AN ABSTRACT OF THE THESIS OF

Keeley Abbott for the degree of Master of Science in Computer Science presented on December 14, 2017.

Title: Formative Work Toward a Mixed-Initiative Programming Language

Abstract approved: ____

Eric T. Walkingshaw

Mixed-initiative programming entails collaboration between a computer system, and a human to achieve some desired goal or set of goals. Often these goals change or are amended in real time during the course of program execution. As such, the plans these programs are based on must adapt and evolve to accommodate this iterative process. This thesis collects a literature review of research done in the field of mixed-initiative programs that provides an understanding of the problems faced when attempting to integrate computer systems and human users, a previously published paper with formative work in understanding how humans write programs for other humans, and finally some initial work done to develop an embedded domain-specific language for mixed-initiative drone control.

©Copyright by Keeley Abbott
December 14, 2017
All Rights Reserved

Formative Work Toward a Mixed-Initiative Programming Language

by

Keeley Abbott

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 14, 2017
Commencement June 2018

Master of Science thesis of Keeley Abbott presented on December 14, 2017.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Keeley Abbott, Author

ACKNOWLEDGEMENTS

First and foremost, I would like to recognize and thank my advisor, collaborator, and friend Eric Walkingshaw, whose constant encouragement and willingness to assist have formed the foundation of my work. I count myself fortunate that he was willing to take a risk on me as a student.

I also want to offer a heartfelt thanks to Martin Erwig, who took a chance on an unproven non-traditional student who showed up in his office with little more than an interest in Computer Science.

Last, but certainly not least, thank you to my best friend and partner and mother of our two fantastic children, Joanna. Thank you for believing in me when I didn't, pushing me when I wasn't sure, and reminding me of the immense opportunities that lay before me.

CONTRIBUTION OF AUTHORS

This section briefly describes our relative contribution to that paper in order to demonstrate that it satisfies the requirements for a manuscript format thesis. Also described in this section are contributions provided from additional written but unpublished papers contained within this manuscript which further substantiate manuscript requirement satisfaction.

The formative work presented in Chapter 2 was driven primarily by myself. Eric Walkingshaw provided essential critique and insight throughout the process of developing and framing the literature review. The development of the design concepts and the synthesis of concepts as they relate to previous work in mixed-initiative programming was done by myself.

Eric Walkingshaw and Christopher Bogart are co-authors of "Programs for People: What We Can Learn from Lab Protocols" [1] contained in Chapter 3. This formative work was led by me, but contains significant contributions by Eric and Christopher. Eric and Christopher both provided background knowledge in the area of related work and in the analysis techniques employed, as well as revisions of the written content of the paper.

The design work described in Chapter 4 on the mixed-initiative drone DSL was highly collaborative. This work was written and presented as the extended abstract, "Valkyrie: A DSL for Mixed-Initiative Drone Control" [2]. The work on the mixed-initiative drone DSL was originally done by me as a class project, and the extended abstract presented at IFL was written by Eric and myself. Alex Grasley provided some additional work on the back end coding for the proposed DSL subsequent to the submission. Revisions to the work done on the drone DSL paper were completed primarily by myself with additional input from Eric. The presentation of the work at IFL 2016 was done by myself.

[1]Abbott, Keeley and Bogart, Christopher and Walkingshaw, Eric. "Programs for People: What We Can Learn from Lab Protocols." *Visual Languages and Human-Centric Computing (VLHCC), 2015 IEEE Symposium on.* IEEE, 2015.

[2]Abbott, Keeley and Walkingshaw, Eric. "Valkyrie: A DSL for Mixed-Initiative Drone Control." Research abstract presented at *28th Symposium on Implementation and Application of Functional Languages (IFL), 2 September 2016, KU Leuven, Leuven, Belgium.*

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

Chapter 1: Introduction

Computer systems and human users have differing and often complementary sets of skills that pertain to problem solving. One potential way to harness these skills to the benefit of program execution is to create *mixed-initiative programs* in which both the computer system and human user are involved in the execution of programs. These mixed-initiative programs are designed to meet some goal or set of goals through the completion of a plan that can be easily modified during execution. Often the computer system includes the use of an *Artificial Intelligence (AI)* component that has its own set of motivations for completing the task or tasks given to it.

Success in mixed-initiative programming is punctuated by the ability to assist the programmer—as well as the computer system and human user—in taking advantage of these differing strengths. At the same time, mixed-initiative programs are responsible for mitigating the weaknesses that are presented by both types of participants.

A side effect of mixed-initiative programming is the relationship to human-computer symbiosis—owing to its natural affinity to the promotion of cooperation between humans and other intelligent systems [38]. In order to achieve the eventual outcome of symbiosis, there must be an extraordinarily close mapping between computer systems and human users that also needs to occur as natural—as opposed to overtly scripted—interactions between all systems concerned. The long-term goals of such closeness are: to (1) allow computer systems to assist humans in understanding formulative thinking in much the same way computers already solve formulative problems; (2) similarly to facilitate the understanding of natural language input and nonverbal communication by the computer system in much the same way humans communicate naturally with each other; as well as (3) to facilitate the flexible exchange of control when executing complex decision-based programs, without relying on programmers to provide rigid pre-conditions.

Chapter 2 contains a review of related work in the area of mixed-initiative programming. We examine work on collaborative planning systems that involve both AI and human components in the design and modification of program plans. In addition we review work done in mixed-initiative computing as it relates to multi-agent systems, and functional reactive programming (FRP) which is a well-known method for programming event-driven processes. We discuss some of the issues in the use of AI systems, and provide a framework for the application of these ideas of collaboration to a mixed-initiative programming language.

Specifically we outline the concerns presented in early work by Licklider [39], which are later re-examined by Lesh et al. [36] to understand where challenges still exist in achieving the closeness Licklider deems necessary for true human-computer symbiosis. These concerns can be viewed through the lens of work done in mixed-initiative planning for drone control [7], robotic systems [21], and planning assistants [19, 20].

While advances in software and hardware design have directly led to increases in symbiosis, some of the longstanding barriers to true symbiosis—the free exchange of initiative and goal-setting during execution—still exist [37]. Traditional programming languages instruct the computer system in a mixed-initiative system, but we still don't fully understand how we should instruct human users. To explore this problem, in Chapter 3 we develop strategies for implementing design decisions that address problems traditionally encountered when people write programs for other people [1]. These strategies can be adapted for use in systems where humans and computers must interact with one another.

Protocols are instruction sets written for people by some "expert" in a process or procedure that is of interest in some similar context. In a very simple way, they can be interpreted as programs to be executed by a human. The difference between these programs, and those written for computer systems, is the amount of flexibility inherent in such instruction sets. Writers of protocols understand that people executing them often see them as "guidelines" [43, 61] rather than as rigid step-by-step documents.

Protocols are often not explicit enough for just anyone to follow them. It takes experience to know how to carry out a protocol. Lynch notes that protocols often omit "substeps and contingent repairs", and describe activities using vague adverbs like "exactly", "gently", "approximately", that require experience to interpret. Timmermans and Berg [62] also found that it takes not just field-specialist knowledge, but actual previous experience with the specific protocol itself; for example nurses asked to follow a new experimental treatment protocol might not know if a fever at a particular stage of the protocol was expected or exceptional, and so not know whether to carry out the hospital's existing fever protocol in addition to the new protocol.

These attitudes toward the interpretation and execution of protocols can be seen in some common design practices we identified in [1]. Protocols are written in surprisingly *simple* and *rigid* terms, so the intent from the writer's perspective is to be more concrete than would be typically expected. Surprisingly, the concrete nature of these protocols, actually makes them more easily adaptable for specific situations. Because the low-level instructions for protocol users are handled by the protocol writer, the user is freed to focus efforts on more complex tasks like error handling and instruction set modifications based on the execution environment. These higher-level functions often require domain expertise.

From the perspective of human users of protocols this freedom from low-level reasoning allows them the option to *interleave* or *integrate* their own plans. These plans could include additional instruction sets, or other procedures developed separately from the main protocol, but having some bearing on the situation or process. We learned that much of this behavior is expected—or at least anticipated—and protocol writers write intermediary checks to ensure protocol users have a means to remain on course throughout execution.

To maintain a simple interface, protocol writers employ a very linear model. This may play a role in the acceptance, and modification of protocols by users. The simplified "path" view of protocols allows users to parse the document in a large chunk, as well as anticipate future consequences for making modifications or deviating from the protocol as written.

Ultimately we define the design elements necessary for the formulation of a mixed-initiative programming language in Chapter 4. The language must support: (1) *negotiation* of initiative control; (2) *adaptable autonomy* in the execution of tasks; and (3) *natural communication* between the computer system and human user. In order to accomplish this, we use the design implications gleaned from the work in Chapter 3. We maintain the simplified "path" view familiar to human users, and in doing so provide ridged instruction sets that can easily be adapted.

The ideas represented by this initial work on a mixed-initiative programming language are applied to a real-world problem in Chapter 4 as well. We propose the steps necessary to create a mixed-initiative programming language for drone control. The advantage of a domain-specific language (DSL) like this lies in its ability to minimize the "planned" contingencies, and instead rely on the human user to provide troubleshooting during execution. This allows the programmer to exploit the strengths of the human users, while minimizing their weaknesses by allowing the computer system to perform the mundane or repetitive tasks for which they are best suited.

Chapter 2: Literature Review of Human-Computer Collaboration

2.1 Introduction

As early as 1960 Licklider [39] acknowledged the importance of delegating tasks to both computer systems and humans, as well as the management of their interaction. These concepts have been a point of consideration in the creation of languages, libraries, and design patterns. The emphasis placed on how to meet these needs, as well as the ultimate goal in meeting them has paved many divergent paths leading to a broad assortment of fields in which similar work is being done. Each of these fields attempts to address the common concerns in their own way, representing the solutions as varying degrees of human and/or computer system control.

Although the variety of approaches across fields does not represent a major concern, it does highlight the issue that while much has been done to address the primary aims of Licklider [39] such as: (1) allowing computers to assist users in formulative thinking as much as they already provide solutions in this format; and to (2) enable a collaborative framework on which computers and users can each contribute to decision-making and control processes. For Licklider the intent was a balance between the extreme view of extending the human [47], and that of complete management by an artificially intelligent system.

Lesh et al. [36] examined these aims 40 years after their introduction, and while some portions of each have been addressed, true symbiosis has yet to be realized. Echoing Lickilder, they suggest that there are three major components needed to achieve true symbiotic interaction: (1) the segregation of duties between the computer system and the human user need to be complementary and effective; (2) computer systems require a concrete representation of the human user—the domain knowledge they represent; and (3) the inclusion of forms of nonverbal communication modalities—like gesture-based communications.

Of particular importance when discussing the development of a mixed-initiative programming language are the need to create effective and complementary divisions of computer system and human duties, which requires the inclusion of an accurate rendition of the human user's intentions, ideals, and beliefs. By extension this entails an efficient method for reconciling changes to goals and plans made during execution.

As a way of examining these concerns from a broader perspective we have been working on ways to model interactions not as systems which delegate control to the human or computer agent dominantly, but rather as a spectrum with varying degrees of control throughout execution. Our work on understanding human programs [1] has led to some insights about the ways in which humans interact with one another through programs that have led to an understanding of how interactions might be represented more satisfactorily from the human's standpoint—thereby creating a more natural interaction.

While developing more natural interactions between human and computer systems is a fundamental piece of our research, a base knowledge of efforts that have previously been made and the gaps still present in achieving true human-computer symbiosis is a crucial component. This thesis attempts to catalog and analyze such efforts in order to explicate the various goals of such attempts into a single thread of work focused on the unification of human and computer system tasks into negotiated programs capable of adapting to a variety of situations and environments.

In the remainder of this chapter we present work in the areas of concern for the design of a mixed-initiative programming language (Sections 2.2—2.5). In addition, we examine work using *Artificial Intelligence (AI)* in designing collaborative plans with human users (Section 2.6). We also explore a logical framework that could be leveraged to develop a semantic basis for a mixed-initiative programming language (Section 2.7). Finally, we synthesize design concepts for a mixed-initiative programming language from the information collected in Section 2.2—2.5 (Section 2.8), and provide a path for the implementation of these design concepts (Section 2.9).

2.2 Symbiosis

The main thrust of the work done by Lesh et al. is to recognize areas in which symbiosis is still evasive. They suggest much of the work in human-computer interaction focuses on providing more efficient forms of interaction (i.e. text-to-speech, novel display technologies, verbal command input, etc.), as opposed to changes in human-computer interaction styles. As such, when designing systems adept at optimization problems the approach has been to fully automate rather than providing segregated and complementary duties to both the computer system and human.

