MICL: a Mixed-Initiative Control Language
DSL Report Card

Keeley Abbott
School of EECS
Oregon State University

June 11, 2016

1 Introduction

The number of semi-autonomous controlled vehicles and devices entering the consumer and industrial markets has increased sharply in recent years. They have become increasingly important research tools in many fields, including robotics, computer science, and biology in addition to military applications. The majority of programs written to control these devices and vehicles are written in low-level languages like `C`, which limits the amount of abstraction available to the programmers.

Here we present a domain-specific language embedded in Haskell, intended to lift the level of abstraction available to programmers who intend to not only utilize the abilities of the control system, but also incorporate human input. Unlike previous languages, whose primary intent is to describe a program executed strictly by the system, our language allows for more complex interplay between the initiatives represented by the system, and the goals of the users.

The specification of a program written in MICL is a description of the tasks to be completed by the system, as well as those that require human intervention or input (as they are seen by the programmer). In addition MICL provides the human actor the opportunity to interleave or intersperse their own goals and objectives as well.

1.1 Contributions

The contributions of this paper are as follows:

C1. a *description language* for controlling semi-autonomous vehicles and devices that incorporates mundane instruction sets to be executed by the system, as well as *goal-directed* instruction sets to be executed by the user;

C2. an *execution mechanism* for control sequences that allows the negotiation of *flow control* within the defined program between the system and the user.

In many cases, once a program is generated for a semi-autonomous vehicle or device, there is little question of what is *done* with that program once it has been written by the programmer. Our approach allows the human actor to provide additional insight based on previous experience or current circumstances that may not have been considered by the programmer.

Our approach in MICL is to provide a set of descriptive, strongly typed and composable language features. The features are such that the type of the resulting execution is evident in the provided features. We provide examples of executing the language and show the simulation of moves performed by the system, as well as those performed by the human actor.

MICL is an *embedded domain-specific language* within Haskell that presents a way for programmers to design mixed-initiative controller programs that are strongly typed. Type information for these programs is preserved, making it easy to "stitch" together existing processes into a larger flow for programs, as well as modify the order of events without harming the integrity of the overall system-provided goals.

2 Users

Our language is directed at programmers of automated vehicle and device control sequences who are seeking to interject human interaction into their routines. As a side effect, the human-readable portion of these control sequences is heavily influenced by previous work done on how people understand programs written by other people. [1]

Under normal circumstances these users have a plethora of domain knowledge in regard to writing programs for the system components they are targeting, but may not have as much knowledge about dealing with the users of these systems. However, they often have a good understanding of the parameters and requirements of the programs they are writing – where more flexibility can be provided to the users, and where they may need additional direction in accomplishing an assigned goal.

Users of the programs written may have little to no coding experience, however they should not need additional technical background in order to execute the scripts provided to them by the program. They will however need to have some specific knowledge regarding the vehicle or device they are assisting the operation of – how to take manual control when necessary, and how to return to a state of semi-autonomous control.

We intend our DSL to be a description of interactions between a programmer (represented by a compiled program), and a human actor. As such much of what we have done with this language is contained within simulations, and so the code generated bears little resemblance to the low-level code that would be necessary to actually execute these programs on controller boards. That being said, there has already been some work done in the area of providing type safe low-level code within Haskell. [3]

3 Outcomes

The output of programs written in our language are sets of control sequences interpreted by the system being controlled, as well as a set of annotated goal-directed tasks, providing the user with a protocol for the execution of the program. At the initialization of the program, the system will begin sending command sequences to the controller, and the user will begin executing their tasks using the instruction set provided.

The program provides the user suggestions about where they can deviate or alter the execution of the control sequence to accomplish any number of the goal-directed tasks provided. In addition it provides explicit locations where the user can or should return to a state where the control sequence can once again return to a state of autonomous action.

4 Use Cases / Scenarios

In this section we describe the constructs used to represent semi-autonomous device programs. We will introduce basic program constructs in Section 4.1 and discuss the special case of human actor interaction in Section 4.2.

