ISSN 2754-1118 (Online)
ISSN 2753-6599 (Print)

Robust AI Planning

Colin O'Halloran, Paul Whittaker and Nick Tudor

D-RisQ Ltd., Malvern, UK

Abstract

Autonomous systems are intended to be used to achieve tasks that enable humans to be removed from dull, dirty, or dangerous situations. In order to be able to do this autonomously, there are 2 main constraints: Reliability and Safety. An element of reliability is persistence, in other words, the ability to continue to achieve a task safely, and is the focus of this paper. In order to do this without human intervention, there has to be an ability to provide the system with a plan of how to achieve a task. AI Planning is a computer science discipline that has been used in various scenarios where some degree of autonomy is required for the computer system. Its use relies upon inputs that describe a task that needs to be achieved, a language that can be used to describe the task, and a machine called a "solver". The output is typically a plan that may be used to try to achieve the task. The types of tasks can range from route planning to cyber-attack, and the output is a plan in the form of a program to be followed. However, there are a number of issues with the usual approaches. AI planning tools typically require the use of elastic resources, memory and processing power, so use of the cloud helps with the computational effort and hence is an off-line activity. Further, the use of heuristics means that AI planning tools might not even return an answer. Replanning when conditions change in the environment or in the system therefore presents issues for embedded real-time systems. This paper will outline the issues with typical approaches, and present a potential solution for generating domain-specific planners implemented on hardware. The dedicated planner enables real-time replanning and the whole approach is placed in conjunction with safety assurance.

1 Introduction

Planning is a central and hard computer science problem that is essential in the development of autonomous systems. Many existing planning solutions require a controlled environment in order to function correctly and reliably. However, there are situations where adaptive autonomous systems are required to run for a long period of time and cope with uncertain events during the deployment without the possibility of human intervention but still exhibit explainable behaviour, even if it is undesirable. Such planning can be specifically applied, for example, in the use of autonomous systems for tasks normally completed by humans. For example, reconnaissance of a building in an urban environment, or survey in an underwater environment, where communications might be contested or indeed absent. Such activities without the ability for human intervention will require autonomous planning and re-planning in real time in order to cope with a dynamic environment.

© D-RisQ Ltd 2025.

For example, consider a decision-making system suitable for an Autonomous Underwater Vehicle (AUV) designed to stay underwater for up to 6 months with very limited communication with the outside world. The AUV is expected to carry out survey missions on its own and report details of its surveillance at semi-regular intervals. During the mission, the AUV may encounter underwater currents, deep ocean terrain, fishing boats, objects and places of interest, etc., each of which may affect its ability to achieve its goals. The AUV must be able to decide which goals to pursue when such uncertain events occur and plan tasks to achieve the goals in an agile manner. In the face of uncertain events in execution, planning becomes an even harder problem. In this case, the agent's goal may be affected and thus both selecting a new goal and re-planning are necessary. Currently, this would require either communication back to a base station, with the problems of latency, reliability, and detection of the vehicle; or a large amount of on-board processing using standard processors that would constrain the use of autonomy.

2 Problem

A human is able replan how to achieve a task as events unfold within the limits of training, knowledge, skills, and experience as well as capability. For an autonomous system, we need to be able to provide a plan of what it is required to achieve. While this is possible, the plan is typically somewhat inflexible and not able to adapt to changing events ether on-board or in the surrounding environment, often needing a human to intervene (see Figure 1). This means that the utility of an autonomous system is limited by the communication system to a human supervisor as well as their ability to undertake a replan. The technologies used for planning such tasks typically require specialist knowledge. The tools used are not easy to access, and embedding them into autonomous vehicles for use in real time replanning is often impractical. Finally, the tools provide a solution that may not be guaranteed to actually achieve the task in the most optimal, efficient, fast, sensible manner or even, in some cases, at all. We therefore need to be able to produce a plan that can be updated in real time with a high degree of integrity such that we know that any plan produced will achieve the task in the most efficient way.

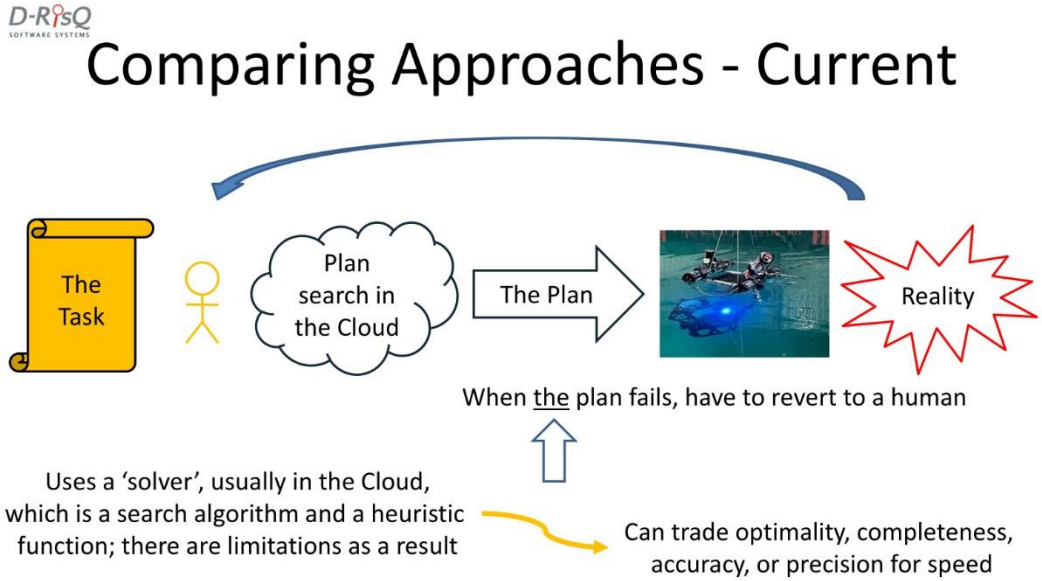


Figure 1 ~ Fragility of Current Planning

3 Aim

3.1 General

The aim is to equip an autonomous system with a dependable onboard real-time Artificial Intelligence (AI) planning system that is resilient to changes in the environment and failures in the system. The AI system combines symbolic reasoning, responding in real time, to select new goals and re-plan, with the potential to use machine learning to synthesize the symbolic reasoning rules for an application domain such as underwater or an urban environment.