Nonverbal communication is often intertwined with verbal communication [44] when humans communicate with one another. Gestures in particular indicate much about the person making the gestures, such as their intentions or beliefs. These unspoken indications of our meaning can be as subtle as facial muscle movements or as explicit as large hand gestures. In particular, hand gestures are a way of augmenting verbal communication in a tactile way.

In the time between the work of Licklider and that of Lesh et al. there has been an abundance of work done to accumulate an understanding of and theory about the ways humans work together. `COLLAGEN` [55, 56] applies this understanding of human communication and collaboration—particularly work on SharedPlan theory [25, 26, 42]—to human-computer interaction. The collaboration represented by `COLLAGEN` essentially extends what might commonly be called a "dialogue manager" with a representation of goals and plans developed by both the computer system and the human.

One of the tenets of Lickilder's work suggests that in order to achieve symbiosis we must be able to assist users in formulative thinking. In a very general way attempts to allow computer systems to assist humans in understanding formulative thinking entail the inclusion of human input in setting goals, generating hypotheses about outcomes, determining operating criteria, and analyzing outcomes. Computer systems are then freed to complete the tasks they are best suited for—namely performing routine work and mundane calculations—in order to support the human in focusing on the insight and decision-based operations for which they are better suited. However, unlike systems in which the computers are merely extensions of the human, symbiosis requires a level of initiative sharing that keeps the computer agent at the forefront of the decision-making process.

The effect of initiative sharing can be thought of in terms of a similar paradigm, *pair-programming* [11], where programmers complete programming tasks collaboratively. There are ways in which both can benefit from initiative sharing—as is true in traditional pair-programming where both participants are humans—where each participant provides its own set of skills and specialized knowledge. The goal of pair-programming draws parallels to initiative sharing in that it combines the strengths of each participant involved in the decision-making processes by allowing them to interject goals and criteria where they have insight not originally considered by the programmer.

These specialized skills and knowledge can be particularly useful when execution must occur in real time, and systems are required to adapt to updated information and changes in the environment as execution occurs. For example, in the operation of *unmanned aerial vehicles (UAVs)*, where unexpected flight conditions or changes in flight goals can occur long after a program has been written for the vehicle to execute. Maintaining human involvement in these cases can be beneficial, where the decision-making and problem-solving abilities of humans can be fully utilized by allowing them to participate in the problem-solving process in things like error handling where they excel.

2.3 Mixed-Initiative Planning

The explicit interleaving of human and computer tasks through hardwiring (e.g. [35]), allowing machine learning searches to be interactively shaped by human users through the addition of constraints (e.g. [32,36]), and discourse-based techniques (e.g. [45,59]) are all mechanisms for trading initiative that have been explored in the past. These design strategies have all been employed to implement or improve mixed-initiative programs.

Mixed-initiative planning systems provide an avenue for designing and developing systems in which both computer systems and humans collaborate to develop execution plans. In such systems the "initiative" can be shared between the computer and human to conceive of, produce, and make changes in order to manage plans as they unfold. This allows for plan development that takes place naturally, where neither of the participants are "asked" explicitly to take initiative. Intuitively, the ultimate goal of these types of systems is to develop interaction techniques where both computer and human can influence the course of actions and adjust goals in a computational environment where activity planning is ongoing throughout state changes.

Research around mixed-initiative planning focuses on exploring various ways to produce reliable synthesis of the strengths of both computer systems and humans in order to construct reliable and flexible plans. Humans require intelligent and active problem solving support, and computer systems including AI planning systems need support from human counterparts to define problem scope, in some information interpretation tasks, and any computations involving spatial or perspective reasoning (particularly in real-world applications like the UAV example above).

From a pragmatic standpoint, it is important to frame an understanding of how to achieve the optimum information sharing strategy as it applies to both computer systems and humans. To approach this we need to understand: (1) how to ensure computer systems and humans each focus on the areas of computation they are best suited for; (2) how to facilitate different styles of communication between computer systems and humans where each represents a different set of computational strengths or functionality; and (3) ensure that both computer systems and humans have the ability to transfer initiative authority in all planning related tasks.

2.4 Mixed-Initiative Computing

Researchers have called for more flexible flows of control between people and computers [39] as early as 1960. The explicit interleaving of human and computer tasks through hardwiring (e.g. [35]), allowing machine learning searches to be interactively shaped by human users through the addition of constraints (e.g. [36], [32]), and discourse-based techniques (e.g. [59], [45]) are all mechanisms for trading initiative that have been explored in the past. This prior work has focused on implementing or improving mixed-initiative programs, however our work focuses on applying language features behind programs written for people to mixed-initiative programming.

Human agents have been used to identify visual patterns as well as in aerial search activities in the form of crowd-sourcing applications [53]. Often these tasks come in the form of games, or provide some monetary compensation for the user's time in classifying or searching images for data. Frameworks for distributing these tasks to the crowd have also been developed to assist programmers in developing their queries and honing them through successive collections of data obtained from the crowd [41].

One difficulty that is often encountered when using crowd-sourcing for the purposes of surveying or search and rescue, is that the queries often take a long time to return results. Which can degrade the chances of a successful outcome, and can result in the use of outdated information when reacting to data collected from participants. To some extent this can be mitigated by the use of increased monetary incentives to the crowd participants. In addition to timing issues, crowd-sourcing applications often treat humans as monetized data processing labor, and fail to take advantage of the crowd's facilities beyond visual pattern matching.

In a mixed-initiative DSL, the goal would be to allow the human user to directly interact with and manipulate the flow of control and the expected outcomes in real time. This provides the benefit of making new information gathered by the human user's visual pattern matching skills more readily available in addition to allowing us to take advantage of the human user's skills and knowledge that may supersede that of the original goals or written program.

Another area of research with potential applicability to the design and implementation of a mixed-initiative DSL is *agent communication language (ACL)* [46,48] or *multi-agent systems (MAS)* [22]. In such systems, a complicated network of channels is used to coordinate the work of multiple autonomous vehicles in the completion of a task or tasks. The reason there is applicability to a mixed-initiative DSL is that in a way we are dealing with multiple intelligent agents "competing" for control in order to complete shared goals.

However, unlike most of the systems that employ these frameworks, a mixed-initiative DSL needs to have a more simply defined communication network. We cannot expect a human user to receive, process, and respond to system-generated messages in the same way an autonomous intelligent system would. Messaging systems need to accommodate messages that can be quickly and easily interpreted by the human. The principles presented by ACL and MAS are helpful in formulating ways to prevent deadlock when multiple agents are attempting to control the flow of the program at the same time. For the purposes of a mixed-initiative DLS though, much less complex forms of these controls are required owing to the inclusion of a human user.

2.5 Functional Reactive Programming (FRP)

Applications of FRP like Frob [50] are also instructive in implementing mixed-initiative programs in the area of controlling autonomous vehicles by providing a basis for understanding the implementation of event-driven programatic reactions. However, unlike Frob, a mixed-initiative DSL would not be concerned strictly with the execution of code from the standpoint of the computer system. Having to take a human user into consideration adds an additional perspective and necessitates some fundamental differences in the way a mixed-initiative DSL implementation utilizes the FRP framework. In order to accomplish the integration of both types of input, it would have to concurrently monitor behaviors and events in constant time as they relate to the program input, as well as manual key inputs from the user.

The fusion of these two types of inputs could result in errors from a deadlock in the system if control is not adequately delineated at the time a negotiation command is received by either participant. In other words, any FRP that would meet the purposes of a mixed-initiative DSL implementation must remain decidable, but at the same time allow for an interpretation of the plan that may be unanticipated by the programmer. Which makes it difficult to directly implement the core intent of such a DSL in a traditional way even when using an event-driven programming language library like FRP.

2.6 Collaboration in Planning

In the work in the area of collaborative planning, much of the focus is placed on designing "stand-alone" planning systems, where collaboration of efforts and resolution of differing communication styles are not concerns. To achieve the necessary closeness of computer system and human it becomes incumbent upon developers to place these issues at the center of design considerations. In order to develop this further, consider a system in which multiple "actors" of both human and computer system origins are actively cooperating with one another. From that perspective we will examine aspects of current planning systems theory and its practice to define potential changes.

From an AI perspective, much of the current understanding relies on the search through and negotiation of spaces of partial plans to arrive at some acceptable work plan [69]. Classically, this involves refining goals incrementally using a back-tracking search algorithm. The issue in using this view to represent collaboration is in the search algorithm, where the introduction of multiple "actors" undermines the reliability of such methods. This is because the fundamental understanding of this process requires the systematic exploration of programatically controlled refinement alternatives based on the plan. This results in a search for a "satisfactory" plan, not necessarily and optimal plan. We know from [1] that users don't behave in such a systematic way of approaching plans, and often decisions about what to explore are based on personal notions of the information provided or some previous experience that relates. Where plans are concerned, if it is a particularly important objective a human may explore many different avenues leading to similar (or the same) result before they implement a more detailed version of the plan.

Therefore, in a mixed-initiative approach, we have to take into account not only computer systems that have been given authority to search the problem space, but also humans that may want to refine or change the search area or define the search breadth in some other way. This may result in some necessary (and ultimately essential) redundancy of efforts, because to the extent where the search results and parameters overlap between computer systems and humans they can each reinforce and refine each others collaborative efforts.

Another necessary argument in the discussion of mixed-initiative planning is the need to support differing communication styles during planning in the form of dialogues between computer systems and humans. Such communication has been accomplished using *multimodal interaction* [4–6], as in [7].

Multimodal interface development is at the crux of several areas of research that include vision, psychology, and AI. The thrust of multimodal interface research is to create computer technology that is more usable by humans. To be successful in designing such systems requires knowledge of the human interacting with the system, and the computer systems—including AI agents—that are being interacted with. Collaboration is supported by having an understanding of the interactions that take place during the course of execution.

In [7], multimodal interaction is accomplished by employing several overlapping avenues for operators to interface with planning—including speech, gesture, joypad, and tablet interfaces. The intent of providing diverse modalities for communication is to foster pliable communication that won't break easily, and feels less artificial to the human. These methods of control are of course tailored to the problem space of controlling multiple UAVs in executing a search and rescue operation.

Managing multimodal inputs and coalescing them into a single stream requires the implementation of a framework like the late fusion approach used in [57]. This framework integrates information from individual modalities once the input has been deciphered by an interpreter. The strength of this approach lies in the focus on the beneficial attributes of each of the modalities being introduced into the analysis. That being said, the expense comes in the form of learning effort required for this type of multimodal fusion, where each of the modalities is provided a "learner" to score the data provided in addition to the final analysis and scoring of the collated data [60].

2.7 Semantics of Negotiation

The logical framework presented by `KARO` [63] is a potential semantic basis for a mixed-initiative programming language that supports active exchange of initiative between computer systems and human users. `KARO` has been used in other mixed-initiative applications like [13, 64] where multi-agent systems are concerned. `KARO` is a framework for logic that combines dynamic and epsitemic/doxastic logic including modal operators to deal with computer system or human user motivation. In `KARO` these motivational modalities include, *beliefs*, *abilities*, *desires*, and *commitment*.

- `belief`: expressing an agent's belief about some state that leads to an action.
- `able`: expressing that an agent has the capability to perform an action or determine some result.
- `desire`: expressing that an agent has a desire for some given state.
- `commit`: expressing an agents commitment to an action.

In this way, we can think of the meaning behind programs as the act of two agents resolving the course of action through the negotiation of their motivational desires in order to arrive at a `goal`. This `goal` is accomplished through the execution of `plans`—or a series of `tasks`. Alternatively we can think of this as a `plan` and the `goal` to be achieved in executing that `plan`. The resolution achieved can be roughly approximated by the idea that agents have some *beliefs*, *desires*, and *intentions* in reaching the stated `goal` [10, 18]. From a social perspective, this is the idea that agents are entering into a sort of "social contract" where each of their decisions are considered in order to effectively achieve the goal.

2.8 Design Concepts and Plans

From the literature we have identified three fundamental interdependent conceptual issues in the creation of a mixed-initiative programming language that takes advantage of both the computer system and human user capabilities: flexible yet well defined *negotiation* of initiative, *adaptable autonomy* for the computer system, and *natural communication.* This triad of conceptual issues can be used as a foundation for developing a well-defined and principled set of design concepts for mixed-initiative interaction that can be used to clarify, validate and verify different kinds of interaction between computer systems and human users. The concept of initiative handling is particularly important to our framework, providing a bridge between natural communication and adaptable autonomy.