4.1 Basic Program Construction

At its core, MICL provides a set of data types that represent actions that can be performed by semi-autonomous vehicles, which when strung together represent a flight path or driving directions for the vehicle or device. If these programs are interpreted they result in the vehicle or device servos being actuated in such a way that they accomplish the intermediary and longterm goals of the program. For example, two very simple programs might ask a drone to take off and climb to a determined height, or land safely.

```
takeOff :: State Status ()
takeOff f = do ascend fullPower `to` meters f

land :: State Status ()
land = do descend fullPower `to` meters 0
```

We take advantage of the nature of the `State Monad` to construct more complex programs from combinations of simpler ones using Haskell's `do` notation. For example, a "flight" program that uses the `takeOff` and `land` to safely both take off and land a drone.

```
takeOffAndLand :: State Status ()
takeOffAndLand = do takeOff (meters 250)
                    land
```

4.2 Interactive Program Construction

In order to achieve sharing of process flow control between the program as described by the programmer, and the goals presented to and derived by the human actor, it is necessary to model interactions between the two in order to negotiate this control. We represent these interactions to the user through the `Display` field in the `Status`, and they are modeled as either an instruction for the user to complete, or a waypoint for the user or the program to arrive at. The sharing of control is represented by the `OpMode` field in the `Status`, which indicates which `Agent` has control and the current `Mode` of operation.

```
type Status = (Location,Display,OpMode)

type Display = [Task]
type Task = Either Waypoint Instruction
type Waypoint = Location
type Instruction = String
```

When a task is either completed or is no longer deemed required by either party, it can be removed from the display. Additionally, waypoints are removed once the user or the program has visited the location associated with that waypoint.

We provide interaction capabilities to the user through the use of the `interaction` construct, which is nested inside of another set or sets of instructions written by the programmer. This allows the user to control

the drone, until some constraint is meant, at which point control returns to the system and the remainder of the program is executed as needed.

```
interactiveFlight :: State Status ()
interactiveFlight = do takeOff (meters 250)
                       forward fullPower `to` meters 250
                       strafeR fullPower `to` meters 250

                       newTask (waypoint (north 0,east 0,down 0))
                       newTask (instruction "return the drone to its home waypoint.")

                       updateOpMode (wait Human)

                       while (untilW [backward fullPower,strafeL fullPower]
                              (north 0,east 0,down 250))

                         until0 interaction (Computer,Wait)

                       land
```

From the `interaction` command the user can provide additional input signals to the drone in order to control its flight path, or to make changes to the `Display` being presented to them. These input signals are accumulated by the state monad, and the changes are reflected to the user, and in the continuation of the program once the user returns control to the system.

5 Basic Objects

The basic objects of MICL include signals that are passed between the program and the device, or between the human actor and the device. These `Signal`s represent the operating modes of the device (whether the human or computer actor is supplying the input and whether the device or vehicle is in a "recovery", "normal", "return" or "wait" mode), as well as the orientation signals that control the actuation of servos on the device.

```
data Input = Input { channel :: Agent
                   , mode :: Mode
                   , enable :: Bool
                   , roll :: Float
                   , pitch :: Float
                   , gaz :: Float
                   , yaw :: Float
                   }

type Output = Task
```

```
type Task = Either Waypoint Instruction

type Waypoint = Location

type Instruction = String

data Agent = Human
           | Computer

data Mode = Recovery
          | Normal
          | Return
          | Wait

type OpMode = (Agent,Mode)
```

We use relative location for the device in order to determine when waypoints have been reached in addition to assigning new waypoints. This method of calculating the current location of the drone is known as dead-reckoning [5].

```
type Location = (North,East,Down)

type North = Float
type East = Float
type Down = Float
```

While we have addressed some of the temporal aspects of drone operation through our DSL in Section 6, another way of addressing this in the future would be to include *functional reactive programming (FRP)* [6]. In addition to providing more concrete representations of time as well as functions for dealing with time-varying values, FRP would provide a more concrete mechanism for reacting to user input.

Additionally, in many cases current drone manufacturers choose to eliminate the variable of drone orientation from their representations entirely (in fact this is fairly standard). While this eliminates a whole host of issues when it comes to calculating location (i.e. because north is always north you don't have to adjust when the drone spins), it is on the whole a somewhat unsatisfying solution. As such, there is still work to do in addressing this issue from the standpoint of MICL.

6 Operators and Combinators

When a `Signal` is received we have to transform that into some meaningful transformation of the `State`. In order to accomplish this, we have several operators for dealing with the different types of signals we might expect the programmer to anticipate, as well as those signal types we have to anticipate from the human actor. Addressing movements is the chief among these.

```
type Program a = a -> State (Time,Status) ()

move :: Program Input
move sig = do (t,(loc,dis,opm)) <- get
              put (time (addTime tick (t,(loc,dis,opm)))
                  ,(locate sig loc,dis,opm))
              (t,(loc,dis,opm)) <- get
              put (t,(loc,clearWaypoint loc dis,opm))
```

In any move command we have to update the current location of the device using the device's maximum rate of travel and the throttle provided by the programmer or the human actor.

```
locate :: Input -> Location -> Location
locate inp loc = (locateNorth ((pitch inp) * rate)
                  (locateEast ((roll inp) * rate)
                   (locateDown ((gaz inp) * rate) loc)))

locateNorth :: Float -> Location -> Location
locateNorth f (n,e,d) = (n + f,e,d)

locateEast :: Float -> Location -> Location
locateEast f (n,e,d) = (n,e + f,d)

locateDown :: Float -> Location -> Location
locateDown f (n,e,d) = (n,e,d + f)
```

