Taking Recoveries to Task: Recovery-Driven Development for Recipe-based Robot Tasks

Siddhartha Banerjee*, Angel Daruna*, David Kent*, Weiyu Liu*, Jonathan Balloch, Abhinav Jain, Akshay Krishnan, Muhammad Asif Rana, Harish Ravichandar, Binit Shah, Nithin Shrivatsav, and Sonia Chernova

Georgia Institute of Technology, Atlanta GA 30332, USA,
Corresponding authors:
{siddhartha.banerjee,dekent,adaruna3,wliu88}@gatech.edu

Abstract. Robot task execution when situated in real-world environments is fragile. As such, robot architectures must rely on robust error recovery, adding non-trivial complexity to highly-complex robot systems. To handle this complexity in development, we introduce Recovery-Driven Development (RDD), an iterative task scripting process that facilitates rapid task and recovery development by leveraging hierarchical specification, separation of nominal task and recovery development, and situated testing. We validate our approach with our challenge-winning mobile manipulator software architecture developed using RDD for the FetchIt! Challenge at the IEEE 2019 International Conference on Robotics and Automation. We attribute the success of our system to the level of robustness achieved using RDD, and conclude with lessons learned for developing such systems.

Keywords: failure recovery, robot architectures, mobile manipulation, design and prototyping

1 Introduction

Robot execution is fragile and often overfits to the development test bed [2, 9]. As such, robust robot architectures must rely on recovery behaviors in order to maintain autonomy when assumptions are violated [20]. Recovery during robot tasks is non-trivial, however, as resetting to a known state can be difficult [16] and knowing where to resume execution can be context dependent [5]. In addition, unforeseen faults create ambiguity in recovery strategies.

In this work, we address the development of robust recovery for recipe-based tasks—a class of robot tasks that dictate a pre-specified sequence of steps to accomplish a goal. Such tasks include common mobile manipulation tasks in unstructured environments, such as kit packing, machine assembly, table setting, or food preparation. Even for seemingly straightforward recipe-based tasks, the many interactions with the environment, and between robot components, lead to faults that are difficult to identify *a priori* [11], often resulting in systems that are inflexible or not robust to failures.

* Indicates equal contribution

State machines [5], hybrid automata [10], and planning approaches [3] are common methods of sequencing robot execution that can be made robust to failures. However, robustness is often achieved at the cost of a complexity explosion in the task sequence specification or a loss of interpretability of the task recipe. Crucially, the increased complexity and the lack of interpretability negatively impact the iterative development of the main task and recovery processes, both of which are necessary in the face of potentially innumerable failure conditions.

We therefore propose *Recovery-Driven Development* (RDD), a development process for recipe-based tasks couched in agile methodology. The key tenet of RDD is the separation of nominal task specification from recovery behavior definition. Another guiding principle of RDD is the support of hierarchical task specification, which both allows for re-use in the task recipe and provides higher-level context to recovery behavior selection. As such, the RDD methodology enables system developers to easily explore aspects of a robot's system design, such as those identified by Eppner et al. [10]—assumptions, generality, modularity, etc.

We define RDD as a 2-pronged iterative approach to developing robust task execution, in which designers can move back and forth between both prongs without risk of one phase interfering with the other:

1. **Specification** (Section 4.1): scripting a hierarchical task sequence incrementally from a task recipe, using strong assumptions
2. **Refinement** (Section 4.2): developing recovery behaviors by executing a situated task, noting a fault, specifying new recoveries, and repeating

The result is a development methodology that supports rapid task and recovery prototyping, without a noticeable loss in the robot's robustness when deployed.

In this work, we present our task execution and monitoring system as an example framework designed to enable RDD for recipe-based robot tasks[1]. We validate both the task system and the RDD workflow with our team's winning approach to the FetchIt! Challenge at the IEEE 2019 International Conference on Robotics and Automation (ICRA), in which our success was achieved mainly due to the robustness afforded to our system from the RDD methodology. Additionally, we provide details and open-source code for our complete system developed for the FetchIt! Challenge as a concrete example of a complex mobile manipulation system developed using RDD. We conclude with a discussion of lessons learned for the fast and robust development of recipe-based robot tasks.

2 Related Work

Designing a robot's software is often application-dependent, requiring tradeoffs between multiple approaches [14]. In this section we enumerate common design choices that emerge across applications for robot architectures, situate our task execution framework within the design practices, and motivate the development of our approach.

Robot architectures are generally three-tiered with the following levels [14]:

[1] https://github.com/GT-RAIL/derail-fetchit-public/tree/master/task_execution

- A *behavioral* level for highly reactive and highly situated robot execution. Modules at this level, sometimes termed skills, have a tight perception-action loop and are the focus of much research.
- An *executive* level that bridges low-level tasks (skills) and high-level tasks (goals). The executive is responsible for sequencing skills, monitoring execution, and handling exceptions.
- A *planning* level responsible for tasking the executive level with goals to achieve based on future objectives, robot constraints, environmental situations, etc.

Our primary contribution is in enabling the executive level to support an RDD workflow, and as such the remainder of this section examines executive level design and recovery. We discuss a behavioral implementation of mobile manipulation in Section 3 to provide context for our executive implementation. We also note that planning-level requirements are minimal for autonomous recipe-based tasks, although we return to this assumption at the end of Section 6.

