Modelling the Turtle Python library in CSP

Dara MacConville Marie Farrell Matt Luckcuck
Rosemary Monahan
Department of Computer Science/Hamilton Institute, Maynooth University, Ireland *
dara.macconville.2018@mumail.ie

Software verification is an important tool in establishing the reliability of critical systems. One potential area of application is in the field of robotics, as robots take on more tasks in both day-to-day areas and highly specialised domains. Robots are usually given a plan to follow, if there are errors in this plan the robot will not perform reliably. The capability to check plans for errors in advance could prevent this. Python is a popular programming language in the robotics domain, through the use of the Robot Operating System (ROS) and various other libraries. Python's Turtle package provides a mobile agent, which we formally model here using Communicating Sequential Processes (CSP). Our interactive toolchain CSP2Turtle with CSP model and Python components, enables Turtle plans to be verified in CSP before being executed in Python. This means that certain classes of errors can be avoided, and provides a starting point for more detailed verification of Turtle programs and more complex robotic systems. We illustrate our approach with examples of robot navigation and obstacle avoidance in a 2D grid-world.

1 Introduction

Robotics is a large and complex field, and autonomous robots are often designed to perform essential tasks. When it comes to such hybrid safety-critical systems, they demand a high level of confidence in their design and specifications. Beyond software testing, the highest level of assurance comes from formal verification, and many different approaches have been taken to apply this to robotic systems [5].

Robots are typically given a plan to follow in the form of instructions ("move forward 10m, turn 90°") or a goal to accomplish ("reach point A"). If this plan asks for something impossible because it does not align with the physical reality of the robot's environment, or if it asks for something logically contradictory, then this plan is incorrect. This would cause the robot to have incorrect or unpredictable behaviour. Testing a plan's correctness may be difficult and expensive in both time and money. Simulations can be time consuming to run, and real world tests may be risky and infeasible. Checking the plan against the robot's specifications and relevant properties of its environment during the design phase would avoid producing incorrect plans, and avoid the overheads associated with a design loop of producing a plan, running it on the robot, and seeing how the robot performs.

In this paper we present a model of Python's Turtle package in Communicating Sequential Processes (CSP) and show how this can be used to check that a plan for the turtle's behaviour is valid. We assume the plan is supplied by the user as planning itself encompasses a broad range of activities which are outside the scope of this paper. We also develop and describe a Python-based tool chain that takes a plan, performs checks on the model, and generates the corresponding Python Turtle code that performs the validated plan.

*This work has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number SFI 18/CRT/6049

R.C. Cardoso, A. Ferrando, F. Papacchini,
M. Askarpour, L.A. Dennis (Eds.): Second
Workshop on Agents and Robots for reliable
Engineered Autonomy (AREA'22).
EPTCS 362, 2022, pp. 15–22, doi:10.4204/EPTCS.362.4

In the remainder of this section, we provide an overview of both Turtle and CSP. Next, Sect. 2 describes our approach to modelling and the associated toolchain that was leveraged for verification and code synthesis. We illustrate our approach via examples and evaluate it in Sect. 3. Finally, Sect. 4 concludes and identifies future research directions.

1.1 Turtle

Turtle is an in-built Python package[1] centred around controlling an agent (the "turtle") on a 2D plane (the "canvas"). It is based on the Logo programming language, which has been used to program physical turtle robots[2], and provides a Logo-like set of commands via Python methods. These produce a graphical display of the turtle following the plan of these commands, tracing a line from its path. These features make Turtle useful for modelling and testing simple robotics plans and simulating the result.

The turtle has navigation commands for movement (`forward(), backward()`) and changing its direction (`left(), right()`); arguments passed to these functions indicate the distance it should move and the angle it should turn, respectively. The turtle also has commands controlling a "pen" that draws as the turtle moves along its canvas but only if it is in the down position (`penup(), pendown()`). These six commands are the focus of our modelling approach, described in Sect. 2.1.

Only a certain amount of the canvas is visible to the user, but the turtle can move and draw outside this. There are other cosmetic options such as pen colour, pen width, turtle shape, etc.; and additional commands, including leaving a stamp at the turtle's current location, and the ability to start 'recording' the turtles movements and draw a polygon along its path when it's finished moving.

1.2 Communicating Sequential Processes

CSP is a formal language for modelling concurrent systems [3]. The two fundamental units that CSP provides are processes and events. Events are communications over a channel, and a channel's declaration determines if its events take parameters. Parametrised channels are not used in this model, so none of the events take parameters either. A specification's behaviour is defined by processes, which themselves are sequences of events.