In addition, we have to increment `time` as movement commands are executed, in order to accurately reflect the amount of time that has elapsed in the execution of a movement, and to more precisely calculate the location using dead-reckoning. To accomplish this, we store the current "time" in the `State`, and update it as actions that consume time are executed.

```
addTime :: Time -> (Time,Status) -> (Time,Status)
addTime t (t',s) = (t + t',s)

time :: (Time,Status) -> Time
time (t,s) = t
```

Crafty readers may have noticed there was an additional check for the removal of waypoints at the end of the move command. As discussed in Section 4.2 waypoints are removed from the display once they are visited, which means every time we move we have to check to see if a waypoint has been visited and clear it.

```
clearWaypoint :: Location -> Display -> Display
clearWaypoint _   []     = []
clearWaypoint loc (d:ds) = case d of
  (Left loc')
    | loc       == loc' -> ds
    | otherwise         -> d : clearWaypoint loc ds
  _ -> d : clearWaypoint loc ds
```

In addition to movements, we are concerned with understanding and potentially changing the process control flow between the program and the human actor. As such, we must also update and maintain the current signal origination and the current mode of operation which were both discussed in Section 5.

```
updateAgent :: Program Input
updateAgent sig = do (t,(loc,dis,opm)) <- get
                     put (t,(loc,dis,(changeAgent opm,snd opm)))

updateMode :: Program Input
updateMode sig = do (t,(loc,dis,opm)) <- get
                    put (t,(loc,dis,(opm fst,changeMode opm)))

changeAgent :: OpMode -> Agent
changeAgent opm = case (fst opm) of
  Computer -> Human
  Human    -> Computer

changeMode :: OpMode -> Mode
changeMode opm = case (snd opm) of
  Recovery -> Normal
  Normal   -> Wait
  Return   -> Wait
  Wait     -> Normal
```

MICL is temporarily limited in the amount of interaction that is provided between the program (representing the programmer) and the human actor. There are further plans to address this as outlined in Section 5 through the use of functional reactive programming. Additionally, we discuss the temporary limitation that exists in the execution of orientation changing commands in Section 5.

One of the fundamental limitations of MICL is it's inability to compile down to low-level code necessary to execute on most modern controller hardware. As discussed in Section 2, there is a plethora of work out there for converting Haskell code into `C` code (which is the standard language among controller hardware).

7 Interpretation and Analyses

Execution of simulations of programs within MICL is handled through the GHCi terminal screen using the `State` monad, and the `runState` function. When programs are executed, they also take advantage of Haskell's `IO` monad to display status information to the user, as well as tasks and waypoints provided by the program. `IO` is also the means of receiving commands from the user, and transmitting the signals back to the drone.

In order to execute a program in MICL, the user provides the program to be run, and an initial `state` for the program to operate on. In MICL we provide a `statusDefault` construct that initiates the drone with a starting location of `(north 0,east 0,down 0)`, an empty beginning display, and an `OpMode` of `(Computer,Normal)`.

```
statusDefault :: (Time,Status)
statusDefault = (seconds 0.0,((north meters 0.0,east meters 0.0,down meters 0.0)
                              ,[],(Computer,Normal)))

> runState simpleGrid statusDefault
((),(62.0,((0.0,0.0,0.0),[],(Computer,Normal))))
```

Interrupts in the program provide the user with an `IO` screen with command options for the user to execute.

8 Cognitive Dimension Evaluation

8.1 Abstraction Gradient

MICL is an *abstraction-tolerant* language that permits the use of constructs exactly as they come, however owing to its affinity for composing functions and programs, it allows for the creation of additional abstractions. The level of abstraction available from MICL is initially low, which allows it to be used by less experienced programmers more easily. Novice programmers can start by understanding the high-level concepts behind signals.

8.2 Closeness of Mapping

Problems that can be modeled in MICL as programs are mapped relatively closely to those that can be observed in the real world. We are able to map entities in the user's task domain directly to task-specific program entities, and operations on the program entities can be likewise mapped directly onto program operations.

8.3 Consistency

One area that MICL has issues in accomplishing is consistency. While the constructs within the language are fairly easy to understand once they have been learned, it may be more difficult for users to "guess" at additional constructs they may not have learned yet. In addition, the `do` notation that we rely on may present a barrier to users of other languages where this type of notation has a different meaning.