2.1 Executive Level Design

The most informative consideration in executive level design is how dynamic or static the task should be. A task can be dynamic due to environments with uncontrolled agents such as humans [4] or competing objectives [21]. There are four paradigms to behavior sequencing at the executive level, with differing levels of support for dynamic tasks:

- *Agent-based control* partitions control into separate, synchronized agents that maintain consistency with the global robot objective. This works well for dynamic environments, and was implemented through behavior trees in Playful [4] and resource agents in ROAR [8]. However, debugging and reasoning about the interactions between agents can be difficult.
- *Planning* is a principled manner of sequencing skills in dynamic environments, and was implemented by CRAM [3]. However, when designers know the exact sequence of skills they want, as in recipe-based tasks, the design process for planning can be non-intuitive or even counter-productive [5].
- *Finite State Machines* retain some of the autonomy in sequence specification provided by planning, and also allow system designers to explicitly specify state transitions a priori based on expected sub-task outcomes. Additionally, state machines support model verification and composition for incremental construction of complex behaviors [13, 5]. However, state machines, and the related method of hybrid automata [10], suffer from an explosion of transitions as the number of skills or the task complexity grows.
- *Scripting* allows for the compositionality of state machines with the simple declaration, rather than programming, of robot behavior [20]. Additionally, scripting provides easier error recovery to handle exceptions at the executive level than state machines. However, scripting puts the burden of sequence specification on the designer, raising scalability issues, especially for multi-objective tasks [5].

In this work, we focus on relatively static environments and recipe-based tasks that can be decomposed into subtasks. In order to facilitate the rapid prototyping and incremental inclusion of error recoveries inherent to RDD, we sequence behaviors through hierarchical scripting. We address scripting's scalability issues by separating task specification and recovery.

2.2 Reactivity and Recovery

A robust robot executive level must include failover mechanisms that maintain autonomy when behavior design assumptions are not satisfied [20]. Therefore, recovery systems must address many challenges, including determining how to reset to a known state [16], handling context dependent execution resumption [5], and deciding on recovery strategies for unforeseen faults. A common recovery strategy for resetting to a known state is to re-attempt the entire task, as in [10], although such approaches are less reactive to failures.

Planning approaches maintain reactivity by recovering from seen and unforeseen failures by replanning [3]. Further, plans provide theoretical guarantees on robustness to unforeseen execution failures [15, 10]. Prior works have treated recovery as a planning problem with the goal of reaching any state where a diagnosed fault does not exist [24]. However, as noted earlier, planning approaches can be difficult to iteratively develop, or can result in not-easily interpretable task specifications. Our approach avoids planning at the executive layer in favor of scripting, to facilitate rapid and highly-interpretable behavior development.

In the absence of planning, reactively resetting to a good known state can be accomplished through fault forecasting, such as FMECA, which reasons about expected faults and the explicit recovery steps to address them [7]. However, such approaches are time consuming and not guaranteed to find all faults [11]. We instead take an empirical approach to fault discovery through situated task execution, exploiting the inherent structure of recipe-based tasks, allowing for pre-scripted recovery to intermediate task steps.

3 System Overview

Before presenting our task execution and recovery system, we first describe the behavioral level of our general mobile manipulation architecture, to serve two purposes: (1) to share our open-source challenge-winning mobile manipulation system developed to support RDD, and (2) to establish a task context and a set of robot capabilities that we will refer to throughout the paper, grounding our discussion of RDD's benefits and drawbacks in a fully-realized robot system. The architecture consists of a set of independent mobile manipulation modules, implemented using the Robot Operating System (ROS) [18], shown in Figure 1. Object perception modules are implemented as ROS service servers, and object manipulation and base navigation modules are implemented as `actionlib`[2] servers. Each independent module can be called by the task executor, and pro-

[2] http://wiki.ros.org/actionlib

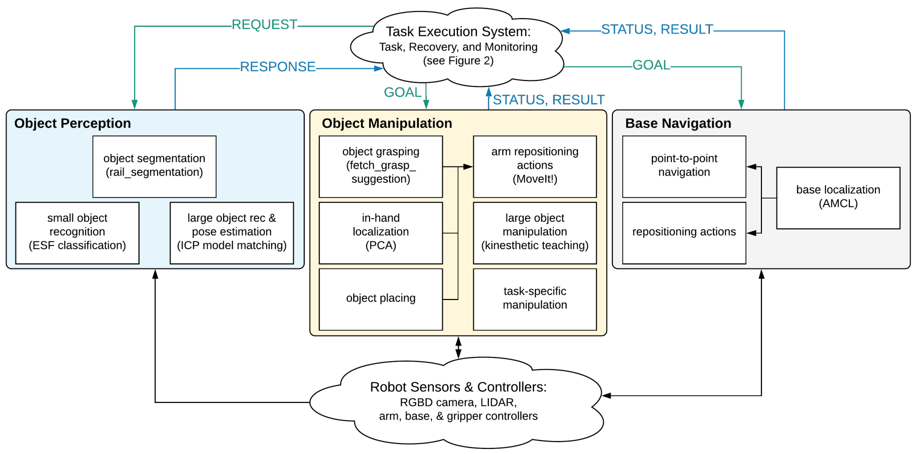

Task Execution System:
Task, Recovery, and Monitoring
(see Figure 2)
REQUEST
RESPONSE
GOAL
STATUS, RESULT
STATUS, RESULT
GOAL
Object Perception
object segmentation (rail_segmentation)
small object recognition (ESF classification)
large object rec & pose estimation (ICP model matching)
Object Manipulation
object grasping (fetch_grasp_ suggestion)
arm repositioning actions (MoveIt!)
in-hand localization (PCA)
large object manipulation (kinesthetic teaching)
object placing
task-specific manipulation
Base Navigation
point-to-point navigation
repositioning actions
base localization (AMCL)
Robot Sensors & Controllers:
RGBD camera, LIDAR,
arm, base, & gripper controllers

