も持っている．これらの機能は，Navigation コンポーネントのインタフェースだけでは定義できない動作ではあるが，これらを Virtual Navigation コンポーネントの内部で行う「機能の実装の差異」と捉えれば，規格に沿った正当なコンポーネントの 1 つであると言える．また，仮想コンポーネントを利用する利点として，ロボットの割り当てや帰還の指示などのコンポーネント固有のロジックをコンポーネント内部に押し込められるという点がある．RoIS Framework の仕様書では，(サブ) HRI エンジンと HRI コンポーネントを動的に切り替えられることが要求として存在するため，その仕様を満たすには，HRI エンジンのモジュラリティを保つことのできるこの手法が有効であると言える．

4.2 サービスアプリケーション実装の観点から

本研究では，道案内サービスのアプリケーションの実装を 2 段階に分け，プロトタイプと改良版に分けて開発した．プロトタイプではエラーハンドリングや対話的動作が一切含まれていないため，タスクは固定の順番で進行する．そのため，サービスアプリケーション内部では，タスク終了の通知を受け取ったら単に次のタスクの要求を出すようにすればよい．しかし，サービスアプリケーションの中でエラーハンドリングや対話的動作が含まれるようになると，タスクの実行結果によって次のタスクが変わってくる．例えば，本研究で実装したサービスアプリケーションでは，ユーザに目的地を尋ね，結果を取得した際に，指定された目的地が存在すればそこまでユーザを連れて行くタスクに移るが，存在しなければ再びユーザに目的地を尋ねるタスクに移る．このような対応が増えると，タスクの種類と分岐の個数にしたがってサービスアプリケーションの実装が複雑になってしまうという問題がある．そこで，タスクの進行を図 3.3.15 に示したような状態遷移図で表現し，それをもとにサービスアプリケーション内部にステートマシンを構築することでタスクの進行を簡潔に実装できた．

4.3 今後の課題

今回実装したロボットサービスの欠点としては，複数台のロボットによる連携の要素が薄く，単体のロボットによるロボットシステムと複数のロボットによるロボットシステムの違いがそこまで顕著でないという点が挙げられる．サービスを提供するのに掛かる時間や障害に対するロバスト性が向上したことは，連携の要素が起因することであるが，スペースをカバーリングする機能あるいはエリアごとに分担してユーザを案内するなどの機能などロボットが複数利用できることを活かす方法はまだ考えられる．さらに，本研究で実装したロボットサービスは複数のユーザによる同時の案内要請に対応できない．これは自律移動タスクのモード切替が，それを実行した回数のみで行うといったナイーブな実装になっているからである．このようなサービスを効率化，ロバスト化するための，連携の要素を強めたロボットシステムでも互換性を保つ手法の確立が必要である．

また，RoIS Framework の仕様では，エラーの伝達方法が規格化されているが，ロボットシステム固有の，あるいはユーザー定義のエラーの詳細を伝達するための共通のフォーマッ

トが存在しない．これは，これはエラー処理における互換性を損ねる要因であり，また RoIS Framework を実用化する際に開発者への負担となり得る問題点であるため，RFC (Request For Comments) を利用してエラー情報のフォーマットを標準化するなどの対策を講じる必要がある．

4.4 まとめ

本章では，第 3 章に述べた実装および実験を通して RoIS Framework を利用したロボットサービスを構築する手法と機能レベルを超えた差異の存在するロボットシステム間の互換性を保つための手法について述べ，今後の課題として本研究での実装だけでは互換性を保つことができない状況と RoIS Framework を実用化する上での仕様の問題点について考察した．

第5章　結論

本研究では，RoIS Framework を利用したロボットサービスの具体例として，道案内サービスアプリケーションと台数の異なる 2 種類のロボットシステムを構築し，それらをシミュレーションを用いて実際に動作させ，RoIS Framework を利用したロボットサービスのイグザンプルを示すとともにこれを利用することでサービスアプリケーションの互換性を実現できることを示した．また，台数とロボット間の連携の有無の異なるロボットシステムが同一サービスアプリケーションで動作可能であることを確認することで，RoIS Framework を利用することでコンポーネントレベルよりも高いレベルの実装の差異を吸収できることを示した．