Pragmatically and conceptually there is an issue central to these three concepts along with their theoretical and pragmatic integration—that of plan representations and semantics in practical application. Any representation of a plan needs to be *flexible* and *dynamic*. Each plan representation needs to be able to be delegated at differing levels of abstraction and also must be able to be expanded and/or modified as portions of plans are recursively delegated between the computer system and the human user. This plan structure is a representation of compromise between a wholesale compiled plan and a plan generated by a self-contained planning system [33].

Because a plan is essentially a linear model of the program that embodies the "ideal" execution of a series of actions. This is important, because it supports the design insights provided by [1] regarding how humans use and relate to protocols. Although there are some uses of `looping` and `branching` within human written protocols, for the most part they are flattened programs. They represent the intent of the protocol writer, and provide markers to indicate important or necessary steps along the way, but leave a great deal of the ordering of steps and error recovery up to the human executing them.

A plan consists of multiple kinds of tasks. A program task implements a procedure for execution by the computer system, and also provides a brief explanation of the task for the human user. These tasks represent an atomic unit of control the human has over re-ordering and executing aspects of the computer plan.

Other kinds of tasks include guideposts (e.g. expected outcomes) that support error detection. Guideposts also help the human user recover from errors by providing information about where and how to successfully return the system to the original protocol.

The structure of these tasks into plans provides the human user with freedom to execute the individual components of the plan in an arbitrary order (provided there are no existing unmet requirements from the standpoint of the block being called). Similarly it supports the concept of adjustable autonomy. It also frees the program writer from having to conceive of necessarily complex and obfuscatory error handling procedures, but rather relegates this to the discretion of the human agent—who is naturally better at reacting to unforeseen events compared to a computer system.

2.9 Conclusions and Future Work

We have described the challenges presented in using AI systems for collaborating to generate and manage plans. We have also described a set of concepts and a logical semantic basis that could be used to create a mixed-initiative programming language that will allow for the inclusion of both computer system and human user initiative in negotiating plan execution. Ultimately it is our intent to use this framework to design and implement a DSL that will allow for a real-world application of these concepts.

One of the primary applications for such a language would be in controlling drones—or swarms of drones—for search and rescue or surveying operations, where contingencies for changes in operating environment or parameters would require plan modification during execution. Such contingencies cannot necessarily be completely enumerated or anticipated by the programmer, and reliance on the human user's ability to adapt to and solve such problems is a beneficial effect of our language design concepts. The ability to rely on the collaboration between the computer system and the human user not only relieves the programmer of such thinking—and creating the error handling mechanisms behind them—but also allows the human to contribute to the plan in more meaningful ways.

In addition the lack of need for error handling frees the programmer to focus on the more necessary aspects of programming the plans—as well as the subgoals/tasks involved—and opens mixed-initiative programming to interactions that go beyond simple give and take by the computer system and human user. Programmers still have the option of providing the human user relevant "check points" in the style of protocols, as well as creating a "path" for both parties to follow. At the same time, these design concepts are flexible in their execution.

By exposing the programmer to a small set of negotiation primitives that allow the exchange of control between the computer system and the human user, the flow of the program can be synchronous or asynchronous depending on the context and can adapt during execution. This simplified view also translates to the human user view, where the human user has access to an explicit interpretation of the plan as a whole that can have additional plans or tasks interleaved with or inserted into pre-existing plans. This high-level view also makes the plans easily understandable and negotiable by the human user.

Chapter 3: Protocols: Programs for People

3.1 Introduction

A growing number of end-user programming scenarios such as workflows and interactive notebooks serve the dual purpose of being how-to documents for humans to learn from, as well as mixed-initiative execution and programming environments for the accomplishment of real tasks. Minimalist learning theory [9] labels these situations "active learning": learning skills in the context of one's work. Although research on the cognitive aspects of active learning continues, less is known about what *programming language features* are needed to support this interplay between: (1) the designer-specified sequence of activities and explanatory examples, and (2) the user-specified sequences that emerge as the user transitions from learning to doing.

This interplay is a form of *mixed-initiative* execution [28] in which human and computer activities are interleaved. But unlike the carefully scripted interaction of, say, a wizard, in these active learning tools it is not always clear which party, computer or human, is leading the way. How can the same program not only communicate a fixed example but also continue to help the learner understand the hows and whys of generalizing that example to some real application? How can a mixed-initiative programming environment be designed to allow the user to take the reigns when necessary, without altogether abandoning the computer's guidance?

Like workflows and interactive notebooks, biological laboratory protocols also have this same dual purpose: they teach a process by example, while also serving as a template for applying the process to a real-world goal. However, there are many differences in how protocols are *executed* compared to computer programs. Correspondingly, there are also many differences in how protocols are *written* compared to computer programs. Much of the existing work with protocols has been sociological work focusing on their role in human activity. In this thesis, however, we focus on the distinctive language features necessary to support humans executing protocols, as contrasted with computers executing programs.

Understanding protocols can lead us to discover new models for creating or improving current designs for mixed-initiative programs. If the language features found in protocols can be translated into design insights for mixed-initiative execution (Section 3.1.1), they could potentially be integrated into existing programs for active learning, where human participants already play an active role in execution.

Furthermore, the design insights extracted from the language features found in protocols can inform the design of a mixed-initiative domain-specific language. This could be used to build a new class of flexible mixed-initiative execution programs, in which control of process and workflow would not be "dominated" by either the human or machine, but rather shifted along a continuum of control. This could allow both human and computer to work in concert to achieve a goal, using the best of each of their abilities.

3.1.1 Mixed-initiative execution

There are many programs that involve a mix of work between human and computer actors that can be construed as part of the broader spectrum of mixed-initiativity: distributed human computation [54] in which a computer organizes and integrates feedback from many people; wizards and interactive forms in which a human supplies information in between processing steps; computational notebooks like IPython [49] in which computers fill in intermediate results as humans edit programs; and the familiar interactive computing scenario where subprograms are offered to users as tools to use freely in the course of the their work.

In all these cases, however, either the computer or the human maintains primary control of the process. In *computer-led* mixed-initiative programs the wizard or form mostly forces the user along a particular path, and the distributed computing paradigm parcels out small tasks to humans at its own convenience. Conversely, in *human-led* mixed-initiative programs, the interactive tool paradigm puts the computer's services completely at the user's disposal.

However there are tasks for which a more flexible exchange of initiative would be useful; none of these paradigms suit such situations. Computer-led mixed-initiative programs, for example, do not easily allow the human actor to *interleave* or *integrate* their own plans, goals, and constraints with the actions the computer needs them to perform. They also do not take advantage of resources or capabilities the human has beyond the program designer's expected audience. This might make the decision to submit themselves to the computer's guidance an unnecessarily significant time investment, both in carrying out the human tasks the computer might assign, and in tailoring the results the combined task ends up producing.

Conversely, human-led mixed-initiative programs may not provide enough guidance for a user without the experience to carry out the task at the macro level. For example, de Souza and Leitão [14] describe the plight of a Microsoft Word user trying to make page numbers appear consistently in a document where some pages are oriented vertically and some horizontally. In their example, the computer provided the pieces of functionality needed to complete the task, but did not succeed in communicating to the human which pieces and in what sequence the pieces should be applied.

3.1.2 Research questions and contributions

Although there has been research on the flexible relationship between human action and human plans, less is known at a granular level about the features and properties of written protocols that accommodate such flexibility. Thus the high-level goal of this research is to understand how people write detailed, reusable processes (that is, programs) to be carried out (executed) by other people, in a way that manages to convey both constraints and degrees of freedom for the protocol user. We look specifically at *biological protocols*, which are used to describe step-by-step how to accomplish tasks in a lab.

Protocols are written at different levels of formality for different purposes. At the least formal extreme, a scientist might create a protocol in their own lab notebook for personal use as a way to recall the process between iterations or when writing up results. Slightly more formal are protocols developed for specific internal or institutional use, which may address safety or quality standards specific to the organization, or may be employed to ensure consistency between actors. At the most formal extreme, there exist fully elaborated and highly refined protocols used to communicate novel techniques to other scientists around the world.

We focus on the formal extreme in this study. Specifically, we analyze peer-reviewed protocols published in *Cold Spring Harbor Protocols*. Unlike personal or internal protocols, these represent programs written for people unknown to the author(s) of the program. This aligns with our long-term goal of developing a language for mixed-initiative programming. Additionally, the protocols are more consistently formatted, and similar protocols contain similar visual and organization features, which makes them easier to analyze.

We have the following research questions:

RQ1. How are individual instructions expressed, and what is the granularity of an instruction? What balance of constraint and freedom do they allow, and how do they accomplish this?

RQ2. In what ways do protocol writers control and manage the factors that contribute to imprecise execution of protocols? Are there any distinguishing features that highlight the importance of steps *within* the protocol, and when there can be variance, how do protocol authors clue users in to this?

RQ3. How are relationships *among* protocols expressed in ways that constrain or allow flexibility in composing them? Is there any inter-protocol structure, and if so, what does it look like? Are large protocols separated into multiple smaller protocols? Do protocols refer to external protocols?

In answering these questions, we make these contributions:

C1. A protocol coding scheme, summarized in Tables 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, and 3.7.

C2. Descriptions and examples of how protocols specify instructions in terms of simple actions, goals, tasks, and open-ended tasks (RQ1; Subsections 3.4.2,3.4.4).

C3. Characterizations of constraints and allowable variability within instructions (RQ2; Subsection 3.4.1).

C4. Characterizations of control flow, error handling, and references to external protocols are specified in a lightweight fashion that relies on human adaptability and the flexibility of the instructions (RQ2 & 3; Subsections 3.4.3).

C5. Design implications for workflow tools (Section 3.6).

3.2 Related Work

As early as 1960 [38], researchers have noticed the differences between human instructions and computer programs, and called for more flexible flows of control between people and computers. Mechanisms for trading off initiative have been explored, for example discourse-based techniques (e.g. [59], [45]), allowing humans to shape machine learning searches by interactively adding constraints (e.g. [37], [32]), or simply explicitly hard-wiring the interleaving of human and computer tasks (e.g. [35]). That prior work has been more about how to implement or improve mixed-initiative programs; in this thesis we are investigating what role the written language behind mixed-initiative programs needs to play.

While programs and protocols seem superficially similar, the *pragmatics*[1] around their use are quite different. Researchers have noted that people use plans [61] and protocols [43] as resources, not as rigid instructions to be followed mechanically. Timmermans and Berg [62] argue that:

> *"subordination and rearticulation of the protocol to meet the primary goals of the actors involved is a sine qua non for the functioning of the protocol in the first place. [...] Tinkering is a prerequisite for the protocols functioning."*

Their claim is that people simply would refuse to adopt an overly inflexible protocol, because it would be so unlikely to precisely fit their goals, expertise, and resources.

Suchman [61] argues that human plans are more like descriptions or predictions than prescriptions of what actions a person will take given the most likely sequence of future actions; they are resources for anticipating likely future events. In programming language terms, perhaps, they are more declarative than imperative in that people draw on knowledge of the whole plan rather than blindly following each step.

Different actors adapt protocols to their own purposes. Timmermans and Berg emphasize that actors following a protocol have their own "trajectories", that is their own goals and priorities; they stray from the protocol to various extents, in various ways, for a variety of reasons. For example, a cancer patient may be careful to take medications

[1]Horn and Ward [68] define pragmatics, a subfield of linguistics, as "the study of those context-dependent aspects of meaning a which are systematically abstracted away from the construction of content or logical form." Pragmatics explains how context can turn "it's getting late" from a literal observation about time to a hint for guests to leave after a party.

prescribed in a research protocol, but might skip some measurement steps, since their priority is getting better, not furthering the research. Timmermans and Berg also cite the example of emergency personnel choosing to follow the CPR protocol even when the patient is not likely to be saved, using the time and visible activity as a way to soften the blow to a family coping with a sudden death. Lynch [43] describes protocols as being "situated in other procedures": they are not executed in a vacuum, but are meant to apply in situations where appropriate actors, goals, materials, and values are already present.

Protocols simplify coordination because, they "contain explicit criteria on whether, when and how next steps are to be taken. Personnel delegate some of their coordinating tasks to it, and the protocol appoints specific tasks to them." [62]. Sidner [59] et al. have shown one way this simplification can have practical benefit in human-computer interaction: knowledge about task state can help a computer guess the correct interpretation of an otherwise ambiguous human action.

Some existing software tools play a similar role to protocols as tailorable process descriptions. For example the Bioconductor project [23] uses literate [31] R programs, called "workflows" and "vignettes", to describe how various packages can be marshalled to perform tasks. In these programs the data file and task are merely illustrative examples; the program does not differ *semantically* from any other R program, but it is meant *pragmatically* to be used primarily as a resource for copy/paste creation of a new program to do a similar task.