Fig. 1: Mobile manipulation system overview. Arrows denote ROS information flow, through publishers, subscribers, services, and actionlib.

vides feedback to the task executor and task monitor[3]. The modules consist of the following capabilities:

Object Perception. Our perception modules implement a perception pipeline for RGBD sensor data, using the Point Cloud Library (PCL) [19]. The object segmentation module uses the `rail_segmentation`[4] package to identify point cloud clusters-of-interest through table surface detection and Euclidean distance clustering. We divide our object recognition approaches between large and small objects. For large objects, we perform model matching using the `rail_mesh_icp`[5] package that uses an Iterative Closest Point (ICP) PCL pipeline, which also provides object pose detection. For small object recognition, we train an SVM classifier over Ensemble of Shape Functions (ESF) descriptors [23]. We do not need to perform pose estimation for small objects due to our object grasping approach, described below.

Object Manipulation. Most of our manipulation modules make use of MoveIt! to perform arm planning to either joint goals or end-effector pose goals using OMPL's `RRTConnect` motion planner [6]. This includes both general arm repositioning actions, which the task executor can call directly (e.g. to move the arm out of the way of the camera), and execution actions, called by other object manipulation modules. Object grasping calculates antipodal grasps over an object point cloud using the `agile_grasp` package [17], which are then ordered and executed using pairwise ranking through `fetch_grasp_suggestion` [12]. As objects can shift during the grasping process, we perform post-grasp pose detection using the in-hand localization module, which identifies the object point cloud by performing background subtraction on the robot's gripper, and calculates the

[3] Each module must necessarily provide feedback on its own faults so that the executive level can make relevant recovery decisions.
[4] http://wiki.ros.org/rail_segmentation
[5] http://wiki.ros.org/rail_mesh_icp

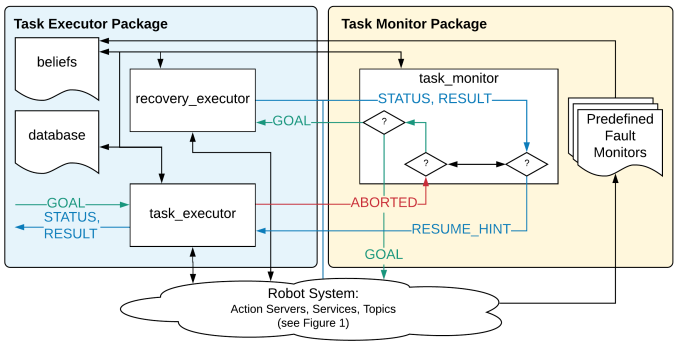


Fig. 2: Overview of the two packages in our the executive level. Arrows denote ROS information flow, through publishers, subscribers, services, and actionlib.

object's pose based on its principal axes determined by Principal Component Analysis (PCA). Given a known object pose and a desired place location, object placing calculates and executes a pose goal for placing an object that ensures the gripper fingers and palm are out of the way of the object's fall trajectory.

We also include some manipulation modules that do not use MoveIt!, due to the limitations of sampling-based motion planning. For large object manipulation, such as lifting and placing kits of objects, we include a kinesthetic teaching module [1]. This allows system designers to record and play back arm trajectories, either in full or as a set of waypoints. Additionally, we include task-specific manipulation actions to implement specific manipulation skills such as raising and lowering objects, using a Cartesian end-effector controller[6], and peg-in-hole insertion, using a controller with end-effector pose and joint effort as feedback.

Base Navigation. LIDAR-based localization uses AMCL provided by ROS's `nav_stack`[7] to localize the base with respect to a pre-collected 2D occupancy grid of the environment. Navigation is primarily done using point-to-point navigation between waypoints on the map, executed using a PID controller[8]. We also include local repositioning actions, which implement short movement primitives such as backing up from a table. The repositioning actions are implemented using a PID controller with gains tuned for shorter, more precise base goals.

With each module implemented, the navigation, perception, and manipulation actions can be sequenced in a robust manner to complete mobile manipulation tasks by the task execution system described in the next section.

4 Task Execution and Recovery

In this section, we describe our executive level, which consists of two packages seen in Figure 2: the *task executor* and the *task monitor*[9]. The task executor (Section 4.1) contains scaffolding to specify and incrementally develop a main

[6] Available at https://github.com/GT-RAIL/fetch_simple_linear_controller
[7] http://wiki.ros.org/navigation
[8] In complex environments, `nav_stack`'s global and local planners can be used instead.
[9] Stand-alone packages under development at https://github.com/GT-RAIL/assistance_arbitration

task recipe. The task monitor (Section 4.2) contains the utilities necessary recover from general failures during task execution.

4.1 RDD Specification: The Task Executor

During the *Specification* phase of RDD, developers translate the nominal behaviour of the robot executing a task recipe into a script. Crucially, developers should fulfill two objectives in this phase: (1) declare robot behavior under the strong assumption of perfect robustness in execution, and (2) provide structure to the specified script so that in the event of an error, it is easy to garner error context as well as resume execution once the error is resolved. The task executor package facillitates meeting such objectives.