以下に本論文の要約と結論を述べる．

第 1 章「序論」では，始めに，様々な分野において特定のタスクを遂行するロボット技術が発展している中で，問題となっているロボットサービスのソフトウェアにおける課題について述べ，その課題を解決する RoIS Framework の概要について説明した．次に，ロボット開発におけるシミュレーションの長所について説明した．それらを踏まえ，本研究では，RoIS Framework において確立されていなかった，実際のロボットサービスを構築する手法と台数の変化を伴うロボットシステム間におけるサービスアプリケーションの互換性について実装と検証を行うことを述べた．

第 2 章「互換性実現のための方針」では，ロボットの運動レベルの概念について述べ，それを踏まえて，関連研究である ROS と SIGVerse それぞれについて比較した．具体的には，ROS とは標準化を目指す運動レベル，SIGVerse とはロボットサービスにおける焦点を当てる分野が異なることを述べ，RoIS Framework の新規性を示した．次に，RoIS Framework の構造について説明し，それぞれがどのような役割を果たすのか，そして RoIS Framework が如何にしてサービスアプリケーションの互換性を実現するのかを述べた．次に，上で述べた RoIS Framework の説明を踏まえ，RoIS Framework を利用した上で実際に商用利用で

きるロボットサービスに求められる仕様要求と，その要求を達成するための実装における開発ポリシーについて述べた．最後に，実際に仕様要求が満たされることを検証するためのシミュレーションの活用方法について述べた．

第 3 章「ロボットサービスのシミュレーションと互換性の評価」では，始めに具体例として構築したロボットサービスの流れと想定される障害，ロボットシステムの動作の違いについて述べた．次にロボットサービスを構成する要素と，開発にあたって利用した Gazebo と ROS について説明した．次に Gazebo，ROS，RoiS Framework を連携させてロボットサービスを構築した流れをモデルの作成，ROS の自律移動の確立，RoIS Framework の導入，サービスアプリケーションの実装の項目に分けて説明した．最後に，6 パターンの実験条件それぞれについて異なるロボットシステムを同一のサービスアプリケーションで動作させ，タスクを遂行できるかの検証を行い，結果として互換性が保てることを示した．

第 4 章「考察」では，ロボットサービスを実装した中で得られた，RoIS Framework を利用したロボットサービスを構築する手法とロボットシステムの実装の差異を吸収する手法について提案し，今後の課題について考察した．

本研究を通して，シミュレーションを用いて RoIS Framework を利用したロボットサービス実装のノウハウは得られたが，ロボットシステムにおける実装の差異を吸収するための手法の候補を提案したに過ぎない．また，本研究の複数台のロボットを用いたシステムは連携の要素こそあるが，ワークスペースのカバーリングや複数人同時による案内要請に対応しておらず，複数台のロボットを用いることの利点を活用していない．

今後の展望としては，スペースのカバーリングなどのロボットが複数あることをさらに活用した，より複雑な動作を伴うサービスや商用化可能なレベルまでエラー処理や対話機能を改良したサービス，あるいは道案内とは別のサービスを提供するロボットサービスを構築することで，ロボットサービスの発展および実用化に向けて同一のサービスアプケーションを複数台のロボットシステムにそのまま適用する手法を確立する研究開発が必要である．

謝辞

本研究は九州工業大学工学部総合システム工学科猪平研究室で行われました．

猪平栄一講師には本研究を進めるにあたり，多くのご指導，ご助言を賜り，多くのことを学ばせていただきました．また，私が同学の生命体工学研究科の推薦入試を受けた際にも，志望調書の書き方や面接の受け方など細やかなご指導を賜り，お陰様で早い段階で合格することができ，研究に専念することができました．この場を借りて，深く御礼申し上げます．

同研究室の修士 1 年の東亮太先輩には，研究で行き詰まった際にご助言をくださるなど，研究を進める上で多くのお力添えをいただきました．ありがとうございました．

最後に，本研究を行うにあたり，多くの方にお世話になりました．この場を借りて，心よりお礼申し上げます．

2020 年 2 月 13 日

田中大揮

参考文献

[1] 森田 武史 山口 高平 小野 宙生, 小池 開人. 教育現場における議論支援ロボット. **人工知能学会全国大会論文集**, JSAI2019:1O4J1201–1O4J1201, 2019.