Process description tools like CoScripter [40], and Automator [2] provide platforms for designing and annotating *scripts* for repetitive tasks. In the case of CoScripter, these scripts are limited to parameterized web-based tasks, where the parameters are instantiated from a database of local values. The "You" construct in CoScripter presents an opportunity for mixed-initiative execution by pausing the script and providing the user an opportunity to respond to a prompt [8]. Like CoScripter, Automator provides similar functionality for turning control over to the user, but these are more structured than CoScripter's open-ended "You" construct.

3.3 Methodology

We (me and the co-authors of the paper this chapter is based on [1])performed a qualitative analysis of lab protocols using open-ended coding. In this section, we describe the coding scheme we used and discuss how it was developed and applied. We present the coding scheme in two parts. In the first, we describe the coding scheme applied to each *step* of a protocol. These codes provided the data needed to address RQ1 and RQ2. In the second, we describe the coding scheme applied to each *external reference* in a protocol. These codes provided the data needed to address RQ3.

3.3.1 Coding scheme for protocol steps

To start, the researchers reviewed a set of 2–3 protocols and independently developed a coding scheme to categorize each protocol step and to note features in each step relevant to our research questions. By comparing and discussing our individual efforts, and by working through each of the annotated protocols together, we developed a single coding scheme by consensus to use throughout the protocol analysis.

Within this scheme, each step of the protocol is assigned (1) a *kind*, which describes the primary action of the step (e.g. physical, measurement, monitoring); (2) a *precision*, which describes how the step is stated (e.g. as an instruction, as a goal); and (3) zero or more *additional features* relevant to our research questions (e.g. gives additional advice, describes contingencies). The complete coding scheme is described in Tables 3.1, 3.2, and 3.3. For each kind, precision, and feature, we provide an example from the protocols assigned that code. We reference protocols using the notation [ID; Step], where *ID* is the protocol ID number, and *Step* is the step or section number.[2]

Next, two of the researchers coded the same protocol using the established coding scheme to establish agreement in the use of the scheme. We used Cohen's Kappa [34] to measure the agreement on each of the three components (kind, precision, features) of the coding scheme. We chose Cohen's Kappa as a conservative measure that corrects for coincidental agreements since we checked agreement on a small sample. The results of the analysis are listed below.

- *Kind*: 52 coding assignments, of which 50 were identically coded by both researchers (Kappa=0.868).
- *Precision*: 52 coding assignments, of which 47 were identically coded by both re-

[2]To retrieve a protocol, click its ID number in the PDF version of the thesis, or append the ID number to the url "`http://dx.doi.org/10.1101/pdb.prot`".

Kind	Example
`physical` *Manipulate physical objects*	"Slowly pour 14 mL of lysis buffer ..." [5384; 3]
`virtual` *Manipulate virtual objects*	"Using NimbleScan software, open the scanned image ..." [5385; 56]
`cognitive` *Evaluate or analyze*	"... check the quality of the amplification." [4974; 12]
`measurement` *Take a measurement*	"Quantify the DNA yield ..." [4974; 14]
`non-human` *No required human action*	"This method was adapted from an original protocol ..." [4918; pg. 3]
`selection` *Make a selection decision*	"Select a gene/genomic locus ..." [5491; 1]
`monitoring` *Watch or monitor*	"... monitor wet slides ..." [4706; 12]
`branching` *Perform parallel actions*	"This step can be performed during ... (Step 12)." [5237; 17]
`looping` *Repeat a process or step*	"Repeat Step 10." [5385; 12]

Table 3.1: *Kind* of action described in the step.

Precision	Example
`instruction` *Straightforward instruction*	"Anesthetize the animal by injecting intraperitoneally with 16 µL ketamine/xylazine per gram of body weight." [5410; 1]
`goal-directed` *Describes goal, not action*	"Reduce the volume to a 15-µL concentrate using ..." [4974; 20]
`task-directed` *Describes task, not goal/action*	"Examine the ruffled morphology characteristic of mats by ..." [085076; 3.ii]
`open-ended` *Requires creativity or ingenuity*	"Determine the amount of MNase to use for each cell type experimentally." [5237; 23]

Table 3.2: *Precision* of the description of the step.

Features	Example
`advice` *Suggested method or process*	"For best results . . . " [4918; 8]
`constraints` *Avoid this method or process*	"Do not use syringe needles to load the pipettes . . . " [5201; 1]
`expected outcomes` *Expected result or outcome*	"This provides a final effective dilution of 10–6 viable cells per milliliter . . . " [5492; 7]
`optionality` *Optional process*	"(Optional) Remove cells for additional analysis." [085076; 4.iii]
`wiggle room` *Alternative method or process*	"This mixture can be stored for up to 6 mo at -20°C." [4974; 8]
`contingencies` *In-line troubleshooting*	"If the density of cells and chromosome spreads is insufficient, pellet the cells, resuspend in a smaller volume of fixative, and repeat with fresh drops." [4706; 8]
`information` *Background information*	"The PCR product is denatured and the nonbiotinylated strand is removed in this step." [5491; 37]
`reference` *Outside source*	". . . see Preparation of Fixed Xenopus Embryos for Confocal Imaging (Wallingford 2010b)." [5427; 2.i]
`mapping` *Map instruction over a set*	"Treat all reserved aliquots with 20 µg of proteinase K for 30 min at 65°C." [084848; 33]
`visual pattern match` *Human visual processing*	"View whole-cell K+ currents and record for further analysis (Fig. 3)." [5014; 35]

Table 3.3: Additional *features* that appear in the step.

searchers. (Kappa=0.732).

- *Features*: 55 coding assignments, of which 51 were identically coded by both researchers. (Kappa=0.815).

This demonstrates a "substantial" to "almost perfect" agreement [34] for each of the components. Therefore, the author coded an additional 19 protocols, bringing the total number of coded protocols to 20. The coded protocols include randomly chosen recent free protocols from CSHP's website where ten were from the Bioinformatics category, five from the Neuroscience category, and five from the Laboratory Organisms category.

3.3.2 Coding scheme for references

To address RQ3, we developed a separate coding scheme for references to other instructions, protocols, papers, or policies. The coding scheme was developed through a similar consensus-based approach as described in Section 3.3.1.

References in the analyzed protocols can be separated into two broad categories. The first category includes references to other instructions, protocols, or policies that are intended to be *integrated* into the execution of the current protocol. The second category includes references to other documents that provide supplementary information, but are *not integrated* into the execution of the protocol.

Within our coding scheme, each reference is assigned (1) a *reference type*, which indicates the type of document referred to (e.g. manufacturer instructions, institutional protocol); (2) either an *integration strategy* for integrated references (e.g. use concurrently), or alternatively the *role* of a non-integrated reference (e.g. background information); (3) zero or more additional features. The complete coding scheme is described in Tables 3.4, 3.5, 3.6, and 3.7.

To establish agreement in the use of the coding scheme, two researchers coded the same two protocols. Again, we apply Cohen's Kappa as a conservative measure of agreement for each of the coding components. The results of this analysis are listed below.

- *Reference Type*: 57 coding assignments; 50 were identically coded by both researchers (Kappa=0.748).

Reference Type	Example
`manufacturer` *Manufacturer instructions*	"... according to the manufacturer's recommendation ..." [4974; 4.i]
`academic` *Academic protocol or paper*	"This can be accomplished in yeast by overexpressing herpes simplex virus thymidine kinase to phosphorylate nucleosides (Lengronne et al. 2001)" [5385; 2.ix]
`laboratory` *Institutional protocol or policy*	"... using standard restriction digestion protocols." [5168; 7.iii]
`equipment` *Equipment instructions*	"For other array platforms, perform washes as directed by the microarray supplier." [5385; 8.ii]

Table 3.4: *Type* of document referred to.

Integration Strategy	Example
`before` *Use reference before starting*	"The double-stranded RNA needed for RNAi is prepared as in Preparation of Double-Stranded RNA for Drosophila RNA Interference (RNAi)." [4918; Related Information]
`concurrent` *Use reference in parallel*	"Imaging can also be performed simultaneously with ..." [5427; Discussion]
`splice` *Use reference, then continue*	"Combine PCR products of replicate samples and purify them using the QIAquick PCR Purification kit ..." [4974; 4.i]
`merge` *Combine reference with protocol in an unspecified way*	"... the power of such a screening method is further enhanced when it is combined with the simple pyro-screening enrichment protocol, another pyrosequencing-based innovation we have developed (Liu et al. 2009)." [5491; 9.i]

Table 3.5: *Strategy* for integrating a reference into the protocol.

Non-integration Role	Example
`derivation` *Derived from reference*	"The 6C assay combines three different methodologies: chromosome conformation capture (3C) (Dekker et al. 2002), chromatin immunoprecipitation (ChIP), and cloning (Fig. 1)." [5168; 1.i]
`comparison` *Similarity to reference*	"... it may be useful to employ other labeling methods, such as those described in ..." [5460; 1.i]
`background` *Additional information*	"... (for details, see CSH Protocols articles Embedding Mouse Embryos and Tissues in Wax and Sectioning Mouse Embryos)." [4820; 4.ii]

Table 3.6: *Role* of a non-integrated reference.

Features	Example
`modifications` *Use reference with modification*	"If another species is used, surgical conditions and drug dosages should be optimized for that species." [5410; 2.iii]
`link` *Location/link to external source*	"For imaging cleared embryos, see Preparation of Fixed Xenopus Embryos for Confocal Imaging (Wallingford 2010b)." [5427; 5.i]

Table 3.7: Additional *features* that appear in the reference.

- *Integration Strategy*: 55 coding assignments; 50 were identically coded by both researchers (Kappa=0.914).
- *Non-integration Role*: 57 coding assignments; 55 were identically coded by both researchers (Kappa=0.737).
- *Features*: 58 coding assignments; 51 were identically coded by both researchers (Kappa=0.611).

This shows "substantial" to "almost perfect" agreement [34] for three of the four components. We observed only "moderate" agreement on the *features* component. Since this category has a small number of possibilities, and a high frequency for one of them (the `link` reference type), it is more difficult to distinguish the raw agreement rate we observed (88%) from chance. Since the accuracy of this component is not central to our results, we did not perform another iteration. Therefore, the author coded the remaining 18 protocols included in our data set.

3.4 Results

In this section we summarize and interpret the results of the study in response to the research questions listed in Section 3.1.2. In the spirit of open science, our raw results (coding data and calculations) are published in-full online.[3]

3.4.1 Precision of protocol steps

In contrast to Licklider's claim that "instructions directed to computers specify courses; instructions directed to human beings specify goals" [38], we found a surprising number of steps in the form of straightforward `instructions` (Table 3.2). These are expressed as directives to the user that contain little to no ambiguity about the task the user is to complete:

- *"Wash the cells with 5 mL of ice-cold PBS containing protease inhibitor cocktail." [5168; 4]*

[3]`https://github.com/lambda-land/ProtocolStudy-Data`

- *"The moment that the slide lightly taps the tip and the tip breaks, a small amount of the dye will leak from the needle tip. Release the 'clean' button."* *[4918; 7.ii]*
- *"Remove the Schneider's insect medium from the samples."* *[5460; 6]*

Overall, we observed 777 examples of direct instructions out of 833 total coded steps (93.3%).

The next most precise kind of step was also the second most frequent. A `goal-directed` step describes an expected goal, but leaves a degree of ambiguity about *how* the user is to achieve it. Most examples of `goal-directed` steps we observed involve bringing materials to a particular physical state such as temperature or concentration:

> *"Add BrdU to a final concentration of 400 μg/mL and incubate for the desired period."* *[5385; 2]*

Others instruct the actor to produce a measurement or calculation without spelling out a procedure:

> *"Calculate the percentage of the total responses."* *[5453; 12]*

Overall, we observed 33 examples of `goal-directed` steps, across 11 of 20 protocols.

Moving further from straightforward instructions, a `task-directed` step describes a task where the goal itself is ambiguous or depends on the higher-level goals of the person carrying out the protocol. Many of the examples we observed instruct the actor to perform an evaluation without providing an explicit indication of what the actor should be evaluating or looking for:

- *"Analyze 10 μL of the PCR products on a 1.5% agarose gel to check the quality of the amplification."* *[4974; 12]*
- *"Using a microscope (100X and 400X total magnification), evaluate the degree of digestion."* *[5014; 13.ii]*

We observed 21 examples of `task-directed` steps, across 12 of 20 protocols.

Finally, we observed `open-ended` steps in just two of the protocols. These steps are the least precise. They provide only a general description of the intent of a step but leave the goal and task unspecified.