Specifically, the task executor provides the following utilities to aid in rapid task prototyping and testing:

- A Python-based abstraction for specifying semantically meaningful interfaces to the robot's behavior layer, to form sequenceable primitive actions.
- A custom domain-specific language using YAML syntax for scripting recipe-based tasks, with a view towards facilitating hierarchical task declaration for code modularity and reuse.
- A consistent API to tasks and actions to facilitate testing in isolation and to enable easy invocation from other system components.
- A database to provide a common knowledge-base of task relevant information to all tasks and primitive behaviors.
- An automatically populated belief system encapsulating pertinent robot, environment, and task states, to provide additional context during recovery.

The following sections provide additional details on the above utilities.

Actions Primitive actions are specified as Python objects derived from a common abstract class. They are implemented either as a client to individual robot components, such as to point-to-point navigation, or as a client to semantic groupings of robot components, such as to the grasp calculation packages.

Tasks Tasks manifest as a Python class derived from the same abstract class as *actions*, but whose execution is specified using a custom domain-specific language, which uses YAML syntax, to allow loading and reloading of tasks from the ROS parameter server. The language allows tasks to:

- reuse other tasks for the creation of complex task hierarchies
- accept parameters for adaptation and compositionality in task specification
- create, maintain, and manipulate local variables for data transfer between actions and for adaptation to environmental or execution conditions
- utilize rudimentary control flow through conditional statements and loops, aiding in concise task specification

We present the formal task syntax in Listing 1, with example tasks used for large object pose estimation and for object picking at the FetchIt! Challenge (Section 5.1) shown in Listing 2. A task is defined as a dictionary entry with

```
(task name):
  [params: [...]]
  [var: [...]]
  steps:
  - (action | task | op | choice | loop) : (name)
    [params: {...}]
    [var: [...]]

  [...]
```

Listing 1: Task Syntax

```
detect_schunk_pose_task:
  params:
  - look_location

  var:
  - chuck_approach_pose

  steps:
  - action: look
    params:
      pose: params.look_location

  - action: detect_schunk
    var:
    - chuck_approach_pose
```

```
pick_task:
  params: [object_idx, grasps, object_key]
  var: [grasped]
  steps:
  - action: pick
    params:
      object_idx: params.object_idx
      grasps: params.grasps
      object_key: params.object_key

  - action: verify_grasp
    params:
      abort_on_false: false
    var:
    - grasped
```

Listing 2: Example Tasks

the required key of `steps`, which defines a list of the steps in the task, and the optional keys of `params` and `var`, which define the task inputs and outputs. Each `step` in the task is named and can be one of five types: (1) `action`, invoking a primitive action, (2) `task`, invoking another task, (3) `op`, invoking a simple Python function for rudimentary data manipulation, (4) `choice`, to evaluate a boolean expression for control flow, and (5) `loop`, to loop while a boolean expression is `true`. All steps accept a dictionary setting values for their `params`, and return a list of `var` values that become local variables in the parent task[10].

Consistent API Tasks and actions have a consistent API, which mimics that of ROS's `actionlib` interface. This consistent API enables (1) the use of JSON to specify inputs and outputs to tasks and actions easily in order to test them in isolation, and (2) the invocation of individual tasks from other ROS nodes, such as the recovery system, through the `actionlib` interface when required.

Database Recipe-based tasks can be parameterized by semantically meaningful task variables which are then grounded to different values for particular environments or tasks. Example task variables are locations, robot poses, objects, or other real-world entities. The database is a YAML dictionary loaded into the ROS parameter server that provides a single source of truth for grounding all relevant task variables. Rapid environment adaptation is readily facilitated by modifying the values associated with known keys in the database definition.

Beliefs Beliefs are included in the task executive layer for two reasons: (1) to provide context to recovery mechanisms in the event of a failure, and (2) to provide updates to a higher level planner, should one exist, about relevant states of the task, the robot, or the environment. For instance, the expected and actual state may become mismatched: transient localization and navigation errors during point-to-point navigation might compound to leave the robot at a location outside an expected tolerance for manipulation actions. Background monitors on the robot's location can indicate a mismatch, and in turn the recovery system can use this information to reposition the robot.

[10] For more details, see the README in our Github repository.

Discussion The task executor package is optimized to facilitate rapid specification and testing of recipe-based tasks: developed task scripts are deterministic, easy to specify, interpretable, and readily allow the testing of components in isolation. Additionally, the scripting approach to task specification provides the implicit benefits of straightforward state tracking and an efficiency in task execution borne from overestimating the robustness of the robot's behaviors. Indeed, we do not check for most violations to the operating conditions of our primitive behaviors until they report a failure.

We note that the determinism of our scripting approach and overestimation of the robustness of our behaviors leaves us susceptible to violations in assumptions of the environmental state (a susceptibility that reactive sequencing approaches do not share). However, instead of complicating the task scripts and in turn slowing down task specification, our RDD methodology relies on the incremental recovery development to achieve robustness and reactivity.

4.2 RDD Refinement: The Task Monitor

The primary objective of system development during the *Refinement* phase of RDD is to rapidly incorporate diverse recovery strategies for a specified task recipe in order to incrementally improve its robustness. As such, it involves addressing four challenges (mentioned in Section 2):

1. resolving ambiguity in the recovery policy for unforeseen faults
2. taking actions to reset to a known state in the event of a fault
3. deciding how to resume execution once a fault is addressed
4. trying diverse strategies when recovering from a repeated fault

The task monitor package, which provides execution monitoring and error recovery to the task executor, is designed to address each of the above challenges.