[2] 塩見 昌裕 石黒 浩 萩田 紀博 宮下 善太, 神田 崇行. 顧客と顔見知りになるショッピングモール案内ロボット. **日本ロボット学会誌**, 26(7):821–832, 2008.

[3] 橋田 規子 海野 一樹. 企業博物館のための案内ロボットの研究. **日本デザイン学会研究発表大会概要集**, 66:534, 2019.

[4] 康朗 中尾. 図書館サービスにおける ai ロボットの活用. **鹿児島国際大学情報処理センター研究年報**, (24):1–10, 2019.

[5] 森口 智規 岡田 卓也 湊 雄一朗 中野 剛 田中 昌司 下本 英生 堀 俊夫 今井 倫太, 高橋 正樹. 病院内ロボット搬送システムの開発. **日本ロボット学会誌**, 27(10):1101–1104, 2009.

[6] ATR Intelligent Robotics and Communication Laboratories. Robotic interaction service framework (rois). https://irc.atr.jp/std/RoIS.html.

[7] 佐藤 幹 堀 俊夫. Robotic interaction service（rois） framework: サービスロボットシステム用インタフェースの標準化. **日本ロボット学会誌**, 29(4):341–344, 2011.

[8] Python Software Foundation. pyrois・pypi. https://pypi.org/project/pyRoIS/. 参照 Feb.12,2020.

[9] 猪平 栄一 齊藤 明王. Rois フレームワークに基づいたソフトウェアフレームワークの実装と通信性能の評価. **電子情報通信学会技術研究報告 = *IEICE technical report :* 信学技報**, 117(306):17–20, nov 2017.

[10] 國吉 康夫 浅田 稔. **ロボットインテリジェンス**, page 10. 岩波書店, 2005.

[11] Open Robotics. Docimentation - ros wiki. http://wiki.ros.org/.

[12] Open Robotics. navigation - ros wiki. http://wiki.ros.org/navigation.

[13] Tetsunari Inamura Yoshiaki Mizuchi. Cloud-based multimodal human-robot interaction simulator utilizing ros, unity frameworks. *2017 IEEE/SICE International Symposium on System Integration (SII)*, pages 948–955, 2017.

[14] Robotic interaction service framework (rois). https://www.omg.org/spec/RoIS/1.2/PDF, 2018.

[15] 銭 飛. *ROS* プログラミング. 森北出版株式会社, 2016.

[16] Open Robotics. cv_camera - ros wiki. http://wiki.ros.org/cv_camera.

[17] Open Robotics. Gazebo. http://gazebosim.org.

[18] ANA ホールディングス. アバターロボット | avatar-in アバターイン. https://avatarin.com/avatar.

[19] Open Robotics. Ros wiki. http://wiki.ros.org/urdf.

[20] Open Robotics. xacro - ros wiki. http://wiki.ros.org/xacro.

[21] Open Robotics. ros_contollers - ros wiki. http://wiki.ros.org/ros_controllers.

[22] Open Robotics. gmapping - ros wiki. http://wiki.ros.org/gmapping.

[23] Open Robotics. amcl - ros wiki. http://wiki.ros.org/amcl.

[24] Open Robotics. move_base - ros wiki. http://wiki.ros.org/move_base.

[25] Kaiyu Zheng. Ros navigation tuning guide. *arXiv preprint arXiv:1706.09068*, 2017.

[26] Open Robotics. actionlib - ros wiki. http://wiki.ros.org/actionlib.

[27] Open Robotics. Names - ros wiki. http://wiki.ros.org/Names.