Events can be combined using various operators, and CSP provides some in-built processes. Table 1 summarises the CSP operators and the two in-built processes ($Skip, Stop$) that we use in constructing the model. The most essential operator to understand is $a \rightarrow P$, which defines the process that receives event a then after that acts like the process P. From this it is possible to perform actions such as recursively defining a process. The other key operators used here are running two process in parallel with no requirements to act in synch on events ($|||$), and allowing a process to choose between two or more options as to its next action ($\Box$).

The other key concept from CSP employed in this paper is that of a *trace*. The trace of a process is a sequence of the events that have happened throughout its lifetime. In our model, these events are designed to be in close correspondence to the functions that would be called by the turtle process in Python, to allow direct translation.

A specification written in CSP can have assertions made about it which are then checked by an appropriate model checker, here the command line tools from the Failures-Divergences Refinement checker (FDR) [2], are used. FDR supports machine-readable CSP (CSP_M), which allows for defining

[1] Turtle package: `https://docs.python.org/3/library/turtle.html`

[2] Logo Turtle robot: `https://web.archive.org/web/20150131192445/https://el.media.mit.edu/logo-foundation/logo/turtle.html`

Action	Syntax	Description
Skip	*Skip*	The process that immediately terminates
Stop	*Stop*	The process that accepts no events and thus deadlocks
Simple Prefix	$a \rightarrow P$	Communicate event a, then act like process p
External Choice	$P \Box Q$	Offer a choice between two processes P and Q
Interleaving	$P \mid\mid\mid Q$	Processes P and Q run in parallel with no synchronisation
Hide	$P \setminus A$	The process P runs normally but if any event from set A is performed it is hidden from the trace

Table 1: Summary of the CSP operators that are used in this paper.

CSP Event	*fd*	*bk*	*lt*	*rt*	*pu*	*pd*
Turtle command	`forward()`	`backward()`	`left()`	`right()`	`penup()`	`pendown()`

Table 2: The CSP events that we use in our model and their corresponding Turtle commands.

parametrised processes in a functional, Haskell-like way. These assertions are made about the possible traces of the turtle process, in CSP this is known as *trace refinement*. A process P it said to *trace-refine* another process Q if every (finite) trace of P is also one of Q[6].

2 Modelling and Implementation

2.1 Modelling Approach

We modelled a simplified version of Turtle in CSP, while still capturing the package's core ideas and functionality. The turtle is modelled as a CSP process, and the commands are CSP events. The events in our model, and their corresponding turtle commands, are detailed in Table 2. The plans we talk about in this paper are comprised of these events. The first six are abbreviated names of `forward`, `backward`, `penup`, `pendown`, `left`, `right` which were chosen both for brevity and as they are aliases for those methods in Turtle. We use an additional event *goal* to mark the location the user has specified in their plan as being the turtle's destination.

The key simplifications that were made were to restrict the possible directions of the turtle to just the four cardinal directions, and to limit movement to just one unit at a time. This means the turtle inhabits a grid-world, where its location will always be described by integer co-ordinates. The decision was also made to bound the size of the world that the turtle inhabits inside of CSP to some user defined constants, and not have it reside in an unbounded world.

These adjustments were made partially for technical reasons to do with our CSP implementation. To validate properties of a process, CSP has to examine all of its states. This obviously makes infinite state processes impossible to check, which necessitated the bounding of the turtle's grid-world size. Restrictions on movement and turning were made because CSP does not natively allow for handling floats, so it would be difficult to place it in a location that was not an integer grid. Despite these limitations, CSP still functions as a valuable tool for modelling and examining the properties of the turtle agent, because it allowed the core concepts of movement and pen actions to be formalised.

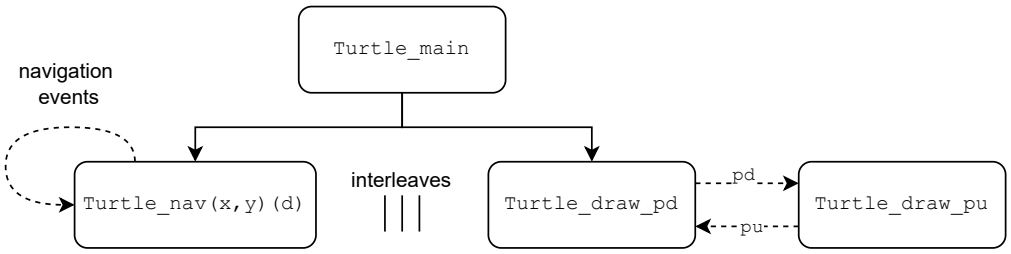

Figure 1: Architecture of the CSP model. Each box represents a process. The solid line represents *Turtle_main* starting the other processes, which are interleaved (|||). The dashed arrows (⇢) indicate events. *Turtle_nav* responds to the navigation events, which are omitted for brevity.