Handling Unseen Errors In the event of an unseen error during development[11], the monitor immediately exits from the task, displaying the entire context of the error in a consistent manner and logging all possible causes. Developers can then inspect the logs to create a tailored (set of) recovery mechanism(s) for such errors, thereby making them "known" errors during future failures. In practice, we quickly accumulate a list of errors that our developed recovery strategies know to address.

During deployment, unseen errors can be dealt with under a domain-dependent context-relevant policy of always exit, always retry, or some combination thereof.

Taking Actions Rapidly developing and testing actions to take in the event of a particular failure requires the presence of (1) a means of determining the diagnosis of an error, and (2) an easy mechanism to invoke actions or subsets of actions. The monitor and task executor are designed to facilitate both.

[11] The system can detect unseen errors in three ways: (1) the behavior level can propagate reported faults (i.e. action servers aborting, nodes crashing, etc.), (2) recipe steps can explicitly check for expected errors, or, in the case of unexpected errors, (3) the developer can stop system execution and write a new error detection module.

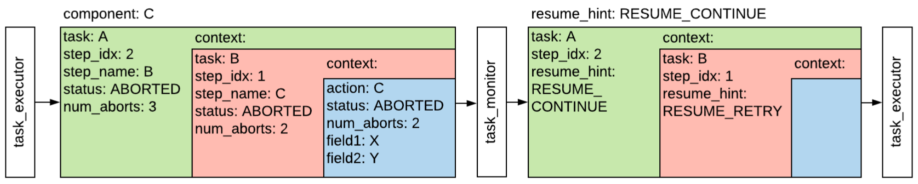


Fig. 3: Metadata passed between the task executor and the task monitor, used to facilitate error diagnosis and task resumption after fault resolution.

The structure and compositional design of our main task recipe aids in fault diagnosis, given the current task state. Specifically, in the event of an error, tasks in the task executor provide a consistent context of their state in a recursive dictionary containing all tasks in a task hierarchy until the primitive action, and primitive actions can also provide error context through custom data fields, as shown in Figure 3. Further, the task executor's beliefs (Section 4.1) can additionally inform error recovery.

When an error is diagnosed, the consistent API for invoking tasks and actions facilitates the monitor in resolving the problem. The monitor uses a redundant instance of the task executor, called the recovery executor (seen in Figure 2), to execute simple task recipes for recovery.

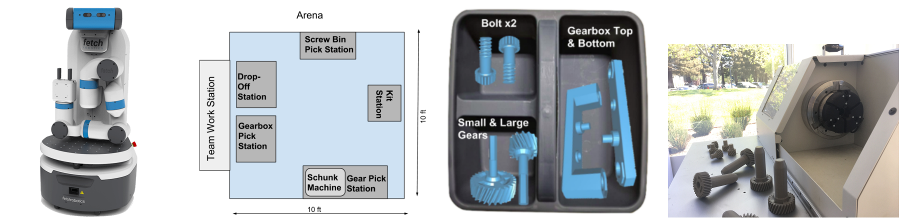
Resuming We have identified five strategies for resuming task execution that have applied to the errors we have encountered:

1. `RESUME_NONE`: stop executing the task.
2. `RESUME_CONTINUE`: resume task execution from the failed step.
3. `RESUME_RETRY`: restart a subtask, or the whole task; useful if, for example, the environment changed during recovery and thus perception must be rerun.
4. `RESUME_NEXT`: resume execution at the next step; useful if the recovery process accomplishes the failed step.
5. `RESUME_PREVIOUS`: resume execution at the previous step; useful if failure assumptions change, but the entire subtask does not need to be restarted.

To enable full flexibility in the recovery mechanism on how tasks[12] are resumed, any of the tasks in a hierarchy can be resumed using any of the above five strategies. An example of the context for task resumption is shown in Figure 3.

Recovery Diversity Due to the larger context of some errors, the same recovery actions taken during the same fault diagnosis can fail: for example, recalculating grasps on a small object when sampling-based arm motion planning fails to pick it up may be insufficient due to an arm workspace limitation, and instead the error should be resolved by repositioning the robot base or by moving the arm to a different start configuration. As such, the context dictionaries included for diagnosis and resumption support development of diverse recovery strategies

[12] Resuming execution from arbitrary stopping points in primitive actions is hard [16], but depending on the implementation of the robot system, might be unnecessary.



(a) Fetch Robot (b) Challenge Arena (c) Assembled Kit (d) Schunk Machine

Fig. 4: FetchIt! challenge hardware and specifications.

for the same faults, based on factors such as task hierarchy location, primitive action failure count, or hierarchical task failure counts.

Discussion The philosophy behind RDD's *Refinement* phase, i.e. incremental and independent recovery development, necessitated a recovery system that is deterministic, easy to specify, interpretable, and readily allows testing of individual recoveries in isolation. Although the current version of our implemented recovery system is not robust to failures during the recovery process, such robustness can either be added in a future iteration of our system, or can be left to the purview of a higher-level planner in the robot system. Finally, we note that our current rule-based system of recoveries does not easily lend itself to analysis or verification, but such a feature can be integrated in the near future. In the meanwhile, the easy testability of individual recoveries mitigates the lack of verification in the system.

5 Validation

Our task execution approach formed our executive level for the FetchIt! Challenge at ICRA 2019. The challenge's goal was to advance autonomy and robustness by using a mobile manipulator to perform an industrial kit assembly task in an unstructured environment. As such, the challenge was a good opportunity to validate our system and development methodology in a real-world, time-sensitive scenario. We provide a brief description of the FetchIt! Challenge, followed by quantitative and qualitative observations of the RDD workflow and the recovery mechanisms we developed for our competition-winning robust task executor.