[28] Open Robotics. teleop_twist_keyboard - ros wiki. http://wiki.ros.org/teleop_twist_keyboard.

付録 A　実験の再現方法

本研究で実装したロボットシステムおよびサービスアプリケーションを動作させるための方法を，環境構築，環境地図，ロボットサービスのシミュレーションの項目に分けて述べる．複数のターミナルを用いるため，別のターミナルで実行する際にはターミナル 1，ターミナル 2…のように番号を付ける．

A.1　環境構築

1. ROS をインストールする．バージョンは melodic を推奨する．
2. ROS ワークスペース用のディレクトリを作成し，ディレクトリ内へ移動する．

```
$ mkdir gazebo_RoIS_ws && cd gazebo_RoIS_ws
```

3. リポジトリを src という名前でクローンする．

```
$ git clone \
  https://inohira.mns.kyutech.ac.jp/git/tanacchi/gazebo_RoIS_tanacchi.git \
  src
```

4. catkin_init_workspace コマンドを実行する．

```
$ cd src && catkin_init_workspace
```

5. インストールスクリプトを実行する．

```
$ sh install_packages.sh
```

6. ワークスペース内のパッケージをビルドする．

```
$ cd ../ && catkin_make
```

7. ワークスペースをインストール環境にオーバーレイする．
   これによってワークスペース内のパッケージにパスが通るようになり，rosrun コマンド等によるプログラムの起動が可能となる．

   ```
   $ source devel/setup.bash
   ```

   このコマンドは， .bashrc ファイルを編集するなどして，自動実行されるように設定することが推奨される．

A.2 環境地図の作成

1. ターミナル 1 でシミュレーション環境，ロボットモデルを一台分起動する．

   ```
   $ roslaunch rois_gazebo setup_gazebo.launch use_multi_robot:=false
   ```

2. ターミナル 2 でキーボード入力によるロボット操作のための ROS ノードを起動する．ロボットのモデル操作はこのターミナルにフォーカスした状態でキーボードを操作する．キーボード操作の方法は teleop_twist_keyboard の公式サイトを参照されたい．

   ```
   $ roslaunch rois_gazebo teleop_keyboard.launch robot_name:=robot1
   ```

3. ターミナル 3 で bag ファイルを生成コマンドを実行する．bag は，ROS におけるメッセージングデータを蓄えるためのファイルフォーマットで，ROS システムの動作中に bag ファイルでメッセージングの状態を記録し，後に再生することで メッセージのやりとりを再現することができる．

   ```
   $ rosbag record --all -O <bag ファイル名>
   ```

4. ターミナル **2** でキーボード操作し，ロボットを操作しながら一通り環境を走破する．

5. ターミナル 3 の rosbag コマンドをキルした後，他のターミナルで実行しているプロセスもキルする．

6. ターミナル 1 で，自動生成用の launch ファイルを実行する．コマンドライン引数で bag ファイルを指定し，それを再生しながら slam_gmapping ノードで環境地図を生成する．

```
$ roslaunch rois_2dnav_gazebo generate_2dmap.launch \
  bagfile:=<bag ファイルへの絶対パス>
```

7. 手順 6 のプロセスをキルせずに，生成された環境地図をファイルに保存する．

```
$ rosrun map_server map_saver -f <map ファイル名>
```

A.3 ロボットサービスのシミュレーション

1. ターミナル 1 で，シミュレーション環境，ロボットモデル，自律移動のための ROS ノードを起動する．use_multi_robot 引数を true にすると複数ロボット用の，false にすると単一ロボット用のシステムが起動する．true の場合は省略可能．

```
$ roslaunch rois_bringup navigation.launch \
  use_multi_robot:=(true|false)
```

2. ターミナル 2 で，ユーザのモデル，ユーザの自己位置推定，キーボード操作のための ROS ノードを起動する．ユーザのモデル操作はこのターミナルにフォーカスした状態でキーボードを操作する．キーボード操作の方法は teleop_twist_keyboard の公式サイトを参照されたい．

```
$ roslaunch rois_gazebo spawn_human.launch
```