8.4 Diffuseness/Terseness

MICL has a relatively compact field of notation to achieve resulting programs written in the language. As such, and also owing to its relative closeness of mapping with the problem world, it offers a fairly terse representation to users and programmers. MICL requires relatively few lexemes to achieve resulting programs.

9 Implementation Strategy

Using a shallow embedding allows us to take advantage of Haskell's `do` notation as a way of representing program syntax within MICL. Additionally, we have the luxury of easily composing programs that have already been written by using of `State` management using monads. However, because MICL does not compile to low-level code there is some additional work that would need to be done in order for programs to be run directly on vehicle or device hardware.

Implementation of MICL in a *language workbench* might be advantageous, because we could compose MICL with some similar EDSLs that do compile down to low-level code that can be run directly on controller hardware. One example of a language that it would be advantageous to compose MICL with would be Ivory [3]. The framework is already there to compile Haskell code to `safe-C`, and by taking advantage of this we need not compile the code from directly within MICL.

10 Find Similar/Related DSLs

As we mentioned in Section 2 there exist languages and libraries within Haskell that deal with compiling or writing `safe-C` code, such as Ivory [3], and Copilot [4]. Both of these language extensions are designed to be used with controllers, and compile `safe-C` code from Haskell that can be run directly on current controller hardware.

Ivory has been used to implement code that can be run directly on Ardupilot hardware, and has been used to generate over 200,000 lines of code for SMACCMPilot as well. In addition, this language has been used in Boeing's STANAG486 program to pilot an unmanned helicopter through real-world situations [2]. Unfortunately the structure of the language presented by Ivory can be a bit difficult to decipher at times, and it is unknown if the design decision to map the language closely with the compiled code represents an additional difficulty for programmers.

Copilot 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.

Where both of these approaches differ from MICL, is in the availability and representation of interactions between the system and the human actor executing the program. In MICL we treat these interactions as first-class citizens, rather than as an afterthought.

11 Design Evolution

When we originally began working with the `State` monad, we thought of a `Program` as simply a `Signal -> State Status ()`, however as we worked with this design, it became increasingly apparent that we were going to have to take `Time` into account.

```
type Program = Signal -> State (Time,Status) ()
```

This allowed us to include time-based functions that would execute a given program at a specified time as well, or react to the passage of time in different ways.

```
at :: Program Input -> Time -> Input -> State (Time,Status) ()
at prg t inp = do (t',s) <- get
                  case (t == t') of
                    True  -> prg inp
                    False -> return ()

untilT :: [Input] -> Time -> State (Time,Status) ()
untilT []     tme = do return ()
untilT (i:is) tme = do (t,s) <- get
                       case (t == tme) of
                         False -> i `to` (projectLoc i s) >> untilT is tme
                         True  -> return ()
```

In addition, we realized there were different signal types we would have to take into consideration – `Input` and `Output` signals. `Input` signals are those that provide instructions to the drone in order to actuate motors and control the amount of power being sent to them. `Output` signals are those that provide instructions and information to the user about the goal-directed tasks.

```
type Program a = a -> State (Time,Status) ()
```

One advantage of this design, is that we can accommodate other types of `Signals` provided by the programmer that may not have already been considered in the current design.

12 Future Work

In order to resolve the temporary limitations of the language mentioned in Section 6 we intend to implement MICL as a deeply embedded FRP language in Haskell. The long-term goal being to provide more seamless interaction between the system and the human actor, as well as allow us to perform additional analysis on the language.

Another advantage of incorporating FRP into the design of the language is the ability to further refine the `Display` system within MICL. Most FRP libraries also include visualization extensions that will allow us to provide the user with a more polished GUI. This will also give us the ability to display the user instructions and waypoints in a more human-readable protocol-like manner.

Currently MICL is focused on the domain of drones, but there are other controller environments that could take advantage of the language we have built. In order to do so, we need to generalize the language constructs further, and this can potentially be assisted through the *deepening* of the language into a *final* form.

References

[1] K. Abbott, C. Bogart, and E. 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] Boeing. Unmanned Little Bird H-6U. `http://www.boeing.com/defense/unmanned-little-bird-h-6u`, 2016. [Online; accessed 01-June-2016].

[3] T. Elliot, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury. Guilt Free Ivory. In *Haskell Symposium*, pages 189 – 200, 2015.

[4] L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A Hard Real-Time Runtime Monitor. In *1st Int. Conference on Runtime Verification*, LNCS. Springer, 2010.

[5] Wikipedia. Dead reckoning – Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/wiki/Dead_reckoning`, 2016. [Online; accessed 01-June-2016].

[6] Wikipedia. Functional reactive programming – Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/wiki/Functional_reactive_programming`, 2016. [Online; accessed 01-June-2016].