The planning problem for autonomous systems is about the decision-making performed by intelligent agents when trying to achieve some goal. It involves choosing a sequence of actions that will transform the state of the world, step by step, so that it will safely satisfy the goal. The world is typically viewed to consist of atomic facts (state variables), and actions make some facts true and some facts false. The atomic facts are created by sensed events, and actions lead to commands, delegated to subsystems to execute, which are recorded as facts. Facts can be falsified by commands superseding previous commands, or sensor information overriding previous sensor readings.

The Planning Domain Definition Language (PDDL) (Ghallab et al. 1998) is a *de facto* standard output for symbolic AI planning languages. Usually, planning is solved by heuristic search, but this approach cannot respond in real time and does not confer a sufficient level of trust. The correctness, safety, and security issues of autonomous systems is particularly important in mission-critical use cases such as the AUV example.

3.2 Background

The core philosophy built into planning description languages (Ghallab et al. 1998), and as a result many planners, is that the techniques applied to solving a problem should know nothing about the problem they are solving. That is the techniques that are applied to solve, for example, a logistics problem, could also be applied to a manufacturing problem. Unlike other fields of AI, such as Machine Learning, the approach is not dependent on there being pre-existing information on the problem, or an example path to a solution. All that is needed is the starting state, the actions we can apply, and the desired end state. Often these three pieces of information are readily available, purely from the nature inherent to the problems a planner is trying to solve. Simply put, you do not start planning until you know what you are trying to achieve. This includes an understanding of what is "safe". For example, it perhaps is not safe to continue to fly when navigation is compromised, but what to do about this is part of the planning problem, in that something in the environment has changed requiring a replan.

For example, if the construction of large stadium is to be planned, in order for the stadium to be finished it is known what needs to be completed, it is known that there is currently no stadium built, and it is known what the different actions that can be performed to change the state closer to the desired position (build frames, install pipework, raise flags, etc.). Knowledge of how previous stadiums were built is not needed, or what orderings were used (for example building foundations then frames, etc.). This lack of need for knowledge means planning problems that have not been previously solved can be applied, with the caveat that there is some implicit knowledge if an action is based upon a current state.

If it is assumed that previous solutions to a planning problem are not as good as they could be, applying a technique like machine learning would essentially train to solve problems sub-optimally. Therefore, an approach is needed which is methodical, objective, and avoids learning from prior approaches which are sub-optimal. Although there is a lot more to solving a plan than a state space search, the general (but not universal) unifying factor of different planners is that they perform some kind of "state space" search.

Suppose that from the initial state various new states can be accessed, by applying actions; the result is a tree of possible states, with the initial state being the root. From that root we perform a search, attempting to find a state as close as possible to the root, which has all of the facts needed in order to satisfy the goal. If a system following a pre-determined plan encounters something that invalidates the assumptions underlying the plan, then replanning is required to accommodate changed circumstances. The aim of the project was to produce reliable and efficient implementations of planning and re-planning given a proposed planning description.

3.3 Approach

The innovation that we have developed is the exploitation of the massively parallel architecture of a Field-Programmable Gate Array (FPGA) to make a PDDL set of rules as concurrent as possible. The maximum concurrency enables all possible solutions to be generated within a real time constraint. Put more simply, all plans that are possible (constrained by what is sensed) will be explored concurrently. The size of current FPGAs means that tens of thousands of ground instances of rules can be run concurrently, which means that a discretized planning model of the world will be sufficient for dynamic planning and re-planning in non-pathological cases. To achieve the same level of concurrency would require tens of thousands of cores communicating with each other. If threading was used on a core, then that would suffer hugely in performance because it would reduce to sequential execution of interleaved threads. Such a parallel execution strategy, based on facts supplied by sensors, generate a selected sequence of steps for an autonomous vehicle to perform.

3.4 Assurance with Regard to Safety

The assurance of safety is addressed by exploiting a formal mathematical representation of PDDL, developed in the process algebra of Communicating Sequential Processes (CSP) (Roscoe 1997). This has the potential to be used to formally-verify all possible behaviours using the FDR model checker (Gibson-Robinson et al. 2016). Such verification would occur before deployment on the FPGA to provide guarantees of what the system will do and, more importantly, that it will not do anything that it is not supposed to do. What is needed is an assurable route to automatically implementing PDDL that can demonstrably re-plan in real time in an uncertain and dynamic environment within the space and time limitations of an FPGA. However, whilst this is a useful addition to safety, it is insufficient on its own to assure the safe behaviour of the vehicle as it may develop a plan that violates some safety in some way; more on this in Section 9.

4 A Formalisation of a Planning Domain Definition Language

4.1 Planning Domain Definition Language Concepts

PDDL is one of the few languages designed for the purpose of creating a standard for AI Planning. It was developed in 1998 and was introduced at the International Conference on Automated Planning and Scheduling, with improvements and extensions being built into it over the years. The different versions of PDDL with their slightly different syntaxes are not the principal interest of this project. Instead, the underlying concepts that can be formalised for on-the-fly replanning based upon sensed input information are those of interest. In particular, the formalisation should be amenable to formal verification, so that when re-planning is required it is reliable under all specified circumstances. No guarantees should be expected if the expected environment is replaced with a different environment with different implicit assumptions about how an autonomous system will sense and move through that environment. The fewer assumptions made about the environment, the greater the applicability to different domains.

To avoid any misunderstandings the key concepts that are formalised are:

- Objects: Things in the world that interest us
- Predicates: Properties of objects that we are interested in; i.e. true, false or undetermined
- Initial State: The state of the world that we start in
- Goal Specification: Things that we want to achieve
- Actions: Ways of changing the state of the world

A simple example of planning for the Towers of Hanoi puzzle is used to explain the formalization before the approach is utilised for planning for an autonomous system to visit a series of waypoints.