> *"DNase concentrations to achieve optimal smearing sizes may differ for each cell line and therefore must be determined empirically for each cell type."* *[5384; 20]*

This type of step seems to require more creativity or ingenuity by the user. As a collection of concrete and highly-polished processes, it is not surprising that our data set provides relatively few examples of such steps.

3.4.2 Linearity of protocols

Overall, the protocols we analyzed were surprisingly linear. That is, they consist mostly of long sequences of instructions with very few `branch` or `loop` steps.[4] The branches that do exist are expressed by directing the actor to skip ahead or back to a particular step (similar to a GOTO) depending on the result of a measurement or the task the actor is trying to accomplish. Similarly, loops are expressed by directing the actor to repeat a certain range of steps.

- *"Proceed with step 7 during centrifugation . . . "* *[5384; 6]*
- *"Repeat this step with the upper phase . . . "* *[5168; 41]*
- *"Repeat steps 25.iii–25.v."* *[5168; 25.vi]*

We observed that only 10 of 20 protocols include any form of branching or looping at all, and among these non-linear protocols, they averaged just 2.3 `branch`/`loop` steps per protocol. One protocol [5385] contained 8 `loop` steps, but no other protocol had more than 3 branches and/or loops. When compared to all steps, `branch` and `loop` steps were just 2.76% of the total coding instances.

In a few instances, instructions resemble higher-order functions [58], such as the following `map`-like step that applies an instruction to every member of a set.

> *"Slowly pipette 120 μL of nuclei suspension into tubes #1–#7 . . . "* *[5384; 9]*

[4]However, we did observe many references to external protocols, which correspond roughly to subroutine calls and complicate the question of control flow somewhat. See Section 3.4.4.

Higher-order functions may be used to "flatten" a non-linear portion of a protocol to make it appear more linear. A single step that maps an instruction over multiple physical elements (tubes, phases, beads) is structurally simpler than `branch`/`loop` steps that reference other steps by name.

3.4.3 Accommodating imprecise execution

Previous research has shown that protocols are only loosely followed in practice (see Section 3.2). A protocol describes an idealized course of action, but the person executing it will frequently deviate from this course by modifying, reordering, adding, or even skipping steps. By way of analogy, consider a program or protocol as describing a path. In a computer program, the path specified is very narrow, like a tightrope, which the computer follows precisely. In a human protocol, the path is much wider, and the person following it may wander freely from side to side, "stop to smell the flowers", and so on.

We observed six different features within the analyzed protocols that explicitly accommodate this imprecise execution model by broadening or narrowing the path or by helping the user find their way back when lost. These are the first six features listed in Table 3.3. The frequency of each of these six features is summarized in Table 3.8.

Code	No. of protocols	No. of instances
`advice`	19 (95%)	108 (12.9%)
`constraints`	20 (100%)	108 (12.9%)
`expected outcomes`	9 (45%)	18 (2.04%)
`optionality`	10 (50%)	23 (2.76%)
`wiggle room`	14 (70%)	35 (4.20%)
`contingencies`	6 (30%)	6 (0.72%)

Table 3.8: Frequencies of features supporting the imprecise execution of protocols.

Broadening. The features `optionality` and `wiggle room` represent attempts by the protocol writer to *broaden* the path. Quite different from anything found in computer programs, steps that exhibit these features explicitly encourage actors to vary from the narrowest interpretation of the step semantics and rely on their own judgment.

Steps with the `optionality` feature indicate explicitly that a step is optional. While people are free to use their judgment to skip steps at any point in a protocol, the optionality feature highlights to the actor that a particular step is not essential to the ultimate goal of the protocol.

> *"At this point, the cell pellet can be snap-frozen in liquid nitrogen and stored at -80° C until use."* *[5168; pg 5]*

Similarly, steps with the `wiggle room` feature provide an explicit range of execution, suggesting, for example, that a particular measurement can be estimated.

> *"Incubate the mixture for 15-30 min at 70° C . . . "* *[5491; 12.ii]*

Narrowing. The features `advice`, `constraints`, and `expected outcomes` represent attempts by the protocol writer to *narrow* the path. These features were much more common. The presence of narrowing features makes it clear that protocol writers assumes that actors will wander from the path they are describing. These features focus the actor's attention on important aspects and expectations of the protocol.

Steps with the `advice` feature provide additional tips to help the actor navigate the most important sections of the protocol.

> *"For best results, inject the embryos with the needle at an angle slightly greater than 45° relative to the embryo surface in the posterior quarter of the embryo . . . "* *[4918; 8]*

Steps with `constraints` explicitly constrain the execution path to ensure that the actor doesn't deviate at an important point. For example, the following instruction about a chamber is followed by a constraint about its positioning.

> *"Close the hybridization chamber tightly and incubate it in the dark for 16 h in a 60° C water bath. The hybridization chamber should be submerged in water, but direct contact with the bottom of the water bath should be avoided."* *[4974; 29]*

Constraints provide a way to alert the actor that an extra level of care and precision should be used on a particular step.

Finally, steps with the `expected outcomes` feature provide a sort of checkpoint after an instruction to ensure that the actor can recognize that they are still on the right path.

> *"This step typically takes 12 min at 14,000g."* [4974; 20.iii]

Expected outcomes are similar to assertions in computer programming languages.

Finding the way back. Steps with the `contingencies` feature provide an explicit way to recover from errors, which might have been caused by deviating from the protocol.

> *"Use the markings on the 1.5-mL tube to provide an estimate of the pellet volume. If the pellet is very small, resuspend it in 300 μL of cell lysis buffer."* [084848; 19]

This feature was quite rare: only 6 instances in our data set. It seems that actors are typically expected to recover from errors on their own.

3.4.4 Integrating other instructions and protocols

One of our research questions (RQ3) concerned relationships between protocols and other instructions, whether protocols refer to each other and how they integrate. We observed that protocols refer to other resources very often. We identified a total of 261 references across all 20 protocols. The vast majority of these (246, 94.3%) were to explicitly named instructions or protocols (e.g. cited academic protocols or specific manufacturer's instructions), while the remainder were generic references to documents in the actor's lab (e.g. lab policies or ethics documents). We coded all 261 references using the coding scheme summarized in Tables 3.4, 3.5, 3.6, and 3.7.

Recall from Section 3.3.2 that we can classify references according to whether they affect the semantics of the current protocol, that is, whether they *integrate* with the protocol or merely provide background or supplementary information. We observed that 220 (84.3%) of the references integrated with their protocol. The observed instances of each integration strategy are summarized in Table 3.9.

The `splice` integration strategy most often occurred with manufacturer's instructions for lab "kits", packages that include solutions and equipment needed to do a common lab procedure. A spliced-in reference is similar to a subroutine call in a computer programming language.

A large source of `before` and `concurrent` integrations was a list of "recipes" and "warnings" given in the preamble of every protocol. Recipes are protocols or instruc-

Code	No. of protocols	No. of instances
`before`	20 (100%)	99 (45.0%)
`concurrent`	16 (80%)	61 (27.7%)
`splice`	13 (65%)	39 (17.7%)
`merge`	8 (40%)	21 (9.5%)

Table 3.9: Frequency of integration strategies for references to other instructions and protocols.

tions that must be followed before the current protocol can be executed. Warnings are additional instructions that must be kept in mind during the execution of the protocol, typically for safety reasons (coded as `concurrent`). For example:

> *"<!>DTT (Dithiothreitol; 0.1 M)"* [5559; materials]

This warning refers to a brief appendix entry:

> *"DTT (Dithiothreitol) is a strong reducing agent that emits a foul odor. It may be harmful by inhalation, ingestion, or skin absorption. When working with the solid form or highly concentrated stocks, wear appropriate gloves and safety glasses and use in a chemical fume hood."* [5559; cautions]

This DTT warning is typical of a `concurrent` integration; it alters the semantics of the rest of the protocol by specifying how to handle DTT safely. In fact, DTT appears only once in the protocol among a list of ingredients in step 5 [5559; 5]. While this step just says to "prepare the mixture", the warning is much more specific and demanding: the worker must wear gloves and glasses and perform the action in a specialized workspace. Also, unlike a spliced integration, a concurrent integration is applied implicitly throughout a protocol, to be triggered when relevant. In this example, the reference appears at the top of the protocol; the user must notice it and interleave it appropriately wherever DTT appears later on. Sometimes the integration point(s) are left even more implicit:

> *"It is essential that you consult the appropriate Material Safety Data Sheets and your institution's Environmental Health and Safety Office for proper handling of equipment and hazardous materials used in this protocol."* [085076; materials]

In some ways, the `concurrent` strategy is similar to aspect-oriented programming [30]. It separates concerns related to specific materials or actions and the actor is responsible for weaving those concerns into the execution of the protocol.

The `modification` feature captured 11 instances where a reference was integrated with some explicit changes.

> *"Purify the eluate using a MinElute PCR Purification Kit according to the manufacturer's protocol with the following modifications: ..."* *[5385; 33-33.i]*

Finally, we observed 21 instances of the `merge` integration strategy across 8 of the protocols. In some cases, these references seemed intended to help the user know when to choose *this* protocol by describing how the protocol might fit into a larger workflow and combine with other processes. The 17 instances across 8 protocols of the `comparison` role for non-integrated references also seemed tailored for this purpose, as illustrated by the following example.

> *"Pei et al. (1997) describe an alternative method to isolate Arabidopsis guard cell protoplasts suitable for patch clamping."* *[5014; pg 6]*

3.5 Discussion

Although the results in Section 3.4.1 show that protocols are *written* precisely, Section 3.2 discusses the substantial evidence that, in practice, protocols are not *executed* precisely. Rather, a protocol describes an idealized path to follow in order to accomplish a task. Protocol authors seem to be well aware of how protocols are used and tailor the presentation of protocols to support this execution strategy by alerting the protocol user about which shortcuts and side-trips are dangerous and which might be useful. By calling extra attention to especially important instructions, or providing checks to help the executor determine whether they are still on track to accomplish their task, the protocol author builds in checkpoints, redundancy, and safety measures to ensure the task will be successfully accomplished.

These features demonstrate how differences between humans and computers motivate differences between protocols and programs. On the one hand, the computer does exactly what it is told, which is useful for predictability and repeatability. On the other hand, the human can use common sense, which supports flexible error handling and recovery. Human common sense stands in sharp contrast to the commonly observed situation in imperative programs where the computer blithely chugs away after an error, unless the programmer has built in an explicit, complex system of fail-safes.

Similarly, the ambiguity and allowance for variation written into protocols are advantageous by allowing the user to *interleave* or *integrate* other plans: their own plans, goals, and constraints, and the auxiliary recipes, warnings, and procedures suggested by the protocol's author. Since protocol writers provide explicit allowances for variation, and explicit constraints and checks to reign in the variation they expect their users to make on their own, this suggests they know these features are important and must be supported if they want users to adopt their protocols.

The linearity that we observed in protocols may also play a role in supporting the integration of user goals and outside plans. Since protocol authors describe the path as a straight line rather than a garden of forking paths, they make it easier for users of the protocol to understand the document as a whole, and reason about the consequences of skipping, adding, or varying steps. That is, linearity allows the user a greater "field of view" when it comes to execution of the protocol. Rather than seeing a small slice of the process, they have access to many steps in advance, allowing them to plan and incorporate outside information in a more interactive way.

3.6 Implications for design

The design of literate "workflow" tools may benefit from our findings. Workflows, such as those provided by Bioconductor [23] or Galaxy [24], and interactive computation tools, such as IPython [49], are like protocols in that they describe a complex sequence of activities that a user may need to adapt to their own purposes, but the user is manipulating code and program entities instead of beakers and solutions. These tools could borrow the following features from protocols:

Path width. Authors of workflow *tools* could include explicit ways to add and execute invariants and tests that constrain a user's modifications to a workflow step. Or even without such changes, authors of *workflows* could add assertions and tests in locations where users are most likely to tinker with the script. Workflow users presumably use trial-and-error to adapt scripts and could benefit from such hints about the expected scope of the exploration process.

Connection to original. Workflow tools could preserve and make visible the relationship between an original demonstration protocol and any script the user creates by adapting it. When users edit example code to meet their own needs, the original becomes harder to recover and relate to the tailored script. This forfeits some of the benefits that protocols provide, since they no longer communicate adherence to a normative process, and their hints about the path and its width may be lost or damaged due to the user's experimentation.

Composition. Some workflow tools do not really allow for composition the way protocols do: IPython for example can only call plain Python code, and Bioconductor workflows can only call ordinary R packages. Perhaps auxiliary workflows could be referenced in some of the ways protocols are (Table 3.5) by adding facilities to help the user interleave the workflows' actions or using the assertions and tests mentioned above to check that their actions interleave correctly.