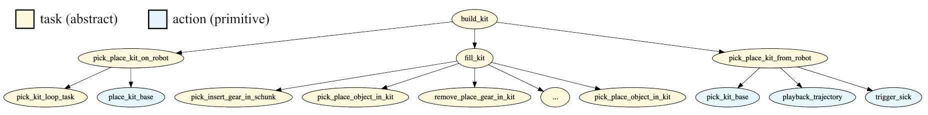
As further context, and to provide a continuous example of recoveries over a 45-minute autonomous task, we provide a video of our final competition run[13].

5.1 FetchIt! Challenge Overview

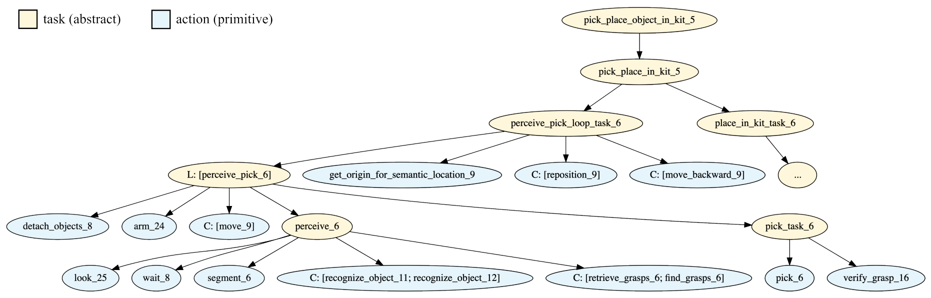
The FetchIt! Challenge was a mobile-manipulation challenge focused on autonomously completing combined manipulation, perception, and navigation tasks on a mobile-manipulator platform[14]. Specifically, the goal of the competition was to have a Fetch mobile manipulator [22], equipped with an RGBD camera and

[13] https://youtu.be/G_ur71h4CNQ

[14] https://opensource.fetchrobotics.com/assets/Rulebook2019.pdf



(a) The top three levels in the hierarchical task tree. Repetitions of *pick_place_object_in_kit* are omitted and denoted with an ellipsis for brevity.



(b) Full expansion of the fifth *pick_place_object_in_kit* task. The suffix n for each node indicates the n^{th} invocation of an action or task over the full task tree (n is often higher in practice due to recovery execution). Details of *place_in_kit_task* are omitted for brevity. C denotes a choice node that denotes conditional execution, and L denotes a loop node.

Fig. 5: Hierarchical task tree for the FetchIt! challenge.

a 2D LIDAR (Figure 6a), autonomously assemble kits (Figure 6c). In order to assemble each kit, the Fetch had to navigate a challenge arena (Figure 6b), perceiving and picking the various parts from tabletops and bins. In addition to pick and place, the Fetch had to operate machinery in the arena via physical manipulation and wireless interfaces as part of the assembly process. For instance, it had to insert the "Large Gear" in Figure 6c into a small opening shown in Figure 6d on a simulated milling machine. Perfectly completed kits (as in Figure 6c) scored 7 points, with no points awarded for incomplete kits (i.e. any parts missing).

The robot was required to run autonomously with no intervention for an allotted time of forty five minutes, completing as many kit assemblies as possible. The strict scoring requiring fully assembled kits made robust task execution one of the largest challenges of the competition.

5.2 Validation of Recovery-Driven Development

Figure 5a shows the high-level structure of the task implemented for the FetchIt! Challenge—an easy to understand script. Figure 5b demonstrates the task complexity, showing an expansion of one of the abstract tasks from Figure 5a. This complexity is manageable thanks to hierarchy-enabled action and task reuse–our task script reuses the look action at least 25 times and the perceive task at least 6 times. The modular nature of the task specification simplified the design process and allowed for independent testing of task recipes.

Further, the RDD requirement of separating specification and refinement allowed us to safely develop new recovery behaviors in high-pressure moments between competition runs. For instance, the simulated milling machine required gear insertion into a smaller hole than we had previously tested with, which caused new faults resulting from false positives in the robot's evaluation of the insertion. With only 45 minutes to test between runs, we were able to quickly update existing recovery strategies to identify insertion failure and retry the task, without risk of disrupting the previously tested nominal task flow.

5.3 Evaluation of Task Robustness

The FetchIt! Challenge provided an opportunity to gather quantitative data on our recovery strategies, allowing us to evaluate the error recovery utilities provided by the task monitor (Section 4.2). We also provide a representative example to qualitatively highlight those utilities.

Quantitative Observations We provide a breakdown of the recovery strategies we implemented for the FetchIt! Challenge. In total, we implemented 18 strategies, which often included multiple sub-strategies to handle dynamic execution under differing fault conditions.