4.2 The Towers of Hanoi Puzzle

4.2.1 Overview

The Towers of Hanoi is a mathematical puzzle consisting of three pegs and a number of disks of various diameters, which can slide onto any peg. The puzzle begins with the disks stacked on one peg in order of decreasing size, the smallest at the top as illustrated in Figure 2.

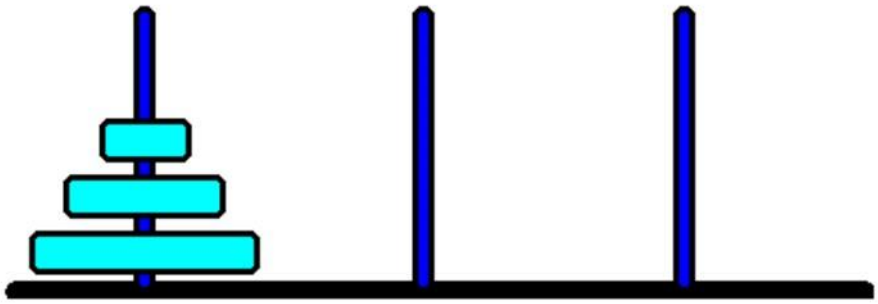


Figure 2 ~ Initial State for the Towers of Hanoi

The objective of the puzzle is to move the entire stack to the furthest peg on the right, obeying the following rules:

- Only one disc may be moved at a time.
- Each move consists of taking the upper disc from one of the stacks and placing it on top of another stack or on an empty peg.
- No disc may be placed on top of a disc that is smaller than it.

With three discs, the puzzle can be solved in seven moves. The minimal number of moves required to solve a Tower of Hanoi puzzle is $2n - 1$, where n is the number of discs.

To represent the puzzle with objects, predicates, an initial state, goal and the actions that can be performed, a functional language is used to define datatypes, functions, sets and sequences. The functional language is similar to a restricted version of Haskell (Haskell 2014) but is defined as part of the machine-readable version of CSP (Roscoe 1997).

4.2.2 Objects

To distinguish discs by their size a datatype definition is made.

```
datatype Size = Large | Medium | Small
```

A function can now be simply defined to map elements of the datatype to an integer value.

```
SizeOf(Large) = 3

SizeOf(Medium) = 2

SizeOf(Small) = 1
```

The legal configurations of discs on a peg can now be defined as the set *Discs*.

```
Discs = {< >, <d>, <d0, d1>, <d2, d3, d4>
        | d <- Size, d0 <- Size, d1 <- Size, d2 <- Size,
          d3 <- Size, d4 <- Size,
          SizeOf(d0) < SizeOf(d1), SizeOf(d2) < SizeOf(d3),
          SizeOf(d3) < SizeOf(d4)
        }
```

This set comprehension states that a peg can have no discs on it (the empty sequence, denoted < >), or it can have one disc (the singleton sequence <*d*>, such that *d* is drawn from the datatype *Size*). The set comprehension of *Discs* continues with two and three discs being permitted to be on a single peg, but their size (defined by the function *SizeOf*) must respect the ordering that no disk may be placed on top of a disk that is smaller than it. The ordering is defined by the predicates:

```
SizeOf(d0) < SizeOf(d1), SizeOf(d2) < SizeOf(d3), SizeOf(d3) < SizeOf(d4)
```

where the separating comma is an implicit conjunction.

The objects of concern now form part of statements that can be made about the state of the puzzle, and are defined by the following datatype.

```
datatype statements = start | On_peg.{1,2,3}.Discs | Not_On_peg.{1,2,3}.Discs
```

In this case there are three pegs, each of which can have none, one, two or three discs. For example *On_peg.1.*<*Large*> denotes the statement that the first peg only has the large disc on it. A dual of this statement is that *Not_On_peg.2.*<*Large*> denotes the state of the

puzzle where the large disc on its own is not on the middle peg. In this formulation the start statement indicates the initialisation of the puzzle; more will be discussed about the start statement a little later.

4.2.3 Predicates

In this example predicates are essentially equivalent to Statements (except for the start statement), in general there might be predicates that link actions together that are not direct Statements of the state of the system. The set of predicates is formalised as:

```
predicates = {s | s <- statements, s != start}
```

A function *NOT* must be defined to link predicates to their negation.

```
NOT(On_peg.i.discs) = Not_On_peg.i.discs
```

```
NOT(Not_On_peg.i.discs) = On_peg.i.discs
```

Note that the power of pattern matching in the functional language makes the definition of *NOT* simple and concise.

It is important to have statements and their negation so that, if a sensed or deduced fact is volatile (i.e. it can be true at some point and at a later point be false), then if the negation is asserted in a conclusion part of an action (e.g. *NOT(On_peg.2.Large)*, which becomes *Not_On_peg.2.Large*) then the statement such as *On_peg.2.Large* is "forgotten". A negation might also be used as a precondition for a rule but does not lead it to being "forgotten"; in the Towers of Hanoi example, a negation is not needed in a precondition.

In the Towers of Hanoi example all predicates can be "forgotten" and are thus labelled as volatile in the underlying semantic model.

```
volatile = predicates
```

A subtlety about the formal semantics of a planning system is whether the world in which the objects exist is Open or Closed. Observations may be incomplete. In an open world, unobserved facts are deemed to be unknown. If a fact *K* or its negation *NOT(K)* are not known then *K* is unobserved. For example, moving a disc from peg 1 to peg 3 only uses facts about the discs on peg 1 and peg 3. No fact about peg 2 is needed and could be unknown. In the Towers of Hanoi puzzle, everything about the state of the puzzle is known (or observed) from the initial state onwards.

4.2.4 Initial State

The initial state of the puzzle is shown in Figure 2, and is denoted by the set *init*.

```
init = { On_peg.1.<Small,Medium,Large>, On_peg.2.<>, On_peg.3.<> }
```

The set *ini*t defines the initial state as all the discs are on peg 1 in the correct size order, and there are no discs on pegs 2 or 3; further the sequence encodes the fact that the Small disc is on top of the Medium disc, which in turn is on top of the Large disc.