Previous work on variation languages is relevant for the design and implementation of these features. Potential techniques for adding variation points to programs and other data structures exist which could be used to explicitly broaden execution paths [16,65,66]; and we have identified potential editing techniques that could be used for maintaining a history of explicit variations which could also be used to maintain a connection to the original example as well as previously explored alternatives [67].

3.7 Conclusion

At the start of this study, we wanted to know how to improve the state of the art in mixed-initiative execution, to help people and computers collaborate more flexibly and effectively on tasks. This is important because in many cases, such as active learning scenarios, neither the computer nor the user alone have complete information about the sequence of tasks that need to be completed. To that end we investigated biological protocols, to mine them for ideas about what the human side of a human-computer program should look like. What we found embedded in these documents was a toolbox of ways to specify tasks, that take advantage of human capabilities:

- Ways of communicating both limits and liberties from the central activity sequence

- A simplified, linear structure, perhaps aimed at lowering the perceived risk of adoption, and cost of tailoring
- Simple statements of expected intermediate results in lieu of elaborate error-handling mechanisms

These tools seem immediately applicable to the design of workflow tools that help end-user programmers learn a complex task while adapting it to a real application. Beyond that, we have used these results to inform the design of more effective mixed-initiative features in programming languages in Chapter 4, in which human-computer interaction is a continuum of control, rather than as a function of “domination” by one or the other.

3.8 Acknowledgments

We would like to thank Margaret Burnett and Stephen Ramsey for helpful comments and advice in the early stages of this research, Stephen for directing us to CSHP as a repository of high-quality protocols to study, and the generous support of the Alfred P. Sloan foundation.

Chapter 4: Mixed-Initiative Drone Control

4.1 Introduction

The consumer and industrial market for unmanned aircraft systems (drones) has exploded in recent years, with the Federal Aviation Administration estimating that 2.5 million drones will be sold in 2016 in the United States alone [17]. These vehicles have a wide range of commercial, industrial, and scientific applications.

Drones are also an ideal platform for exploring *mixed-initiative computing* [29]—a model of computation in which human agents and software agents negotiate control of a process and collaborate to solve a problem. Current consumer drone software is already mixed-initiative (usually called semi-autonomous, in this context) since aspects of a single flight might be controlled remotely by a human pilot and others by an on-board computer. For example, a pilot might use a remote control to move a drone around and take pictures, while the on-board computer makes stability adjustments, implements a stationary hover, and automatically returns to its starting point if it loses contact with the pilot.

Unfortunately, the majority of programs written for drones are written in low-level languages like `C`. This makes it difficult to express more sophisticated interactions between the human pilot and the on-board computer, and also limits potential drone programmers to experts in embedded systems.

In this thesis, we present the early design of a domain-specific language embedded in Haskell for writing mixed-initiative flight control programs for drones. The practical goal of this language is to raise the level of abstraction for drone programming, increasing its accessibility and making feasible new kinds of drone programs that better exploit the strengths of both the human and computer agents in this mixed-initiative setting.

We are also interested in mixed-initiative computing in general. In Chapter 3, we describe a formative study of how people write programs (in the form of lab protocols) for other people to execute. From this study, we extracted design insights to best support the human agent in a mixed-initiative setting [1]. Therefore, a higher-level goal of this work is to put these design insights to practice, evaluating whether they are actionable and lead to a useful language design. In addition, we utilize design concepts gleaned from work being done in the area of mixed-initiative computing (Chapter 2).

In putting these design insights and concepts to practice we discuss some of the relevant background work done on drone languages (Section 4.2) and how they relate to the implementation of our proposed DSL. By way of explanation we provide small examples of our proposed DSL that highlight some of its interesting aspects (Section 4.3), and provide additional detail on the primitives and our interpretation of the agents involved in managing the control flow within the proposed DSL (Section 4.5).

We also provide extended examples that harness the power of our proposed DSL to manage control flows between human and computer agents (Section 4.6), and explain how our work relates to previous efforts made in managing interactions between agents (section 4.2). Lastly, we reflect on the effectiveness of integrating our design insights within the DSL implementation, and postulate on the direction of future efforts in managing human and computer interactions (Section 4.7).

The specific contributions of this work are therefore both: (1) the beginning work of the *DSL itself*, a high-level declarative language for a rapidly emerging application domain, and (2) a new *programming model* for mixed-initiative computing that supports sophisticated negotiation of control flow between human and computer agents.

The fundamental promise of mixed-initiative computing is that humans and computers have complementary strengths and weaknesses, and so together can solve problems more effectively than either could alone. Within the application domain of semi-automated drones (and also more generally), computers excel at executing repetitive strategies, performing rapid calculations, monitoring many sensors at once, and executing precise adjustments. Meanwhile, humans excel at visual recognition, recovering from errors, providing high-level oversight, and revising plans in in the presence of unforeseen events.

The goal of our proposed DSL is to allow programmers to take advantage of these complementary strengths. For example, in fully autonomous situations, planning for contingencies is a tedious, error-prone, and often incomplete process. A mixed-initiative program can instead simply transfer control to the human in such an event, relying on their judgment and creativity to recover. In Chapter 3 we discussed the fact that people write instruction sets (programs) for other people to execute. In these protocols the authors often provide only high-level error checks to ensure the program remains on the correct path. Much of the detail about how to recover from these errors is left to the protocol user.

At a higher level, consider a program for either surveying territory or conducting search-and-rescue operations, which are real-world tasks that employ drones and exemplify how the strengths of human and computer actors can complement each other in a mixed-initiative program. Such a program might have the drone automatically traverse an area while the human monitors camera output for potential objects of interest, taking over manual control when one is found. If, upon closer investigation, the object is not of interest, the human can return control to the auto-pilot to resume traversing the area. The computer can monitor other sensors (such as proximity and battery), and notify the human of significant events. The human can help the drone react to unforeseen circumstances and also adjust the high-level parameters of the search (e.g. expand or shrink the search area) to incorporate new external information as needed.

In Chapter 3, we identified several unique aspects of programs written for people and have used additional related work on how humans *use* plans and protocols [43, 61] to argue that these differences make sense given the respective strengths and weaknesses of humans and computers. We believe that a mixed-initiative programming language will be most useful if it takes both kinds of agents into account. A distinguishing features of programs for people is that they achieve flexibility, reusability, and robustness through *simplicity* and *rigidity*. The idea is that, to execute a plan and adapt it to new situations, a person should be able to understand how it works in its entirety. From this perspective, explicit error handling actually detracts from the utility of a program since it makes it more difficult to understand; it is better to rely on human judgment for robustness. Similarly, programs for people are mostly linear with minimal branching, looping, or other complex control flow. However, human protocols support a rather sophisticated form of composition by using common sense to interleave multiple protocols. This can be used in many scenarios where a computer program would use conditionals, for example, to interleave certain safety measures only when handling certain kinds of dangerous materials.

4.2 Related Work on Drone Languages

In Section 4.1, we mentioned that the majority of programs for drones are written in low-level languages like `C`. There is language support for writing `safe-C` code available, however there has not been much work on providing support for expressing sophisticated interactions between the human pilot and the on-board computer. The closeness of mapping between many of the libraries available and the `C` language they compile to also limits the pool of available drone programmers to those who are either experts in the domain, or those who have invested time in specializing their programming knowledge.

Languages and libraries also exist within Haskell for compiling or writing `safe-C` code, such as Ivory [15], CoPilot [52], Atom [27], and Feldspar [3] (the latter is a digital signal processing DSL). These types of `safe-C` languages are designed to be used with modern controllers and have been used to write drone programs by compiling `safe-C` code from Haskell that can be run directly on current controller hardware.

The focus of languages that compile to `safe-C` is to provide a secure method of producing low-level controller code. Being able to guarantee properties about a low-level language can be useful, but for our purposes this approach doesn't provide enough abstraction to deal with interactions in a meaningful way for the human pilot, which is the main focus of our implementation. The approaches presented by Ivory, its extension Tower, and the SMACCMPilot project that both of these languages support, does provide some useful information and groundwork for controlling drones in the physical world.

The SMACCMPilot project provides an excellent framework for understanding and utilizing the facilities of the drone system in an autonomous way. The programming model and design strategies supported by our proposed DSL provide the additional design principles necessary for supporting more flexible control flow for semi-autonomous systems (systems where human and computer interaction is desired). While the SMACCMPilot project has applicability in the implementation of our proposed DSL for real world flight control, it fills a different need than we address.

Copilot [51] on the other hand is a runtime system that generates streams of small constant-space and constant-time `C` programs that implement embedded monitoring. Because Copilot also generates its own scheduler, there isn't any need to implement a real-time operating system in conjunction with it.

CoPilot is another approach to dealing with `safe-C` programming. This stream-based dataflow language generates small constant-time and constant-space `C` programs that implement embedded monitors. Constant-space programming means that no dynamic memory allocations are needed. CoPilot periodically samples global variables of the program or programs; following a sampling-based monitoring strategy. The language itself provides mechanisms for controlling when to observe the variables.

By using the Atom compiler [27] for its back-end, CoPilot can also automatically generate its own periodic schedule. This negates the need for a real-time operating system in order to control scheduling and concurrency, allowing CoPilot to be easily executed on embedded hardware with minimal system resource availability.

4.3 Language Overview

In this section we provide an overview of unique aspects of the proposed DSL, as well as small examples that highlight these aspects. Our proposed DSL creates a flexible exchange of flow control between human and computer actors during the execution of programs written in our proposed DSL. Unique to our proposed DSL is the ability to program "monitors" providing assistance, guidance, and additional information to the human agent when some predetermined condition is met in the environment. In addition these monitors call on the human actor to troubleshoot unexpected issues encountered during execution.

One way to demonstrate the use of monitors in our proposed DSL, is through the tracking of and reaction to a prescribed state or location. To demonstrate the tracking of a waypoint—location in space where the drone is traveling—we use a changing integer value as a stand in marker. This marker increments as the drone makes movements in a three-dimensional plane, and once the desired value is reached an action is triggered. Either control of the process is transferred to a specified agent, or some auxiliary process is triggered.

4.4 Design Principles

The work presented on design principles in this section were originally implemented as part of a course project in a minimal form. The formative work included below represents a more in-depth representation of the intent of our proposed DSL. There has been some subsequent work done on the back end language features, and while we have explored several ways of providing the logic and features behind the drone control DSL, this work remains open.

4.4.1 Negotiation

The concept of negotiation of control can be seen as a way of managing collaboration between computer and human agents. Creating an useful programming abstractions makes it possible to free the programmer from the underlying detail of understanding any specific AI or computer systems employed. It also creates a flexible design that allows for different types of computer and intelligent systems to interact with human agents fluidly.

The main goal of a mixed-initiative programming language would be to support a flexible exchange of control between the computer and human agents. At a low-level this can be attained through four basic *primitive negotiating controls.* Although both the computer and human agents are always participating in parallel, one can imagine "control" as a token that gets passed between agents; whichever one has the token has exclusive access to certain functionality.

- `give`: Executed by the agent with the token to transfer it to another party.
- `offer`: Executed by the agent with the token to allow for the transfer of control to another agent.
- `request`: Dual to offer; executed by an agent without the token to request if from the agent with the token, which may be refused.
- `take`: Dual to give; executed by an agent without the token to immediately take it from the agent with the token.

To minimize the overhead of control negotiation for the human (e.g. minimize annoying and obtrusive dialog boxes), offers and requests are asynchronous. That is, an offer of control from A to B will succeed if B has made the request since A's last offer, otherwise the runtime system assumes the offer is declined. A synchronous offer can be made by giving control and then optionally requesting control back.

Because `give` and `take` are synchronous, the potential for deadlock exists (i.e. both the computer and human agents attempting to gain ownership of the token at the same time). Another way of expressing this is when A uses `take` to gain control from B, B cannot use `take` to regain control until the current process being performed by A has completed. Additionally A cannot use `give` to relinquish control to B until the current process being performed by A has completed.

However, the ability to adequately manage or eliminate such deadlocks is an extremely difficult and complicated issue. While we can minimize this effect by implementing a strict view like the one above, the problem of deadlock remains at the forefront of our design concerns.

An underpinning to the support of this flexible exchange of control between the computer and human agents is the view of programs as plans. These plans are blocks of executable code that contain a descriptive set of goal-directed tasks provided by the programmer. These blocks can either be executed in the synchronous order as provided by the programmer, or in an asynchronous order specified by the human agent or dictated by modifications to the plan by either the computer or human agents. The programmer can intersperse plans with various `offer` commands to provide control or conversely `request` can be used to interrupt synchronous execution.