To highlight the value of easy rule specification for recoveries and the ability to act upon a rule-based diagnosis, we define the following three situations:

1. *Shared Recovery*: different faults use the same rules for diagnosis and recovery
2. *Immediate Action*: recovery directly invokes a primitive action
3. *Dynamic Recovery*: in the same error diagnosis, error context determines different parameters for recovery actions

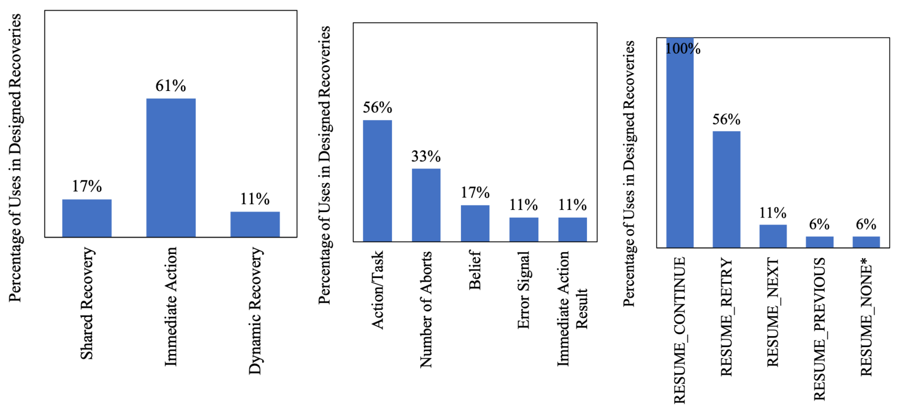
Figure 6a shows the occurrence of these three situations in our developed recoveries. We most frequently use *Immediate Actions*, to create short and responsive recovery actions to bring the task back to a known state. While less frequent, *Shared Recoveries* aided in rapid development and *Dynamic Recoveries* were crucial in creating a reactive system to deal with diverse faults.

The task executor and monitor use the following factors to determine what recovery to perform:

1. *Action/Task*: the location of the error in the task hierarchy
2. *Number of Aborts*: how many times the error has occurred without resolution
3. *Belief*: a subset of the robot's belief of the task, robot, or environment state
4. *Error Signal*: a specific error signal returned by a primitive action
5. *Immediate Action Result*: the result of actions taken for recovery execution

Figure 6b illustrates that localizing the error within the task hierarchy was especially important in determining recoveries because the task and action reuse for different situations often required different recoveries. Overall, the use of a diverse factors shows that robust recovery requires a wide range of context.

Finally, Figure 6c demonstrates that all resumption strategies described in Section 4.2 were necessary for designing a robust recovery system. The diversity in resumption strategies showcase the possibility of resuming from a task beyond simply re-attempting it entirely, and the use of resumption strategies other than `RESUME_CONTINUE` show that resumption cannot always directly return to where the error occurred.



(a) Properties of the recovery (b) Factors used in diagnosis (c) Resumption strategies

Fig. 6: Percentage of times that recovery uses each utility of the task monitor, for the 18 main recovery strategies designed for FetchIt! Challenge. In (c), note that `RESUME_NONE` is also the default strategy for unseen errors.

Representative Example We describe here recoveries for the *pick* action (Figure 5b) to provide concrete examples for the features mentioned above. In our system, *pick* and *arm* can fail due to a common cause—errors in the MoveIt! Motion Planning Framework. Therefore, the default recoveries for these actions, e.g. reinitializing a 3D obstacle map followed by a `RESUME_CONTINUE`, are examples of *Shared Recoveries*. However, depending on the context, the fault sometimes requires additional recovery steps. For instance, after the third consecutive failure of both actions in the *pick_task*, a short upward arm move jogs the system out of its error. When this is not enough, the cause of failures can be a limitation in the arm's workspace, and so the robot repositions itself based on beliefs about the task and environment state. All faults that occur within the context of the *perceive_pick* task retry that task (`RESUME_RETRY`) in order to account for scene changes resulting from recovery execution. We show a selection of these *pick* recoveries in action, as well as other example recovery behaviors selected from our FetchIt! competition runs, in the video supplement to this paper[15]. At the competition, we recovered from all errors in the *pick* task, thanks to the task monitor's utilities.

6 Discussion

We conclude with a discussion of lessons learned using our RDD-inspired task execution and monitoring system at the FetchIt! Challenge.

Testing early and often. The ability to repeatedly test the robot system in its target environment is a critical requirement for the robustness benefits of RDD–simulation alone will likely not lead to the same level of robustness. As such, RDD is not suited to hazardous or remote environments, such as space

[15] We also provide an HD version of the video here: https://youtu.be/AcOdT10q_94

robotics. However, many target environments for robots are neither inaccessible nor catastrophically hazardous, and are therefore compatible with RDD.

Stochasticity in behaviors. As mentioned in Section 4.2, failure recovery requires a high degree of variety in recovery mechanisms. We have found that a degree of non-determinism at the robot's behavior level facilitates such recoveries. For instance, our sampling-based ranking approaches to object selection, grasp calculation, and place pose calculation provided the robot with successful alternatives when retrying actions after previous action attempts failed.

Planning layer integration. Recipe-based tasks can admit multiple recipes, which need to be selected or rescheduled at runtime based on factors such as time constraints or major execution errors. Predefined scripts, such as those created during RDD specification phases, cannot easily handle such situations. The shortcoming can, however, be addressed by the higher level planning layer in the robot architecture. We found that the level of abstraction used in our hierarchical executive layer recipe scripts made the specification of our planning layer almost trivial. Additionally, the executor and monitor utilities we developed (e.g. beliefs) were a great help in the planning layer.

In conclusion, the RDD methodology of separating the nominal task specification from recovery specification provides numerous benefits, which our team validated at the FetchIt! Challenge. Our use of RDD (1) allowed rapid development of the task and recoveries, (2) enabled independent testing and efficient re-use through abstraction for tasks and recoveries, (3) necessitated the development of system utilities that ultimately proved valuable in other aspects of system development, and most importantly, (4) afforded our system a level of robustness that would have been more difficult or time-consuming to achieve through other means.