4.2.5 Goal

The desired goal that a plan should achieve is simply stated by the set Goal.

```
Goal = {On_peg.3.<Small,Medium,Large>}
```

The goal of any plan is to place all the discs on peg 3 in a legal configuration.

4.2.6 Actions

The actions for the puzzle are also expressed in the function language as two parameterised sets denoting legal moves from *peg i* to *peg j*. In the first case it is the set of all possible moves of a disc from *peg i* to *peg j*, when *peg j* already has one or more discs on it. The set, parameterised by the numbers *i* and *j*, is that of a pair of sets. The form of the pair is that of *IF precondition THEN result of action* and is denoted by the pair (*precondition, result of action*). The precondition is the set of predicates that must be true before the predicates that denote the result of the action are deemed to be true. The precondition for the set of actions denoted by:

```
Move_disc_from_peg_i_to_non_empty_peg_j(i,j)
```

is the set of predicates, the first of which is that there is at least one disc, *d1*, on the *peg i* and is denoted by:

```
On_peg.i.<d1>^stacki
```

where *<d1>* is the singleton sequence, ^ is the operator for sequence concatenation, *stacki* denotes some sequence of discs (which could be the empty sequence), and *d1* is constrained to be smaller than any disc in *stacki*. The second predicate in the precondition is that there is at least one disc, *d2*, on the *peg j* and is denoted by:

```
On_peg.j.<d2>^stackj
```

Both of these predicates are constrained by the conditions that disc *d1* must be smaller than disc *d2* (denoted by *SizeOf(d1) < SizeOf(d2*) in the set comprehension) and that the two stacks of discs share no discs (i.e. there are no duplicate discs on the pegs). The only other constraint is that the stack of discs must be only those that are drawn from the set of legal configurations of discs, denoted by *<d1>^stacki <- Discs, <d2>^stackj <- Discs* (where *x <- X*, means *x* is a member of the set *X*).

The result of any of the specified actions where the precondition is satisfied is that the disc *d1* is removed from *peg i*, *On_peg.i.stacki*; *d1* is put on top of disc *d2* on *peg* j, *On_peg.j.<d1,d2>^stackj*; and the predicates in the precondition are negated

```
NOT(On_peg.i.<d1>^stacki),NOT(On_peg.j.<d2>^stackj)
```

Negating the precondition like this means that the previous knowledge about the position of the discs is "forgotten" (technically, it is a non-monotonic deduction system), which means that the exact positioning of discs on pegs is tracked.

The parameterised definition for legal moves for the first set of actions from *peg i* to a *peg j* that already has discs on it is:

```
Move_disc_from_peg_i_to_non_empty_peg_j(i,j) =
{
  ( {On_peg.i.<d1>^stacki, On_peg.j.<d2>^stackj},
    {On_peg.i.stacki, On_peg.j.<d1,d2>^stackj, NOT(On_peg.i.<d1>^stacki),
     NOT(On_peg.j.<d2>^stackj)
    }
  ) | <d1>^stacki <- Discs, <d2>^stackj <- Discs, SizeOf(d1) < SizeOf(d2),
      empty(inter(set(<d1>^stacki),set(<d2>^stackj)))
}
```

The second set of actions covers the case of moving a disc from a *peg i* to a *peg j* that has no discs on it. The precondition for any action is similar to the definition of the previous set, except that the stack of discs on *peg j* is specified as the empty sequence, *On_peg.j.<>*. The predicates describing the result of any action on the set is similar to the previous set: the disc *d1* is removed from *peg i* and the precondition predicates are negated; the second predicate *On_peg.j.<d1>* records the fact that the disc *d1* has been moved to the previously empty *peg j*. The set of actions of moving a disc from a *peg i* to a *peg j* with no discs on it is specified by the following parameterised set.

```
Move_disc_from_peg_i_to_empty_peg_j(i,j) =
{
  ({On_peg.i.<d1>^stacki, On_peg.j.<>},
   {On_peg.i.stacki, On_peg.j.<d1>, NOT(On_peg.i.<d1>^stacki), NOT(On_peg.j.<>)}
  )
 | <d1>^stacki <- Discs
}
```

Finally, an initialisation action is required

```
Initialisation = { ({start}, init)  }
```

The two parameterised sets can be combined together with the singleton set of the initialisation action to specify a set of deductions about the state of the puzzle.

```
deductions =
Union({ Initialisation, Move_disc_from_peg_i_to_non_empty_peg_j(i,j),
                        Move_disc_from_peg_i_to_empty_peg_j(i,j)
        | i <- {1,2,3}, j <- {1,2,3}, i != j
      })
```

The functional language provides a powerful means of expressing actions that are compiled to ground terms that are used by the model checker FDR (Gibson-Robinson et al. 2016). An example of a ground term in the set deductions is the pair

```
({On_peg.2.<Small, Medium, Large>, On_peg.3.<>},
{On_peg.2.<Medium, Large>, On_peg.3.<Small>, Not_On_peg.2.<Small, Medium, Large>,
Not_On_peg.3.<>})
```

that describes the state when all the discs are on peg 2 and none of the destination peg 3. The number of ground terms in deductions for the puzzle description is 79.

4.3 The Semantic Model

The details of the abstract machine formalised in CSP that "execute" the data in the set *deductions* will not be described due to space limitations. Instead, the deduction system that is specified in CSP, and will be implemented on an FPGA, will be broadly described.

A deduction system is a formal (mathematical) setup of reasoning. In order to describe a deductive system, a (formal) language system must first be in place, consisting of (well-formed) formulas, i.e. strings of symbols constructed according to some prescribed syntax. The previous section has defined what the well-formed formulas are for the Towers of Hanoi puzzle, and the PDDL approach provides well-formed formulas for each problem domain it is used for.

With the language in place, reasoning, from a formal point of view, is just derivation of a formula, called a conclusion; from a given set of formulas, called premises; via a set of formulas, called axioms and rules, also called rules of inference. The set of formulas that act as premises are the preconditions for actions. The results of actions are the conclusions. The action is the rule of inference and actions with no precondition are axioms.