4.4.2 Adaptable Autonomy

Plans are essentially a linear model of the program that embody the "ideal" execution of a series of actions or tasks. These plans represent the intent of the programmer, and provide necessary markers to indicate important or necessary steps along the way, but leave some of the ordering and a great deal of the error handling tasks at the discretion of the computer and human agents responsible for executing the plan.

In solving tasks or completing actions in a mixed-initiative program, the computer agent involved will potentially exercise a large spectrum of autonomy, yet should be limited to the level of autonomy necessary to complete a task. Additionally, this level of autonomy should not violate any mandates from the programmer or the human user involved unless there is some agreement reached between the computer and human agents. We can represent these concepts using parameterized negotiation primitives, where we have something like `give(`A`,`B`,plan,constraint)`, where `plan` is a series of `tasks`.

- `task`: An atomic representation of an action or calculation to be executed in the context of a plan.
- `plan`: A linear model of the program that is made up of tasks defined to achieve a given goal under the prescribed constraints.
- `constraint`: A mandate for the degree of autonomy being given during the negotiation of control. When a computer agent gives control to a human agent, it should provide a reason for the transfer and any information needed for the human to decide and act. When a human agent gives control to a computer agent, it should specify any constraints that must be followed—including if there is a desired point in the program that control should be returned.

A task included in the negotiation of control that contains only a goal and no additional plan, with few constraints, allows the computer agent to use a wider spectrum of autonomy in solving the task, whereas tasks specified as a sequence of actions and containing many constraints allows for more limited autonomy.

The idea of adaptable autonomy and decomposition go hand in hand. The ability to break plans into their individual components allows agents and programmers to manage the amount of autonomy exercised by the computer systems. Human agents are an integral part of execution and maintain this connection through selective access to processes by the computer systems that require collaboration. Decomposition is also a mechanism for supporting the *interleaving* and *integration* behaviors for adding or augmenting plans, similar to behaviors examined in [1].

Decomposition of tasks can also support a more sophisticated means of "triggering" action by computer systems. Such systems could be designed to provide triggering criteria based on any or all of the human agent inputs, the current state of the plan being executed, and any additional background data or contingencies represented by changes in the environment in which the plan is being executed. The use of control tokens can greatly simplify this by presenting cues about whether a computer or human agent is in charge, allowing them to focus and avoid the impossible task of searching through and processing the entire set of available data to determine which task or tasks are appropriate for a given point in the execution.

Another reason why working in such explicit models of control for constructing and executing plans is that it may present a critical component for effective adaptive learning by the computer and human agents. Since effective collaboration entails acting in a coordinated fashion—which in turn often requires a learning curve—cues are needed to guide expectations of which tasks meet the capabilities of either a computer or a human agent in different contexts.

4.4.3 Natural Communication

As stated previously, natural communication is an essential component of collaboration, and therefore of mixed-initiative interaction. We define mixed-initiative as the interaction and negotiation between computer and human agents that exploits the skill sets, capacities, and knowledge of each in developing a plan, executing that plan, and adapting to contingencies throughout execution. Mixed-initiative interaction involves a far reaching set of theoretical and pragmatic issues. Central to this is the ability for the human to give control to the computer systems—`give(`*A*`,`*B*`,task,constraint)`—and in a symmetric manner the ability of the computer systems to give control to the human—`give(`*B*`,`*A*`,plan,constraint)`.

There are myriad issues pertaining to safety, system security, agreement (trust), etc that have to be addressed throughout the execution of the program and can be formalized as sets of `constraints`. These `constraints` are associated with a particular `task` or set of `tasks`—`plan`—and are interpreted during the `negotiation` of control through the exchange of the control *token*.

To support exchanges efficiently, we establish an ongoing “dialogue” between the computer and human agents about how these exchanges will be organized. It is likely the human agent will need to maintain primary control of setting major plan objectives, but will want to relinquish control of more mundane plan formation and management. The majority of these dialogues may not take place using natural language—indeed this may not be the most efficient means of handling exchanges in some cases—though a variety of interaction styles will need to be supported.

4.4.4 Agency

Agency would necessarily take several forms in a DSL like this. At its core, the human and computer pilots represent agents. In addition to plans, drone software contains several programmed *monitors* (e.g. a battery level monitor, or an out-of-range monitor) that would need to negotiate for control when they are trigged. These programmed modules can also be seen as agents that vie for control of the system when necessary. Many of these monitors seek to prevent catastrophic failure, so they must also coordinate with the human actor, the plan, and other monitors when the situation dictates.

Each agent in the system is responsible for negotiating control of the system with the other agents. To mitigate this somewhat, the human agent is apportioned a more heavily weighted priority in the scheme of negotiation (except where catastrophic failure prevents this from being the case). The monitors are provided a broad, but more limited priority when it comes to preventing such catastrophic failures. Whereas the plans themselves take an advisory role, and are somewhat subservient to other agents.

4.5 Design Challenges

In a few aspects of our proposed DSL, there are fundamental tensions between the strengths, needs, and weaknesses of the human agent compared to those of the computer agents. These lead to interesting language design challenges.

The first such challenge is that we must provide a language that balances the traditional software engineering goal of writing robust software with our insights from previous work that human agents require a simple, concrete, and mostly linear plan in order to most effectively use a protocol/program. Our current proposal is to minimize the amount of error handling that makes its way into the high-level plan that the user will read and interact with. Instead, the high-level plan will consist mostly of linear sequences of steps with lightweight guideposts to help both the computer and human agents confirm that they are still on the right track. This not only increases the ability of the human agent to interact successfully with the program, but also takes advantage of the human ability to think on the fly, adapt, and recover from a wide variety of errors. This in turn might reduce development time since the language is more explicit about off-loading error checking and handling to the human agent. Lower-level error handling can still be implemented at the intraprocedural level within a particular step of the plan.

However, programmers are used to building in contingencies and error handlers to manage as many potential causes of failure as they can foresee, to make the software as robust as possible. This instinct is probably often good, even in our domain. On the one hand, we want to free programmers from the burden of programming for contingencies and take advantage of the human agent's ability to do this on-the-fly. On the other hand, we don't want to actively prevent programmers from actually dealing with the cases that they do foresee. This creates an interesting language design tension in our proposed DSL.

Another source of design tension exists in the simplicity needed in the negotiation of control. Other languages and program models mentioned in Section 4.2 manage the negotiation of control between several autonomous sources to accomplish some goal.

In these models, the negotiation can be a complex network of decisions navigated concurrently or in parallel by a set of intelligent agents. These networks are too complex for what we require in our proposed DSL. To some extent this is because the human agent is given a *priority* over other agents in the negotiation of control of the system. We have to provide enough freedom for the human agent to respond to situations and perform the tasks for which they are best equipped.

4.6 Extended Examples

We provide the following extended example of a search-and-rescue or territorial surveying program by way of illustrating the features of our proposed DSL in a concrete way. In this program the computer and human agents are tasked with searching a prescribed area or boundary for some intended target (e.g. a species of flora or fauna, or a person). These tasks are described as goal-directed instructions in the form of blocks of actions that are organized into plans using the programming model and negotiation primitives described in Section 4.5.

With our proposed DSL, the programmer writes a series of actions with any intended outcomes or expectations as well as any additional information they feel is relevant, and organizes them into a plan that makes sense for what is known about the scenario at the time. In this set of scenarios, the drone is controlled by the actions as they are laid out by the programmer in a plan, and the human agent is freed to control the camera and perform the visual tasks of the operation.

While the drone is somewhat autonomously performing the search program provided, the human agent can "tag" points of interest as either informational resources, or as a means of returning to them quickly if the need arises. To some extent takes advantage of a limited range of the faculties provided by the human agent, That being said, the programmer also intersperses `offer` commands with the action blocks that give the human agent the opportunity to make changes "on-the-fly."

If the user makes a `request` subsequent to the programmer providing an `offer`, the human agent is given control of the drone free of the current plan. From this point, the human agent can make changes to the plan (such as adjusting the search area, or creating additional waypoints to search), and can either complete tasks manually, or return control to the drone by selecting the action module they wish to continue autonomous drone control from.

In addition, the programmer creates several monitors (agents responsible for monitoring certain aspects of the drone state, and performing actions autonomously when necessary) in order to avoid catastrophic failure, or provide users with guidance when it looks like they may be veering off course. These monitors provide things like safe landing guarantees in the event the battery becomes discharged beyond a certain point, or providing suggestions to the user of they stray too far from the assigned (or modified search area).

In the case of the battery monitor, the monitor will `take` control of the drone, and execute a soft landing. This `take` command provides a lock on any future negotiations until the exception flag caught by the monitor is cleared. Here, the only remedy for the exception flag is a successful safe landing and subsequent battery replacement.

Where a monitor exists to provide the user with guidance to keep them in the correct search area, the monitor will `request` control from the user or provide a suggestion to the user about how to remedy the exception flag that was caught. This provides the human agent with the freedom to ignore the suggestion should the scenario at hand dictate such an action. Unless the human agent provides an `offer` for control, the `request` will be assumed to be declined, and the human agent will maintain control of the program flow.

4.7 Conclusion and Future Work

We have described the use of the design principles behind a DSL for building semi-autonomous drone programs that require potential or anticipated human agent participation. Ultimately it is our intention to implement a DSL language on a drone operating in physical space.

Many of the advantages of a DSL like this one for semi-autonomous drone programming are related to its implementation. Contingencies that cannot necessarily be conceived of by the programmer, can be altered in the field. And although not all contingencies are recoverable, our proposed DSL goes a long way in mitigating some of the unanticipated variability of operating a drone in the physical world.

Rather than spending undue time chasing all possible error points, programmers are freed to focus on programming tasks for the drone and the human actor in more meaningful ways. Programmers can still provide guidance for anticipated concerns in the completion of the necessary tasks, but our framework is flexible enough to give most of the recovery procedures over to the human agent in the field.

The programmer is exposed to a small, but meaningful, set of primitives for negotiating control of the flow of the program. This can be accomplished either synchronously or asynchronously. Whereas the human agent is provided with a simplified by high-level view of the entire program that is understandable and navigable. Our proposed DSL provides understandable units of execution, as well as information and some of the contingencies necessary to the completion of the task or tasks at hand.

In short, our proposed DSL will be able to take advantage of the faculties of both the human agent and the computer agent (represented by the programmer) in meaningful ways, without overwhelming either of them. We are able to accomplish this using negotiation primitives with behaviors designed to prevent deadlocks, and a simplified easy to navigate and compose interface for plans (programs).

Chapter 5: Conclusion

Since the publication of the paper included in this thesis, we have continued to refine our understanding of mixed-initiative programming and our ideas about where such a project could be headed in the future. This section will briefly describe some of the implications of our past research and understanding of work that has been done to drive future advances in mixed-initiative programming.

The fundamental promise of mixed-initiative programming is that humans and computers have complementary strengths and weaknesses, and so together can solve problems more effectively than either could alone.

Within the domain of mixed-initiative systems, computers and AI excel at executing repetitive strategies, performing rapid calculations, monitoring many sensors at once, and executing precise adjustments. Meanwhile humans excel at visual recognition, recovering from errors, providing high-level oversight, and revising plans in the presence if unforeseen events.

The goal of a mixed-initiative programming languages is to allow programmers to take full advantage of these complementary strengths. For example, in fully autonomous situations, planning for contingencies is a tedious, error-prone, and often incomplete process. A mixed-initiative program can instead simplify this by transferring control to the human user in such an event, and rely on their judgment or experiential knowledge or creativity to recover.

In our work on lab protocols, we found that when people write complex instructions (programs) for other people to execute, they often provide only high-level checks to ensure that the person executing the program is still on the "right track", however it is left to the user of the protocol to determine the method of recovery [1].

At a higher level, we consider the application of design implications from reviewed research done in the area of mixed-initiative programming, as well as that of lab protocols [1] in order to propose a DSL for mixed-initiative drone control. Such a DSL could be used to provide programs for drones—working alone or in "swarms"—in conducting search-and-rescue operations or surveying territory. These real-world applications employing the strengths of both computer systems and human users, shows how each can provide an interesting complement when managed using a mixed-initiative program.

For example, a programmer may code a drone to use a flight pattern to perform a search within a prescribed area or boundary, and in addition provide the human user with an intended search target and/or some set of goal-directed tasks that need to be accomplished. We know from previous research [1] that users adopt human instructions or programs more readily when they are supplied in a linear fashion (giving the user a high-level overview of the process). These instruction sets also need to provide users with derivation points that allow them the freedom to modify instructions as needed, while simultaneously providing guideposts for error recovery and locations to return to the original protocol.