2.2 CSP Model Architecture

CSP supports specifying complex systems as an interleaving of smaller processes, each specifying one function. We used this feature in the architecture of our model, as shown in Fig. 1. The `Turtle_main` process is comprised of the `Turtle_nav` and `Turtle_draw_pd` processes running interleaved, it passes the model's initial parameters to each of these more specialised processes. `Turtle_main` is the process we directly make assertions about, the modular design of composing it out of smaller specialised processes makes it more easily extensible. The navigation and drawing functionalities of the turtle can be easily separated in this instance as they run independently of each other and do not need to synchronise on any events. This independence of processes is why the interleave command can be used. If synchronisation were needed, CSP has a parallel operator that takes two processes and a set of events which they must act in synch on.

`Turtle_nav` reacts to the movement and rotation events (*fd*, *bk*, *lt*, and *rt*), and updates the turtle's location (x, y) and direction (d) that it is currently facing. To handle these different cases `Turtle_nav` uses a different specialised process for each direction. Our turtle specification contains knowledge of the world's parameters (see Sect. 2.3) and `Turtle_nav` uses this internal knowledge to avoid moving beyond the boundaries or coming into contact with obstacles.

The `Turtle_draw_pd` process only interacts with the `pu` event, before transitioning to being the opposite process that only waits for a `pd` event. We assume that the pen always start down, and thus start with the *pd* process, as this is the default behaviour in Turtle. This flip-flop system ensures that the trace of this system will always be balanced. Note that the Turtle package does allow the `pendown()` method to be called while the pen is already down, but as this has no effect it would always be redundant to have in a plan for a turtle. This system guarantees such redundancy cannot occur.

2.3 Python Toolchain

We contribute a prototype toolchain CSP2Turtle to support the use of CSP alongside the Turtle package [3]. This Python toolchain is the overall system in which the Turtle CSP model sits and as shown in Fig. 2 it is through this toolchain that information is passed in and out. CSP2Turtle gets world dimensions, a plan, a location in the world to be marked as a goal, and obstacle locations from the user, and synthesises them with the existing CSP model. It then executes FDR's command line program `refines` to automatically perform two checks on the model, and if a valid plan has been provided it generates and runs Turtle code,

[3]CSP2Turtle code and turtle package model: `https://doi.org/10.5281/zenodo.6671802`

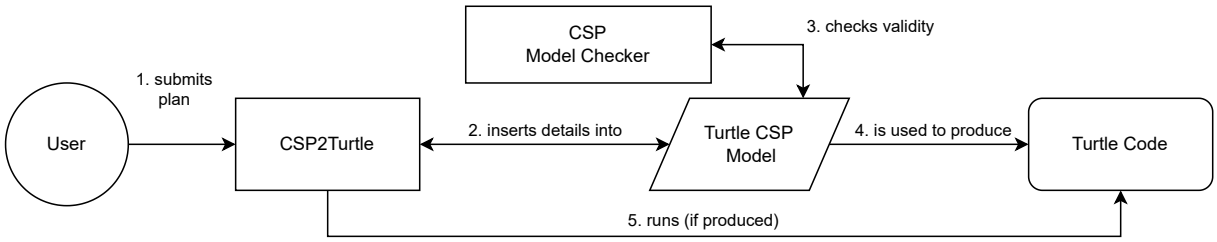

Figure 2: All the stages and components of the toolchain, showing the flow and order of operations, and how the user interacts with it. The arrows indicate how components acts on or produce others. Circle represents the user, the rectangles are running programs, the parallelogram is the model, and the rounded rectangle is source code produced by the programs.

giving live graphical output.

The first check is if the plan is valid for the grid-world's size and potential obstacles. It does this by seeing if the plan corresponds to a possible trace of the system with:

```
assert Turtle_main(0, 0)(E) :[has trace] <plan, as, trace>.
```

Note that `Turtle_main` takes its arguments in two pairs of parenthesis. This allows for it to be partially applied to just the first arguments if necessary, providing programming flexibility. The second check is if the goal location provided by the user is reachable from the turtle's starting location $(0,0)$. The turtle always starts at $(0,0)$ for consistency, this can be visualised as the bottom left corner of a grid extending up and to the right. This is a check on the environmental situation rather than the turtle, as it is only concerned with the location of the goal, and potential obstacles between that and the turtle's starting position. It does not attempt to reach the goal with the trace used in the first assertion. This check is performed with a trace-refines assertion:

```
assert Turtle_main(0, 0)(E) \ nav_events [T= goalpoint.
```