The central task of a deduction system is the construction of deductions, which informally, is the application of inference rules to assumptions (or axioms, if any) to arrive at a conclusion in a finite number of steps. The specified goal (such as all the discs on peg 3) is the desired conclusion through the application of a finite number of inference rules. A record of the inference rules that lead to the goal conclusion is a step-by-step plan to achieve the goal.

A deduction system is often non-deterministic; the conclusion of an inference rule can enable more than one subsequent inference rule to be used. A cascade of possible inference rules can be explored to reach the desired goal, if it is reachable. The abstract machine that acts as an inference engine is therefore naturally described by a collection of Communicating Sequential Processes. An implementation in VHDL[1] of the inference engine on an FPGA exploits parallelism to simultaneously explore all possible sequences of inference rules that can lead to the desired conclusion — the goal wanted from a plan, ideally the shortest plan. In this manner a dedicated PDDL solver for just the Towers of Hanoi can be implemented on an FPGA and the same approach can be used to create a dedicated solver for a specified problem domain.

4.4 Generating a Plan and Checking for Consistency

The FDR model checker automatically verifies whether one CSP process refines another, i.e. that all the observable behaviours of one process are a subset of the observable behaviours of the other. If there is a behaviour that is outside those permitted by the specification process, then FDR reports a counterexample. For the purposes of generating plans, the counterexample is a sequence of events each of which denote either an inference rule used to deduce the desired goal, or house-keeping events such as introducing the initial state or forgetting previously established predicates.

The inference engine acting over input inference rules has all its observable behaviour hidden except for a communication that the desired goal has been achieved. Thus, if the goal cannot be achieved then no communication is observed and the refinement check FDR carries out is reported as successful. If the goal is achieved then the report of its achievement is communicated and a counterexample generated because it does not conform to a process that exhibits no observable behaviour. This may seem paradoxical, but it is a standard technique to achieve efficient compilation and state space exploration for FDR. The reported counterexample (or, in general, counterexamples, if FDR is instructed to report more than one) can be inspected or automatically extracted by other tools for processing or just pretty printing.

As discussed, the counterexample consists of the sequence of inference rules, or house-keeping. Due to FDR's breadth-first search approach, any counterexample reported will be the shortest sequence of deductions that can be made to deduce the goal; hence the shortest plan of actions is reported.

[1] VHDL is the Very High-Speed Integrated Circuit Hardware Description Language, IEEE 1076-2019.

The example of the Towers of Hanoi puzzle demonstrates that predicates in the premise will be negated in the conclusion of an inference rule. Negation can easily lead to inconsistencies, such as a disc is on a particular peg and the same disc is not on the same peg. Testing shows the presence of such bugs but does not show their absence, therefore testing an implementation of an inference engine on an FPGA falls short of the mathematical verification that FDR carries out. For example, race conditions implicit in hardware concurrency might lead to a successful test that would later fail due to environmental factors, such as the order in which inputs happen to arrive.

To check that no inconsistencies can be accidentally introduced through inference rules being slightly wrong, an automated check can be performed. The consistency check is to check whether the goal *P* and *NOT(P)* can be achieved at the same time, for every predicate *P* in the problem description. In this case no report of the goal being achieved is desirable. If an inconsistency exists the counterexample will list the sequence of inference rules that lead to the contradiction, which enables diagnosis and correction.

5 Planning and Re-planning Visits to Waypoints

5.1 Preamble

The re-planning described for the Towers of Hanoi is naïve because once a plan is set, there is not a lot that can interfere with a plan to cause a replan, (the above shows the principles that every move is a "re-evaluation" of the plan), but it is the approach to formalisation and verification that can be applied to more sophisticated re-planning approaches that is important, rather that the particular re-planning approach. A more realistic planning and re-planning approach would be just as amenable to our formalisation and verification approach. We also need to extend the principles beyond the closed system of the Towers of Hanoi to enable interface to the outside world, as would be needed in an autonomous vehicle.

5.2 A Waypoint Planning Problem Description

Following the same pattern as Sub-section 4.1, the following are used:

- Objects: Things in the world that interest us
- Predicates: Properties of objects that we are interested in; i.e. true, false or undetermined
- Initial State: The state of the world that we start in
- Goal Specification: Things that we want to achieve
- Actions: Ways of changing the state of the world

The planner is interacting with the rest of the system, receiving abstracted sensor information, generating a sequence of steps in a plan based upon this and then informing, for example, an autopilot. In the simple example of visiting waypoints, sensor information would inform the planner whether the predicate for arriving at a waypoint was true. The planner would then generate the steps for actions at that waypoint and subsequently direct the autopilot to navigate to the next waypoint.

5.3 Results

Table 1, below, summarises the results of running the inference engine through FDR to generate a sequence of steps that achieve the goal of visiting all accessible waypoints. The compilation time to generate the labelled transition system for 3 up to 8 waypoints representing all possible composition of inference rules, as well as the housekeeping to maintain consistency, is a fraction of a second. On an Intel 2.7 GHz core laptop, with 4 cores, the exploration time to generate a counterexample ranges from 2.4 seconds (for 3 waypoints) up to just under 12 minutes (for 8 waypoints).

Table 1 ~ Statistics for Checking 3 to 8 Waypoint Plans

Number of Waypoints (Excluding waypoint 0)	**Length of counterexample**	**Number of inference rules**	**Number of predicates**	**Compilation Time** (seconds)	**Exploration Time** (seconds)	**Number of States Visited**	**Number of Transitions Visited**
3	17	28	66	0.10	2.40	348	1,198
4	22	36	91	0.12	3.43	821	3,315
5	27	44	120	0.15	3.81	5,567	27,928
6	32	52	153	0.18	7.01	73,939	443,100
7	37	60	190	0.22	34.39	937,502	6,383,286
8	42	68	231	0.29	703.23	13,148,243	100,362,760

5.4 Limitations

5.4.1 General

The re-planning strategy is fairly simplistic, but the purpose of this project is not to propose new planning algorithms. The aim of the work is to produce reliable and efficient implementations given a proposed planning description that can be translated from a PDDL representation.

The main limitation of the approach is the challenge of planning problems with inherent combinatorial complexity. An example of such a planning problem is that of the travelling salesman problem (Applegate et al. 2006). The travelling salesman problem asks the following question: "*Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?*" It is an NP-hard problem in combinatorial optimization and the determining the shortest route would be a planning problem of which city to visit in what order.