In our study of lab protocols [1], we have identified several unique aspects of programs written for people compared to programs written for computers, and have used related work on how humans *use* plans and protocols [43,61] to argue that these differences make sense given the respective strengths and weaknesses of humans and computers.

We believe that a mixed-initiative programming language will be most useful if it takes both the computer system and human user into account.

A distinguishing feature of programs for people is that they achieve flexibility, reusability, and robustness through *simplicity.* The idea is that, to execute a plan and adapt it to new situations, a person should be able to understand how it works in its entirety. From this perspective, explicit error handling actually detracts from the utility of a program since it makes it more difficult to understand; it is better to rely on human judgment for robustness. Similarly, programs for people are mostly linear with minimal branching, looping, or other complex control flow.

However, human protocols support a rather sophisticated form of composition by using common sense to interleave multiple protocols. This can be used in many scenarios where a computer program would use conditionals, for example, to interleave certain safety measures only when handling certain kinds of dangerous materials.

We have used our understanding of the complexity behind programs for people to formulate design concepts for describing and writing mixed-initiative programs through a set of common *negotiation primitives*, that provide *adaptable autonomy* by breaking plans into their components, and also a means for developing *natural communications* between computer systems and human users.

Bibliography

[1] Keeley Abbott, Christopher Bogart, and Eric Walkingshaw. Programs for People: What We Can Learn from Lab Protocols. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 203–211, 2015.

[2] Apple. Mac basics:automator.

[3] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The design and implementation of feldspar. In *Symposium on Implementation and Application of Functional Languages*, pages 121–136. Springer, 2010.

[4] Niels Ole Bernsen. Foundations of multimodal representations: a taxonomy of representational modalities. *Interacting with computers*, 6(4):347–371, 1994.

[5] Niels Ole Bernsen. A reference model for output information in intelligent multimedia presentation systems. In *Proceedings of the ECAI'96 Workshop on: Towards a Standard Reference Model for Intelligent Multimedia Presentation Systems*, 1996.

[6] Niels Ole Bernsen. Defining a taxonomy of output modalities from an hci perspective. *Computer Standards & Interfaces*, 18(6):537–553, 1997.

[7] Giuseppe Bevacqua, Jonathan Cacace, Alberto Finzi, and Vincenzo Lippiello. Mixed-initiative planning and execution for multiple drones in search and rescue missions. In *Int. Conf. on Automated Planning and Scheduling (ICAPS)*, pages 315–323, 2015.

[8] Christopher Bogart, Margaret Burnett, Allen Cypher, and Christopher Scaffidi. End-user programming in the wild: A field study of coscripter scripts. In *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*, pages 39–46. IEEE, 2008.

[9] John M. Carroll. *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill.* MIT press Cambridge, MA, 1990.

[10] Cristiano Castelfranchi and Rino Falcone. Towards a theory of delegation for agent-based systems. *Robotics and Autonomous Systems*, 24(3-4):141–157, 1998.

[11] Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming. *Extreme programming examined*, pages 223–247, 2000.

[12] Philip R Cohen and Hector J Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261, 1990.

[13] Mehdi Dastani, M Birna van Riemsdijk, Frank Dignum, and John-Jules Ch Meyer. A programming language for cognitive agents goal directed 3apl. In *Int. Workshop on Programming Multi-Agent Systems (INF)*, pages 111–130. Springer, 2003.

[14] Clarisse sieckenius de Souza and Clara Faria Leitão. Semiotic engineering methods for scientific research in HCI. *Synthesis Lectures on Human-Centered Informatics*, 2(9781598299441):1–122, 2009.

[15] Trevor Elliot, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. Guilt Free Ivory. In *Haskell Symposium*, pages 189 – 200, 2015.

[16] Martin Erwig and Eric Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27, 2011.

[17] FAA. FAA Aerospace Forecast: Fiscal Years 2016–2036. Technical Report TC16-0002, United States Federal Aviation Administration, Washington, DC, USA, 2016.

[18] Rino Falcone and Cristiano Castelfranchi. The human in the loop of a delegated agent: The theory of adjustable social autonomy. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 31(5):406–418, 2001.

[19] George Ferguson and James F Allen. Arguing about plans: Plan representation and reasoning for mixed-initiative planning. In *Int. Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, pages 43–48, 1994.

[20] George Ferguson, James F Allen, Bradford W Miller, et al. Trains-95: Towards a mixed-initiative planning assistant. In *Int. Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, pages 70–77, 1996.

[21] Alberto Finzi and Andrea Orlandini. Human-robot interaction through mixed-initiative planning for rescue and search rovers. In *AI* IA*, volume 5, pages 483–494. Springer, 2005.

[22] Julia Rose Galliers. *A theoretical framework for computer models of cooperative dialogue, acknowledging multi-agent conflict.* PhD thesis, Open University, 1988.

[23] Robert C Gentleman, Vincent J Carey, Douglas M Bates, Ben Bolstad, Marcel Dettling, Sandrine Dudoit, Byron Ellis, Laurent Gautier, Yongchao Ge, Jeff Gentry, et al. Bioconductor: Open software development for computational biology and bioinformatics. *Genome Biology*, 5(10):R80, 2004.

[24] Belinda Giardine, Cathy Riemer, Ross C Hardison, Richard Burhans, Laura Elnitski, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, James Taylor, et al. Galaxy: a platform for interactive large-scale genome analysis. *Genome Research*, 15(10):1451–1455, 2005.

[25] Barbara J Grosz and Candace L Sidner. Attention, intentions, and the structure of discourse. *Computational linguistics*, 12(3):175–204, 1986.

[26] Barbara J Grosz and Candace L Sidner. Plans for discourse. Technical report, BBN Labs Inc., 1988.

[27] Tom Hawkins. Controlling hybrid vehicles with haskell. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. ACM, 2008.

[28] Eric Horvitz. Principles of mixed-initiative user interfaces. *ACM SIGCHI 1999*, pages 159–166, May 1999.

[29] Eric Horvitz. Principles of mixed-initiative user interfaces. In *ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 159–166. ACM, 1999.

[30] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP 1997-Object-oriented Programming*, pages 220–242. Springer, 1997.

[31] Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[32] Todd Kulesza, Simone Stumpf, Weng-Keen Wong, Margaret M. Burnett, Stephen Perona, Andrew Ko, and Ian Oberst. Why-oriented end-user debugging of naive bayes text classification. *ACM Transactions on Interactive Intelligent Systems*, 1(1):1–31, 2011.

[33] Jonas Kvarnström and Patrick Doherty. Automated planning for collaborative uav systems. In *Int. Conf. on Control Automation Robotics & Vision (ICARCV)*, pages 1078–1085. IEEE, 2010.

[34] J R Landis and G G Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[35] Tessa Lau, Eben M. Haber, Tara Matthews, and Gilly Leshed. Coscripter: Automating & sharing how-to knowledge in the enterprise. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 1719–1728, Florence, Italy, 2008. ACM.

[36] Neal Lesh, Joe Marks, Charles Rich, and Candace L. Sidner. 'Man-Computer Symbiosis' Revisited: Achieving Natural Communication and Collaboration with Computers. *IEICE Transactions on Information and Systems*, E87-D(6):1290–1298, 2004.

[37] Neal Lesh, Joe Marks, Charles Rich, and Candace L. Sidner. "man-computer symbiosis" revisited: Achieving natural communication and collaboration with computers. *IEICE Transactions on Information and Systems*, E87-D(6):1290–1298, 2004.

[38] J. C. R. Licklider. Man-computer symbiosis. *IRE Transactions on Human Factors in Electronics*, HFE-1(1), 1960.

[39] J. C. R. Licklider. Man-computer symbiosis. *IRE Transactions on Human Factors in Electronics*, HFE-1(1), 1960.

[40] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa a. Lau. End-user programming of mashups with vegemite. *IUI 2009*, pages 97–106, 2009.

[41] Greg Little, Lydia B Chilton, Max Goldman, and Robert C Miller. Turkit: human computation algorithms on mechanical turk. In *23rd Annual ACM Symp. on User Interface Software and Technology*, pages 57–66. ACM, 2010.

[42] Karen E Lochbaum. A collaborative planning model of intentional structure. *Computational Linguistics*, 24(4):525–572, 1998.

[43] Michael Lynch. Protocols, practices, and the reproduction of technique in molecular biology. *The British Journal of Sociology*, 53(53):203–220, 2002.

[44] David McNeill. *Hand and Mind: What Gestures Reveal About Thought*. University of Chicago press, 1992.

[45] C.A. Miller and Raymond Larson. An explanatory and "argumentative" interface for a model-based diagnostic system. In *ACM Symposium on User interface Software and Technology (UIST)*, pages 43–52, Monterey, California, United States, 1992. ACM.

[46] Robert Neches, Richard E Fikes, Tim Finin, Thomas Gruber, Ramesh Patil, Ted Senator, and William R Swartout. Enabling technology for knowledge sharing. *AI magazine*, 12(3):36, 1991.

[47] J. D. North. The rational behavior of mechanically extended man. Technical report, Boulton Paul Aircraft Ltd., Se[tember 1954.

[48] Ramesh Patil, Richard Fikes, Peter Patel-Schneider, Don McKay, Tim Finin, Thomas Gruber, and Robert Neches. The darpa knowledge sharing effort: Progress report. In *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92), San Mateo, CA*. Citeseer, 1992.

[49] Fernando Perez and Brian E Granger. IPython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.

[50] John Peterson, Gregory D Hager, and Paul Hudak. A language for declarative robotic programming. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1144–1151. IEEE, 1999.

[51] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot—a coprocessor-based kernel runtime integrity monitor. In *USENIX Conf.*, pages 179–194. San Diego, USA, 2004.

[52] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A Hard Real-Time Runtime Monitor. In *1st Int. Conference on Runtime Verification*, LNCS. Springer, 2010.

[53] Alexander J Quinn and Benjamin B Bederson. Human computation: A survey and taxonomy of a growing field. In *ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 1403–1412. ACM, 2011.

[54] Alexander J Quinn and Benjamin B Bederson. Human computation: A survey and taxonomy of a growing field. *SIGCHI Conference on Human Factors in Computing Systems*, pages 1403–1412, 2011.

[55] Charles Rich and Candace L Sidner. Collagen: A collaboration manager for software interface agents. *User Modeling and User-Adapted Interaction*, 8(3):315–350, 1998.

[56] Charles Rich, Candace L Sidner, and Neal Lesh. Collagen: Applying collaborative discourse theory to human-computer interaction. *AI magazine*, 22(4):15, 2001.

[57] Silvia Rossi, Enrico Leone, Michelangelo Fiore, Alberto Finzi, and Francesco Cutugno. An extensible architecture for robust multimodal human-robot communication. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 2208–2213, 2013.

[58] Peter Sestoft. Higher-order functions. In *Programming Language Concepts*, pages 77–91. Springer, 2012.

[59] C. L. Sidner, C. Rich, and N. Lesh. Collagen: Applying collaborative discourse theory to human-computer interaction. *AI Magazine*, 22(4):15–26, 2001.

[60] Cees GM Snoek, Marcel Worring, and Arnold WM Smeulders. Early versus late fusion in semantic video analysis. In *ACM Int. Conf. on Multimedia*, pages 399–402. ACM, 2005.

[61] L. A. Suchman. *Plans and Situated Actions: The Problem of Human-machine Communication.* Cambridge University Press, 1987.

[62] S. Timmermans and M. Berg. Standardization in action: Achieving local universality through medical protocols. *Social Studies of Science*, 27(2):273–305, 1997.

[63] Wiebe van der Hoek, Bernd van Linder, and John-Jules Ch Meyer. An integrated modal approach to rational agents. In *Foundations of rational agency*, pages 133–167. Springer, 1999.

[64] Birna van Riemsdijk, Wiebe van der Hoek, and John-Jules Ch Meyer. Agent programming in dribble: from beliefs to goals using plans. In *Int. Conference on Autonomous Agents and Multiagent Systems*, pages 393–400. ACM, 2003.

[65] Eric Walkingshaw. *The Choice Calculus: A Formal Language of Variation.* PhD thesis, Oregon State University, 2013. `http://hdl.handle.net/1957/40652`.

[66] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *ACM SIGPLAN Symp. on New Ideas in Programming and Reflections on Software (Onward!)*, pages 213–226, 2014.

[67] Eric Walkingshaw and Klaus Ostermann. Projectional Editing of Variational Software. In *ACM SIGPLAN Int. Conf. on Generative Programming: Concepts and Experiences (GPCE)*, pages 29–38, 2014. **Best paper**.

[68] G. L. Ward and L. R. Horn. *The Handbook of Pragmatics.* Blackwell Publishing, 2004.

[69] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers, Inc., 1988.