3. ターミナル 3 で，route_guidance パッケージの scripts ディレクトリに移動する．

```
$ roscd route_guidance_ros/scripts/
```

4. ターミナル 3 で，Python3 インタプリタから RoIS Framework における HRI エンジンと HRI コンポーネントを起動する．

名称	座標 [m]
point_a	(25, 20, 0)
point_b	(25, 10, 0)
point_c	(25, 0, 0)
point_d	(15, 0, 0)
point_e	(10, 0, 0)
point_f	(0, 0, 0)
point_g	(0, 0, 0)
point_h	(0, 10, 0)
point_i	(0, 20, 0)
point_j	(0, 20, 0)

表 A.3.1: コの字型の環境における有効な目的地の名称と座標の関係

```
>>> import utilities
>>> utilities.setup_single_robot()  # 単一ロボットの場合
>>> utilities.setup_multi_robot()   # 複数ロボットの場合
```

5. ターミナル 4 で，サービスアプリケーションを実行する．route_guidance パッケージの scripts ディレクトリに移動する必要があることに注意．

```
$  python3 service_app_lv2.py
```

6. ターミナル 2 にフォーカスしキーボードからユーザを少し操作して human1/amcl ノードに自己位置推定結果をパブリッシュさせる．パブリッシュされ次第，ユーザの位置に向かってロボットの自律移動が開始される．

7. ターミナル 5 で，Speech Recognition コンポーネントに文字列を送信する．"point_a" は送信される文字列である．

```
$ rostopic pub /robot1/detected_text std_msgs/String "data: 'point_a'"
```

送信された文字列が目的地の名称として有効であれば，その地点に向かって自律移動が開始される．ここで 3.3.1.1 で述べたコの字型の環境において，有効な目的地の名称と座標の関係を A.3.1 に示す．

付録B　リポジトリのファイル構成

3.3 で実装したプログラムは全て，
https://inohira.mns.kyutech.ac.jp/git/tanacchi/gazebo_RoIS_tanacchi
にて管理されている．本研究で実装したプログラム，設定ファイルなどの構成をリポジトリのルートディレクトリと ROS パッケージそれぞれについて分けて説明する．説明に値しない，自動生成ファイルや README などのファイルについては省略する．また，ファイル名との区別のために，ディレクトリ名の末尾には"/"を付ける．

B.1　gazebo_RoIS_tanacchi リポジトリ

gazebo_RoIS_tanacchi リポジトリは 5 つの ROS パッケージの集合 (メタパッケージ) である．5 つの ROS パッケージの他，動作する上で必要となる ROS パッケージと Python モジュールをインストールするためのシェルスクリプトが実装されている．

```
gazebo_RoIS_tanacchi/
├── README.md
├── install_packages.sh
├── rois_2dnav_gazebo/
├── rois_bringup/
├── rois_description/
├── rois_gazebo/
└── route_guidance_ros/
```

名称	説明
install_packages.sh	動作に必要なパッケージを一括インストールするためのシェルスクリプト
rois_2dnav_gazebo/	ROS を利用した自律移動に関するファイルをまとめた ROS パッケージ
rois_bringup/	システムの一括起動などのための ROS パッケージ
rois_description/	ロボットや屋内のレイアウトのモデルを格納するための ROS パッケージ
rois_gazebo/	Gazebo シミュレータを用いたシミュレーション環境の起動のための ROS パッケージ
route_guidance_ros/	RoIS Framework を用いた道案内サービスのソースコード