The time to complete a check given in Sub-section 5.3 shows that determining the shortest plan to visit all accessible waypoints has a combinatorial element. The Towers of Hanoi puzzle has a similar element as the number of discs increases. The problem addressed is the combinatorial explosion of a search space for verifying that the dedicated solver will always return a plan for any inputs from the environment. Once this has been done then the FPGA solver only has to generate a plan for the particular set of inputs encountered. There are two mitigations to this challenge of the combinatorial explosion of a search space.

5.4.2 Divide and Conquer

The first is to split the problem up into sub-problems that can be solved. For example, the waypoints to be visited are already in an order that has been previously determined and pre-loaded, therefore the first N waypoints can be addressed with the additional check that the system can reach the N^{th} waypoint, which will provide the initial state for the next sequence of waypoints to be visited. In general, this is an example of a compositional method of verification and the composition into sub-problems will depend upon the class of problem that is to be addressed. Once a composition method has been identified, it can be mechanistically applied to all problems in that class.

5.4.3 Subtle Sledgehammer

The second mitigation is that, once a process representation has been compiled (which for the waypoints problem is a fraction of a second), then the state space exploration can be split up over whatever core processors are available. Experiments using Amazon's EC2 elastic computing resource has demonstrated a linear speed up of checks as the number of core processors increases linearly. For many practical circumstances employing such elastic computing would be sufficient. However, access by autonomous systems to cloud resources in real-time would be difficult to implement — impossible where there are no communications.

6 Implementation

6.1 VHDL Implementation Required

6.1.1 Preamble

The transfer of the PDDL to VHDL was developed and largely automated. That enabled us to then use commercial tools to configure the FPGA. The FPGA gives instructions to the system to act upon: move a piece in the Towers of Hanoi, follow a heading in a vehicle, etc. However, it proved to not be so straightforward.

6.1.2 Abstraction

We decided that we needed to be able to abstract away many of the implementation details. This removes a great deal of "clutter" from the code, to the extent that significant functionality can often be presented in well under a screen of code. This makes it easier to inspect and review the implementation, allowing us to have increased confidence in its correctness. It also significantly reduces the risk of implementation mistakes, such as "off-by-one errors", that might have been introduced if we had to build much of that logic ourselves.

In practice, it allowed us to specify the algorithm's behaviour without needing to deal with the details of its implementation. It also allowed us to exclude parts of our algorithm from consideration when we were tracking down the cause of some unexpected behaviours during development: it could be seen that the algorithm was clearly doing what it was intended to, and had no room to hide implementation errors.

6.1.3 Goal of the Implementation

For this phase of the work, the FPGA hardware performs an exploration of the Towers of Hanoi problem space, and determines whether it is possible to reach the intended goal state from the initial starting state. The system is initialised so that only one state is active. This state is the one that represents all of the discs being stacked on the starting peg. The system is then allowed to evolve as directed by the collection of state transition rules, as supplied by the Haskell processing phase. Finally, the system is monitored to see whether the target state, which represents all of the discs being stacked on the final peg, is ever reached.

6.1.4 Combining the Parts of the Implementation

By taking the algorithm implementation functions, and adding in the static data structures, we produced the complete source code for our implementation. This source code is fed to a commercially available VHDL auto-generation tool, which produces a directory with a small number of VHDL source files. Those source files implement hardware that has the same behaviour as the code. We selected the Xilinx FPGA and the Vivado development environment. Vivado allows the uploading of the implementation to the FPGA by carrying out the "synthesis" process largely automatically. It also allows simulation and inspection of the code and is useful for our purposes, as it gives a report on the number of FPGA hardware resources that were used in the process of constructing the implementation.

An overview of the process is in Figure 3.

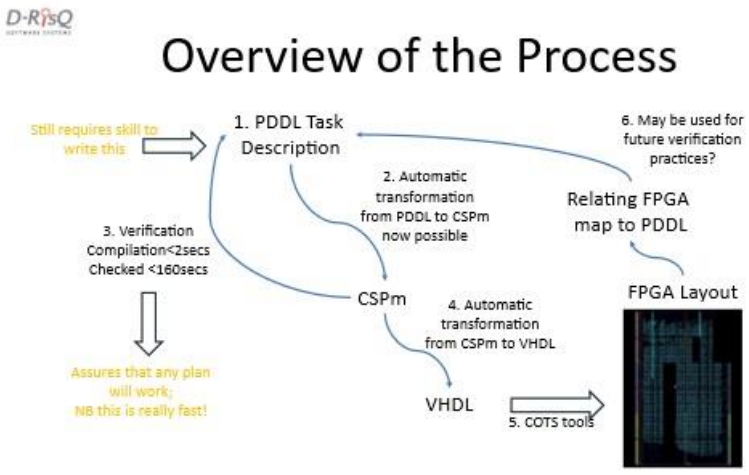


Figure 3 ~ Transformation from PDDL to FPGA

6.2 Test Results

The following observations are made on the results (see Table 2):

- As expected, the numbers of states and transitions observed grew exponentially with the number of discs being modelled[2].
- Hardware resource usage, however, only grew linearly with the numbers of states and transitions. Although the Towers of Hanoi problem data does not allow us to differentiate between these two factors (since both grow at the same rate), it is expected that both contributed: each state should require a register to store whether or not it is currently active, and each transition should require logic cells to capture it.
- By far the most significant factor once we reach the larger problem sizes is the time required by the FPGA-related tooling. There appears to be a steep (greater than linear) growth function compared to the state/transitions count for both the VHDL generation and the Vivado synthesis stages. This is something for future investigation.
- The third-party FPGA tooling was the clear bottleneck in our benchmark testing. The most obvious limit was the combined time they took — over an hour and a half for the largest test, compared to under 20 seconds for the remaining processing steps, and about twenty to thirty minutes for the manual steps required to join the stages together. It is possible that larger problems might need runtimes of a day or longer in order to process them.