This assertion asks if another process *goalpoint* trace refines the `Turtle_main` process. This goal process has the form `goalpoint = goal -> STOP` and so only has one possible trace which is a singular goal event. The *goal* event can only occur in `Turtle_main` after it has entered the user defined goal state. The $\setminus$ *nav_events* hides the set of navigation events from the trace, meaning the trace will only contain either the goal event or nothing. This allows for easier checking as we are not concerned with how the turtle reaches the goal, only if it is possible to. The obstacles that may prevent the turtle from reaching its goal are defined by telling the process to do nothing but *Skip* if in the obstacle location. The results of these checks, pass or fail, are displayed to the user.

If the plan given is a valid trace of the system, a corresponding turtle program will be produced, saved, and ran, with the graphical output being visible to the user. If the trace is invalid, then no program is produced. The next section illustrates our approach via example, including code snippets and screenshots from our toolchain.

3 Example Usage

Our workflow begins with the user initiating the CSP2Turtle prompt. From here they can input the dimensions of the turtle's world by giving a width and height, and mark certain locations as being "obstacles", which are impassable to the turtle. They can then provide a plan for the turtle in the form of a trace, which FDR will check against the parameters of the world to verify if it is a possible plan to

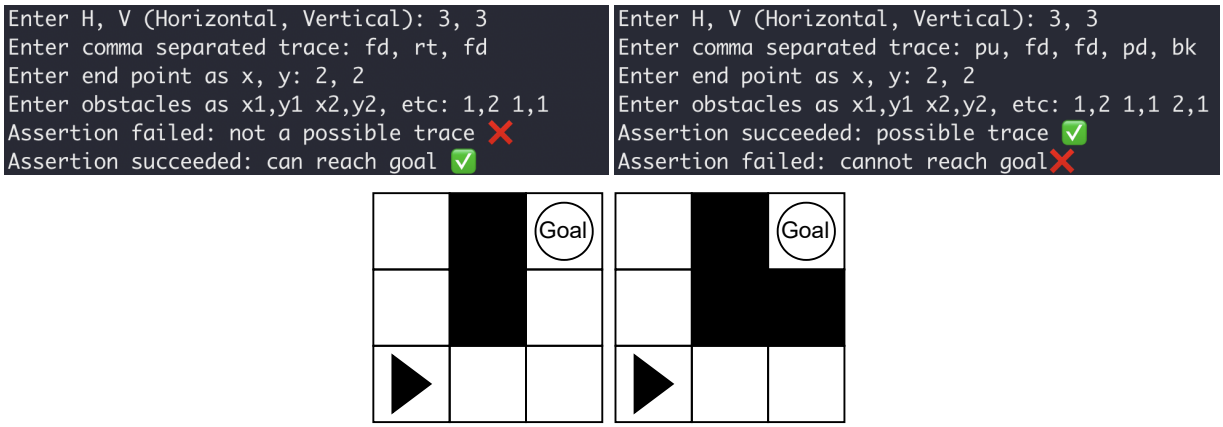

Figure 3: Two usage examples, with corresponding maps of their worlds beneath them. Notice the addition of $(2,1)$ in the second example blocks the turtle's (►) access to the goal and how the trace in the first example cannot fit in the boundaries.

execute. A certain location in the world can be labelled as a goal and FDR will deduce if it is reachable from the turtle's starting state. The starting location is hard-coded to be $(0,0)$ and facing East as this is the default in Turtle, but this could be changed if needed. The screenshots in Fig. 3 depict examples of CSP2Turtle at runtime and an example map of the turtles' worlds.

The left screenshot shows that the given trace is a not a possible one of that Turtle system. Turning right then attempting to move forward would move the turtle outside of the confines of its 3x3 world. Independently of that, the turtle is still able to reach the goal state of (2, 2). The locations (1, 2) and (1, 1) are blocked as obstacles, but FDR finds that it can still move through (2, 1) to reach the goal.

The right screenshot shows a valid trace. The turtle does not attempt to move beyond the bounds of the world, and performs pen ups and pen downs correctly. One additional obstacle is added at location (2, 1) and now the turtle can no longer reach the goal from its starting position, FDR correctly finding that all entrances are now blocked. All of this checking takes place before the plan is run, and incorrect plans cannot be run. This provides a useful 'offline' checking mechanism that prevents it being necessary to run a full simulation.