表 B.1.1: リポジトリのルートディレクトリのファイル構成

B.2 rois_2dnav_gazebo パッケージ

rois_2dnav_gazebo パッケージは，ROS の Navigation Stack に沿って実装した，本研究で実装したロボットシステムにおける自律移動のための ROS パッケージである．move_base, amcl ノードの設定ファイル，起動ファイルの他，本研究で使用した環境地図のファイルも格納されている．

```
rois_2dnav_gazebo/
├── config/
│   ├── amcl_params.yaml
│   ├── move_base_params.yaml
│   ├── costmap/
│   │   ├── robot1/
│   │   │   ├── costmap_common_params.yaml
│   │   │   ├── global_costmap_params.yaml
│   │   │   └── local_costmap_params.yaml
│   │   ├── robot2/
│   │   │   └── (robot1 と同様)
│   │   └── robot3/
│   │       └── (robot1 と同様)
│   └── planner/
```

```
│       ├── base_global_planner_params.yaml
│       ├── base_local_planner_params.yaml
│       └── recovery_behaviors.yaml
├── launch/
│   ├── amcl.launch
│   ├── generate_2dmap.launch
│   └── move_base.launch
└── map/
    ├── c_corridor/
    │   ├── c_corridor.pgm
    │   └── c_corridor.yaml
    └── lab_corridor/
        ├── lab_corridor.pgm
        └── lab_corridor.yaml
```

名称	説明
config/	自律移動の設定ファイル用ディレクトリ
amcl_params.yaml	amcl ノードの設定ファイル
move_base_params.yaml	move_base ノードの設定ファイル
costmap/	コストマップ計算のための設定ファイル用ディレクトリ
costmap_common_params.yaml	コストマップ計算の共通の設定ファイル
global_costmap_params.yaml	グローバルコストマップの計算のための設定ファイル
local_costmap_params.yaml	ローカルコストマップの計算のための設定ファイル
planner/	パスプランニングのための設定ファイル
base_global_planner_params.yaml	グローバルパスの計算のための設定ファイル
base_local_planner_params.yaml	ローカルパスの計算のための設定ファイル
recovery_behaviors.yaml	復帰動作を記述する設定ファイル
launch/	launch ファイル用ディレクトリ
amcl.launch	amcl ノードの起動用 launch ファイル
generate_2dmap.launch	環境地図生成のための launch ファイル
move_base.launch	move_base の起動用 launch ファイル
map/	マップファイル用ディレクトリ
c_corridor/	コの字型の環境の環境地図用のディレクトリ
lab_corridor/	実際の屋内を参考にした環境の環境地図

表 B.2.1: rois_2dnav_gazebo パッケージのファイル構成

B.3 rois_bringup パッケージ

rois_bringup パッケージは，シミュレーション環境やロボットモデルのスポーン，自律移動のための ROS ノードなどの一括で起動するための包括的な launch ファイルを格納するための ROS パッケージである．実際は，包括的な役割を持つ launch ファイルは navigation.launch のみとなっている．

```
rois_brinup/
└── launch/
    └── navigation.launch
```

名称	説明
launch/	launch ファイル用ディレクトリ
navigation.launch	シミュレーション環境，ロボットモデル， 自律移動のための ROS ノードを 一括起動する launch ファイル

表 B.3.1: rois_bringup パッケージのファイル構成

B.4 rois_description パッケージ

rois_description パッケージは，ロボットや実験環境などのモデルを扱う ROS パッケージである．ロボットのモデルを記述する URDF ファイルは，実機での動作のデバッグにも用いられることもあるため，rois_gazebo パッケージとは分けている

```
rois_description/
├── robot/
│   ├── human1.urdf.xacro
│   ├── robot1.urdf.xacro
│   ├── robot2.urdf.xacro
│   ├── robot3.urdf.xacro
│   └── roisbot.urdf.xacro
└── world/
    ├── c_corridor.sdf
    └── lab_corridor.sdf
```

名称	説明
robot/	ロボットモデルのファイル (URDF) 用ディレクトリ
*.urdf.xacro	ロボット，ユーザのモデルを記述するファイル
world/	屋内のモデルのファイル用ディレクトリ
*.sdf	屋内のモデルを記述するファイル