Table 2 ~ Statistics for Checking up to 6 Discs on an FPGA Implementation

Problem Size (discs)	States	Transitions	VHDL (Sec)	Synthesis (sec)	Resource Usage
1	3	6	3.0	67	0.01%
2	9	24	3.0	67	0.02%
3	27	78	1.2	71	0.04%
4	81	240	7.2	78	0.1%
5	243	726	104	212	0.4%
6	729	2184	3649	2068	1.1%

7 Blind Alleys

In the development of the capability, we explored a number of potential technology solutions which were shown to be blind alleys. Two are briefly described below.

7.1 Handel-C

The aim was to exploit apparent parallelism inherent in the use of PDDL, and reflected in CSP, and then to implement and preserve this by use of an FPGA. Handel-C is a high-level programming language, which is most commonly used in the programming of

[2] It is well-known that the number of moves required to solve the problem exhibits exponential growth, being *(2^n)-1* moves when *n* discs are present: see https://www.britannica.com/topic/Tower-of-Hanoi. A similar result (but with a growth rate of approximately *3^n*) applies to the number of reachable states in the graph. This can be seen by observing that adding a single (larger) disc triples the number of valid states: it can be added to the bottom of any of the pegs of any previously valid state, and in doing so will produce a distinct new state in each case. Since every state (apart from a small number of corner cases) has three valid transitions available, a similar result holds for exponential growth in the number of graph transitions we expect to see in the graph.

FPGAs. It is a rich subset of C, with non-standard extensions to control hardware instantiation, with an emphasis on the parallel model of computation shared with CSP. In order to facilitate a way to describe parallel behaviour, some of the CSP keywords are used in Handel-C, along with the general file structure of Occam, which itself was built on the CSP process algebra. While this was initially a useful avenue to explore, and an automatic translation from CSP to Handel-C was developed, the compiler was expensive and support was frugal.

7.2 3rd Party FPGA Tooling

We tried a number of different approaches, and they all have their limitations. Some were not easy to use without specialist knowledge, and only basic functionality could be exploited. Once we tried to undertake more useful processes, including communications within and off board, we experienced considerable difficulty in making the implementation work in the way we had expected. Diagnostics and on-line fora were limited in usefulness. Nevertheless, Vivado did what we expected within normal constraints.

8 Exploitation

8.1 Autonomous Vehicles

This work was specifically developed to assist in the field of autonomous vehicles, although there may well be other uses. The major issue is that when a plan loaded onto an uncrewed vehicle (e.g. air/sea/subsea) fails for whatever reason, the plan needs to be refreshed or the task the vehicle is intended to execute will have to halt. Currently, replanning requires communication as a human has to undertake the task which typically may require the use of cloud resources (see Figure 1). Clearly, in environments such as underwater, this is unrealistic. Hence the ability to react to events on-board as well as in the sensed environment within real-time is needed in order to provide a persistent service. Without such a facility such services are not scalable and this restricts their usefulness. We need to be able to extend from the closed systems of the Towers of Hanoi to enable communications to the vehicle and to incorporate data from the changing external environment.

8.2 Real-time System Use Overview

Accordingly, we developed the FPGA to enable communication via the Robot Operating System (ROS) framework. Figure 4 shows the process and how the FPGA, the vehicle systems, and the environment interact. Initially, the environment will provide data to the vehicle control system and through the ROS Node to the FPGA which will then take account of the status change to initiate a (re)plan, provide a fresh plan which is then transmitted from the ROS node to the vehicle control system. This process is continuous, so will provide a replan per cycle. Meanwhile, in parallel, the vehicle control system will execute the plan by issuing appropriate instructions to the vehicle; this too is a continuous, parallel process. As the environment changes (which can also include the on-board environment), the FPGA updates the plan.

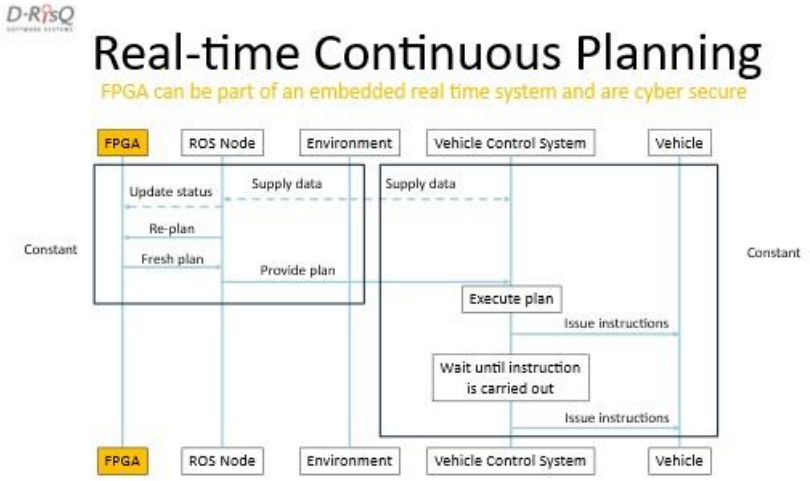


Figure 4 ~ Visualisation of Real-time Replanning

8.3 Use Case Exemplar

8.3.1 Cable Fatigue Monitoring

Offshore windfarms have a significant problem in that export cables from a wind turbine pylon are failing much earlier than expected; sometimes five times faster. The export cable exits the concrete plinth underwater, but above the seafloor, and describes a curve eventually resting on the seafloor some distance from the plinth where it may be buried in a shallow trench; see Figure 5. This means that the loop above the seafloor is subject to tide and currents that flex the cable. This causes metal fatigue in the current-carrying cable, which causes resistance, which exacerbates the problem. Furthermore, the movement of the cable causes de-trenching of the cable which makes matters much worse. While monitors can be attached to the cable, these have to be read, typically by a diver, and this can only be undertaken in benign conditions; typically, in less than 6 months of a year. The number of cables to be inspected means that access is often only once per year, a frequency which does not provide good data upon which to base maintenance decisions. The cable can easily fail before maintenance can even be planned. This therefore is an ideal use case for a persistent at sea, autonomous capability.