CSP lends itself well to modelling the core Turtle components. Processes and events map naturally onto the turtle and its methods. This made the initial modelling decisions easy and provided a good foothold to explore different possibilities for the model. They were quick to prototype and easy to test and probe in FDR.

Adding the Python layer allows for quick and interactive testing of CSP plans. It also introduces another layer of the system that itself is not verified, which is a weakness when trying to have a highly robust system. This added layer of complexity enabled tasks that would not be possible in pure CSP , but made it harder to discern if errors were in the Python program or the CSP model.

Other aspects of CSP also influenced design decisions. The lack of floats meant we decided to not allow arbitrary movement, instead restricting ourselves to the integer grid. The way CSP handles variables meant that recording the path of the turtle to incorporate its `begin_poly` method would require a much more roundabout approach. In addition, implementing the plan as a trace is very easy and natural in CSP but other methods would be possible too, such as letting the plan take the form of a process.

4 Conclusions and Future Work

Robotics can be used to provide safety-critical systems, and such systems require a software verification approach to their design and implementation to ensure that they behave correctly. CSP2Turtle enables quick checks on the validity or possibility of certain actions of the robotic agent turtle given certain movement constraints. FDR could be used not just to verify the plans but also to find them. This work focused only on user submitted plans but extensions like this are certainly an avenue we would like to explore in the future.

Future work includes restructuring the design to capture Turtle's `begin_poly()` and `end_poly()` functions, which facilitate the creation of arbitrary polygons by recording the turtle's movements. This type of memory-storage driven action is less amenable to implementation in CSP but is still possible. The scalability of these methods and the core model to larger scale systems could be tested.

Methods can be developed to alter the topology or geometry of the world. This could involve changing the topology to objects such a loop or torus, or the geometry to non-rectangular shapes. An adaptation in a similar vein would be increasing the turtle's range of motion. Allowing diagonal movement, and a variable move distance would make the model less abstracted This may require using parametrised channels or allowing for more options but from a fixed range (30°, 45°, etc). The plans themselves can be made more dynamic too, as mentioned in Sect. 3 a process could be supplied by the user, better representing the types of choices a robot may have in its plans.

The focus of this paper was on a single turtle agent, but the library has the capability for multiple independent agents, which is an area where CSP has been successfully applied before [4]. The CSP variant tock CSP could be used to introduce the notion of discrete time into the model, this doesn't correspond to the turtle software but could be useful especially with multiple agents. Another approach (noted in Sect. 3) is verifying the Python components themselves. A way around this would be to use Nagini [1], a library for verifying Python code, that can be written inside Python itself instead of having to model it in another external tool. Nagini could be used to make and prove assertions about the correctness of CSP2Turtle, for instance if it inserts the plan into the model correctly, which would help avoid errors that can occur from adding more programming complexity.

References

[1] Marco Eilers & Peter Müller (2018): *Nagini: A Static Verifier for Python*. In Hana Chockler & Georg Weissenbacher, editors: *Computer Aided Verification*, 10981, Springer International Publishing, Cham, pp. 596–603, doi:10.1007/978-3-319-96145-3˙33. Available at `http://link.springer.com/10.1007/978-3-319-96145-3_33`.

[2] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov & Andrew Roscoe (2014): *FDR3 – A Modern Model Checker for CSP*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS* 8413, Springer, pp. 187–201, doi:10.1007/978-3-642-54862-8_13.

[3] Charles Antony Richard Hoare (1978): *Communicating sequential processes*. *Communications of the ACM* 21(8), pp. 666–677, doi:10.1145/359576.359585.

[4] Matt Luckcuck & Rafael C. Cardoso (2022): *Formal Verification of a Map Merging Protocol in the Multi-agent Programming Contest*. In Natasha Alechina, Matteo Baldoni & Brian Logan, editors: *Engineering Multi-Agent Systems*, Springer, Cham, pp. 198–217, doi:10.1007/978-3-030-97457-2˙12. Available at `https://arxiv.org/abs/2106.04512`.

[5] Matt Luckcuck, Marie Farrell, Louise A Dennis, Clare Dixon & Michael Fisher (2019): *Formal specification and verification of autonomous robotic systems: A survey*. *ACM Computing Surveys (CSUR)* 52(5), pp. 1–41, doi:10.1145/3342355. Available at `https://dl.acm.org/doi/10.1145/3342355`.

[6] A.W. Roscoe: *Understanding Concurrent Systems*. Texts in Computer Science, Springer London, doi:10.1007/978-1-84882-258-0. Available at `http://link.springer.com/10.1007/978-1-84882-258-0`.