表 B.4.1: rois_description パッケージのファイル構成

B.5 rois_gazebo パッケージ

rois_gazebo パッケージは，モデルを Gazebo 上で動作させるための設定ファイルや，シミュレーション環境を起動したりモデルをスポーンしたりといった機能を持つ launch ファイルなど，Gazebo シミュレータに関係するファイルを扱う ROS パッケージである．3 台のロボットそれぞれのモデルのデザインの変更に対応するため，コントローラファイルは敢えて別々にしている．

```
rois_gazebo/
├── config/
│   ├── human1_controller.yaml
│   ├── robot1_controller.yaml
│   ├── robot2_controller.yaml
│   └── robot3_controller.yaml
└── launch/
    ├── setup_gazebo.launch
    ├── spawn_human.launch
    ├── spawn_model.launch
    └── teleop_keyboard.launch
```

名称	説明
config/	Gazebo 上でモデルを動作させるための設定ファイル用ディレクトリ
*_controller.yaml	モデルそれぞれの，Gazebo 上で動作させるための設定ファイル
launch/	launch ファイル用ディレクトリ
setup_gazebo.launch	Gazebo シミュレータを起動する launch ファイル
spawn_human1.launch	ユーザのモデルのスポーン，自己位置推定の開始， キーボード操作のための ROS ノード起動のための launch ファイル
spawn_model.launch	ロボットのモデルのスポーンのための launch ファイル
teleop_keyboard.launch	モデルをキーボード操作するための launch ファイル

表 B.5.1: rois_gazebo パッケージのファイル構成

B.6 route_guidance_ros パッケージのファイル構成

RoIS Framework を用いて実装された，道案内サービスのソースファイルを格納した ROS パッケージである．本研究で実装した道案内サービスで使用する HRI コンポーネントは，全て ROS ノードとして動作させるためのものとして実装されているため，それらも ROS パッケージ内で管理する必要がある．サービスアプリケーションのソースファイルは，ROS パッケージ内に置く必要はないが，管理のしやすさのために同じディレクトリ内に置いている．

```
route_guidance_ros/
├── requirements.txt
└── scripts/
    ├── Dummy_Speech_Recognition.py
    ├── Dummy_Speech_Recognition_client.py
    ├── Navigation.py
    ├── Navigation_client.py
    ├── Person_Localization.py
    ├── Person_Localization_client.py
    ├── System_Information.py
    ├── System_Information_client.py
```

```
├── VirtualNavigation.py
├── VirtualNavigation_client.py
├── field_points.yaml
├── goal_sender_ros.py
├── main_engine.py
├── main_engine_client.py
├── service_app_lv1.py
├── service_app_lv2.py
├── sub_engine.py
├── sub_engine_client.py
└── utilities.py
```

名称	説明
requirements.txt	依存する Python モジュールを記述するファイル
scripts/	道案内サービスのソースファイル用ディレクトリ
Dummy_Speech_Recognition.py	Speech Recognition コンポーネントのサーバ
Dummy_Speech_Recognition_client.py	Speech Recognition コンポーネントのクライアント
Navigation.py	Navigation コンポーネントのサーバ
Navigation_client.py	Navigation コンポーネントのクライアント
Person_Localization.py	Person Localization コンポーネントのサーバ
Person_Localization_client.py	Person Localization コンポーネントのクライアント
System_Information.py	System Information コンポーネントのサーバ
System_Information_client.py	System Information コンポーネントのクライアント
VirtualNavigation.py	仮想 Navigation コンポーネントのサーバ
VirtualNavigation_client.py	仮想 Navigation コンポーネントのクライアント
field_points.yaml	目的地の名称と座標の対応を記述するファイル
goal_sender_ros.py	actionlib クライアントのラッパー
main_engine.py	複数ロボット用 HRI エンジンのサーバ
main_engine_client.py	複数ロボット用 HRI エンジンのクライアント
service_app_lv1.py	プロトタイプのサービスアプリケーション
service_app_lv2.py	改良版のサービスアプリケーション
sub_engine.py	単一ロボット用 HRI エンジンのサーバ
sub_engine_client.py	単一ロボット用 HRI エンジンのクライアント
utilities.py	ロボットシステム起動用のモジュール

表 B.6.1: route_guidance_ros パッケージのファイル構成