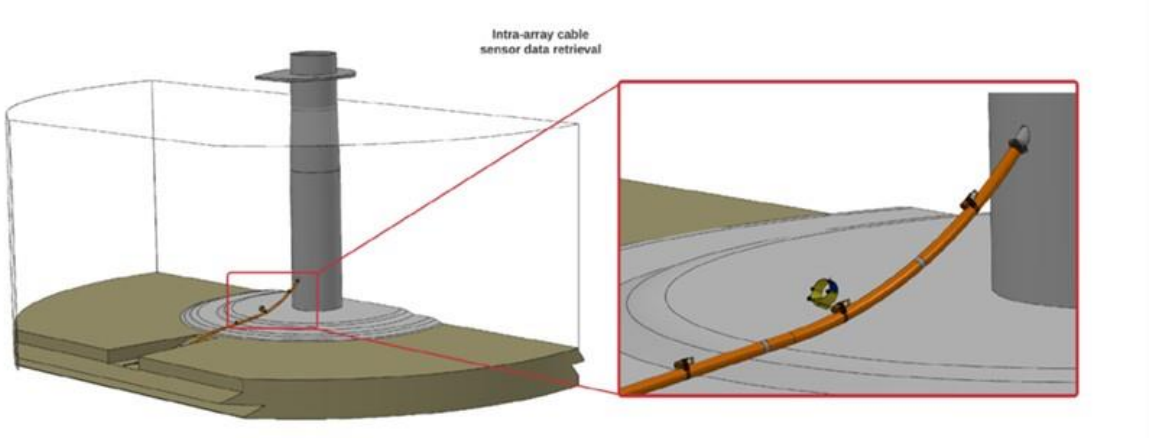


Figure 5 ~ Export Cable Motion Sensors Data Gathering

8.3.2 Persistent Monitoring

The work in this paper was exploited in simulated demonstrations with an underwater vehicle, as partially funded by Innovate UK on the SEAMless and Demeter projects completed in September 2023 (Innovate UK 2023a) (innovate UK 2023b). The Demeter project was specifically targeted at this use case, and an experimental vehicle was developed. The vehicle itself was novel in that it required very little, if any, power to manoeuvre on the surface, using just wave action. This means that the vehicle can be available continuously. To submerge it would require batteries, which will have been recharged whilst travelling between pylons. This gave a limitation on the time available, and became one of the factors in the plan. A series of waypoints once submerged were also part of the task with an end-goal of reading data off the device attached to the cable. The planning part of the project was successfully demonstrated using 8 waypoints (with surfacing actions to re-charge a battery) using an FPGA communicating with the rest of the system through ROS.

8.3.3 Future Uses

The intention is to further develop the technology to make it more accessible. At present, considerable skill is required to use PDDL and while a lot of the transformation to VHDL is automated, there remains a need for human intervention at a number of points in the process that could be made easier or automated. The plan is to develop the technology further, address some scaling issues and deploy on real autonomous platforms in air, sea and land as well as seek other use cases in the non-autonomy domain.

9 Safety Architecture

The work typically undertaken within D-RisQ is focussed on software system safety and security. The techniques described in this paper exploit formal methods and give a degree of assurance, but we believe they are insufficient in themselves, within the planning approach described, to provide sufficient safety assurance. In all our autonomous systems work in air, land and sea we typically produce a "safety cage" that is the foundation of the safety argument for the assurance of vehicle behaviour. This approach is not novel as it has been written into aerospace standards since at least 1992. At Sub-section 2.4.3 of RTCA/DO-178C[3] it outlines that any untrusted function needs to have a safety monitor at the required integrity level; AI, in general, cannot be verified so it is an untrusted function. Our approach, while probably more robust than most, still requires a safety cage.

10 Conclusion

This work has shown that it is theoretically possible to undertake replanning in real time without the need for human intervention. It also showed that it is likely to be scalable, though with a few issues still to be addressed. The approach also needs to be made more

[3] Also available as EUROCAE Document ED-12C (note that this sub-section was also in the previous B version of this *de facto* standard).

accessible, so that typical users may have access to the technology without the need for specialist knowledge or significant training. While we have demonstrated that the technique works, and has been shown in simulations, a real-world demonstration has yet to be completed. While safety to an extent has been addressed in this work, there will remain the requirement for a high-integrity *independent* safety monitor.

Correspondence Address

D-RisQ Ltd., Malvern Hills Science Park, Geraldine Road, Malvern WR14 3LR

Acknowledgments

This work was partially funded by grants provided by Innovate UK, and furthered by DSTL.

References

Applegate D. L, Bixby R. M, Chvátal V, and Cook W. J. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics, Princeton University Press, New Jersey.

Ghallab M, Knoblock C, Wilkins D, Barrett A, Christianson D, Friedman M, Kwok C, Golden K, Penberthy S, Smith D, Sun Y, and Weld D. (1998). *PDDL — The Planning Domain Definition Language*. Yale Center for Computational Vision and Control, Tech Report CVC TR-98-003/DCS TR-1165. https://planning.wiki/_citedpapers/pddl1998.pdf. Accessed 29th May 2025.

Gibson-Robinson T, Armstrong P, Boulgakov A, and Roscoe A. W. (2016). *FDR3: a parallel refinement checker for CSP*. International Journal on Software Tools for Technology Transfer, 18(2), pp149-167. Springer Nature, Berlin.

Haskell. (2014). *Haskell—Declarative, statically typed code*. Haskell.org website https://www.haskell.org/. Accessed 29th May 2025.

Innovate UK. (2023a). *Demeter*. Innovate UK partially-funded project. https://gtr.ukri.org/projects?ref=10012225. Accessed 29th May 2025.

Innovate UK. (2023b). *Subsea Enhanced Autonomous Mapping (SEAMless)*. Innovate UK partially-funded project. https://gtr.ukri.org/projects?ref=10011940. Accessed 29th May 2025.

Roscoe A. W. (1997). *The Theory And Practice Of Concurrency*. Prentice-hall International Series in Computer Science, Prentice Hall, New Jersey.

RTCA/DO-178C. (2012). *Software Considerations in Airborne Systems and Equipment Certification*. RTCA Inc., Washington DC.