Acknowledgements

This work was supported by an Early Career Faculty grant from NASA's Space Technology Research Grants Program, NSF IIS 1564080, NSF GRFP DGE-1650044, and ONR N000141612835.

References

1. Akgun, B., Cakmak, M., Yoo, J.W., Thomaz, A.L.: Trajectories and keyframes for kinesthetic teaching: A human-robot interaction perspective. In: HRI, pp. 391–398. ACM (2012)
2. Atkeson, C.G., Babu, B.P.W., Banerjee, N., Berenson, D., Bove, C.P., Cui, X., DeDonato, M., Du, R., Feng, S., Franklin, P., Gennert, M., Graff, J.P., He, P., Jaeger, A., Kim, J., Knoedler, K., Li, L., Liu, C., Long, X., Padir, T., Polido, F., Tighe, G.G., Xinjilefu, X.: No falls, no resets: Reliable humanoid behavior in the DARPA robotics challenge. In: Humanoids, pp. 623–630. IEEE (2015)
3. Beetz, M., Mosenlechner, L., Tenorth, M.: CRAM x2014; A Cognitive Robot Abstract Machine for everyday manipulation in human environments. In: Int. Conf. on Intelligent Robots and Systems, pp. 1012–1017. IEEE (2010)
4. Berenz, V., Schaal, S.: The Playful Software Platform: Reactive Programming for Orchestrating Robotic Behavior. Robotics & Automation Magazine **25**(3), 49–60 (2018)

5. Bohren, J., Cousins, S.: The SMACH High-Level Executive [ROS News]. Robotics & Automation Magazine **17**(4), 18–20 (2010)
6. Chitta, S., Sucan, I., Cousins, S.: Moveit![ros topics]. Robotics & Automation Magazine **19**(1), 18–19 (2012)
7. Crestani, D., Godary-Dejean, K., Lapierre, L.: Enhancing fault tolerance of autonomous mobile robots. Robotics and Autonomous Systems **68**, 140–155 (2015)
8. Degroote, A., Lacroix, S.: ROAR: Resource oriented agent architecture for the autonomy of robots. In: ICRA, pp. 6090–6095. IEEE (2011)
9. DRC-Teams: What Happened at the DARPA Robotics Challenge? (2015). URL www.cs.cmu.edu/{~}cga/drc/events
10. Eppner, C., Höfer, S., Jonschkowski, R., Mart\'\in-Mart\'\in, R., Sieverling, A., Wall, V., Brock, O.: Lessons from the amazon picking challenge: Four aspects of building robotic systems. In: Robotics: Science and Systems (2016)
11. Guiochet, J., Machin, M., Waeselynck, H.: Safety-critical advanced robots: A survey. Robotics and Autonomous Systems **94**, 43–52 (2017)
12. Kent, D., Toris, R.: Adaptive autonomous grasp selection via pairwise ranking. In: IROS, pp. 2971–2976. IEEE (2018)
13. Klotzbücher, M., Soetens, P., Bruyninckx, H.: Orocos rtt-lua: an execution environment for building real-time robotic domain specific languages. In: Int. Workshop on Dynamic languages for Robotics and Sensors, vol. 8 (2010)
14. Kortenkamp, D., Simmons, R., Brugali, D.: Robotic Systems Architectures and Programming. In: Springer Handbook of Robotics, pp. 283–306 (2016)
15. Kress-Gazit, H., Fainekos, G., Pappas, G.: Temporal-Logic-Based Reactive Mission and Motion Planning. Transactions on Robotics **25**(6), 1370–1381 (2009)
16. Lemaignan, S., Hosseini, A., Dillenbourg, P.: PYROBOTS, a toolset for robot executive control. In: IROS, pp. 2848–2853. IEEE (2015)
17. ten Pas, A., Platt, R.: Using geometry to detect grasp poses in 3d point clouds. In: Robotics Research, pp. 307–324. Springer (2018)
18. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA workshop on open source software, vol. 3, p. 5. Kobe, Japan (2009)
19. Rusu, R.B., Cousins, S.: Point cloud library (pcl). In: ICRA, pp. 1–4 (2011)
20. Srinivasa, S.S., Berenson, D., Cakmak, M., Collet, A., Dogar, M.R., Dragan, A.D., Knepper, R.A., Niemueller, T., Strabala, K., Vande Weghe, M., Ziegler, J.: Herb 2.0: Lessons Learned From Developing a Mobile Manipulator for the Home. Proceedings of the IEEE **100**(8), 2410–2428 (2012)
21. Szafir, D., Mutlu, B., Fong, T.: Designing planning and control interfaces to support user collaboration with flying robots. Int. Journal of Robotics Research **36**(5-7), 514–542 (2017)
22. Wise, M., Ferguson, M., King, D., Diehr, E., Dymesich, D.: Fetch and freight: Standard platforms for service robot applications. In: Workshop on Autonomous Mobile Service Robots (2016)
23. Wohlkinger, W., Vincze, M.: Ensemble of shape functions for 3d object classification. In: Int. Conf. on Robotics and Biomimetics, pp. 2987–2992. IEEE (2011)
24. Zaman, S., Steinbauer, G., Maurer, J., Lepej, P., Uran, S.: An integrated model-based diagnosis and repair architecture for ROS-based robot systems. In: ICRA, pp. 482–489. IEEE (2013)