Automated Synthesis of Magic Tricks

Rory Fotheringham

4th Year Project Report
Artificial Intelligence
School of Informatics
University of Edinburgh

2023

Abstract

This paper reproduces a system described by the paper ‘On ∃∀∃! Solving: A Case Strudy on Automated Synthesis of Magic Card Tricks’. Which automatically synthesises a magic trick named ‘The Baby Hummer’. I identify areas of the system which make the system hard to reproduce and give clear, formal descriptions of the these areas, discussing the relevant engineering choices.

The paper mentions as further work the desire to encode a constraint which prevents deterministic looping subsequences in the tricks. I discuss this problem in detail and come up with a solution. I report the synthesised tricks and demonstrate the effectiveness of preventing these deterministic looping subsequences with a user study which compares reported audience engagement for each trick.

Research Ethics Approval

This project obtained approval from the Informatics Research Ethics committee.
Ethics application number: 697929
Date when approval was obtained: 2023-02-27
The participants' information sheet and a consent form are included in the appendix.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Rory Fotheringham*)

Acknowledgements

I would first like to thank Elizabeth Polgreen, my project supervisor, for their useful feedback, patience and willingness to help me throughout this project. I'm grateful for Philip Wadler who contributed their valuable perspective at a turning point in this project. I would like to thank everyone who gave their time to participate in this project's user study.

I'd like to thank Mosque Monday and it's members for the steadiness of community and friendship it has given me througout my studies. I am grateful for Julia Fotheringham - my mother and seasoned researcher - who's love, radical kindness and academic wisdom has helped me to engage with this project with a playful and curious spirit. I would finally like to acknowledge award winning director and partner Giulia Grillo, who has shared with me her enduring care and love without which a magic trick is nothing but an assignment to a series of vectors.

Table of Contents

Chapter 1

Introduction

Program synthesis is the task of automatically generating a program from a given logical specification. Due to the large solution space of even simple programs, this problem is very hard with general formula solvers [1]. A common way of solving this problem is using Component Based Synthesis [8] where a program is created as a combination of components from a finite library. This technique has many applications and a paper from 2016 showed that it can be used to synthesise card magic tricks [10]. These magic tricks can be used as classroom exercises, as entertainment or as outreach resources.

The original paper is very short and gives only sparse details about both the formulation and implementation of the system. This makes it difficult to reproduce. In this paper I build a component based synthesis system which reproduces the results from the original paper. I document aspects of the implementation and formulation of the problem that were omitted from the original paper and discuss any important engineering decisions I made which makes the system introduced by [10] easier to reproduce.

I define a deterministic loop as a section of a trick which always loops and never changes the state of the deck. I make the argument that this property leads to non-interesting magic tricks. I extend the methods used in the original paper to introduce constraints which prevent the generation of magic tricks which contain a deterministic loop. I provide details on the formulation of this problem as well as a discussion of its implementation challenges.

I evaluate and compare my implementation against the results from the original paper. I also demonstrate the effectiveness of the new deterministic loop constraints with a user study.

1.1 Background

1.1.1 CEGIS

In 2008 Solar-Lezama published *'Sketching for Program Synthesis'* [15] which introduced a novel way to synthesise programs. The method takes a 'sketch' which is a partial program of the desired program as input and generates the complete program.

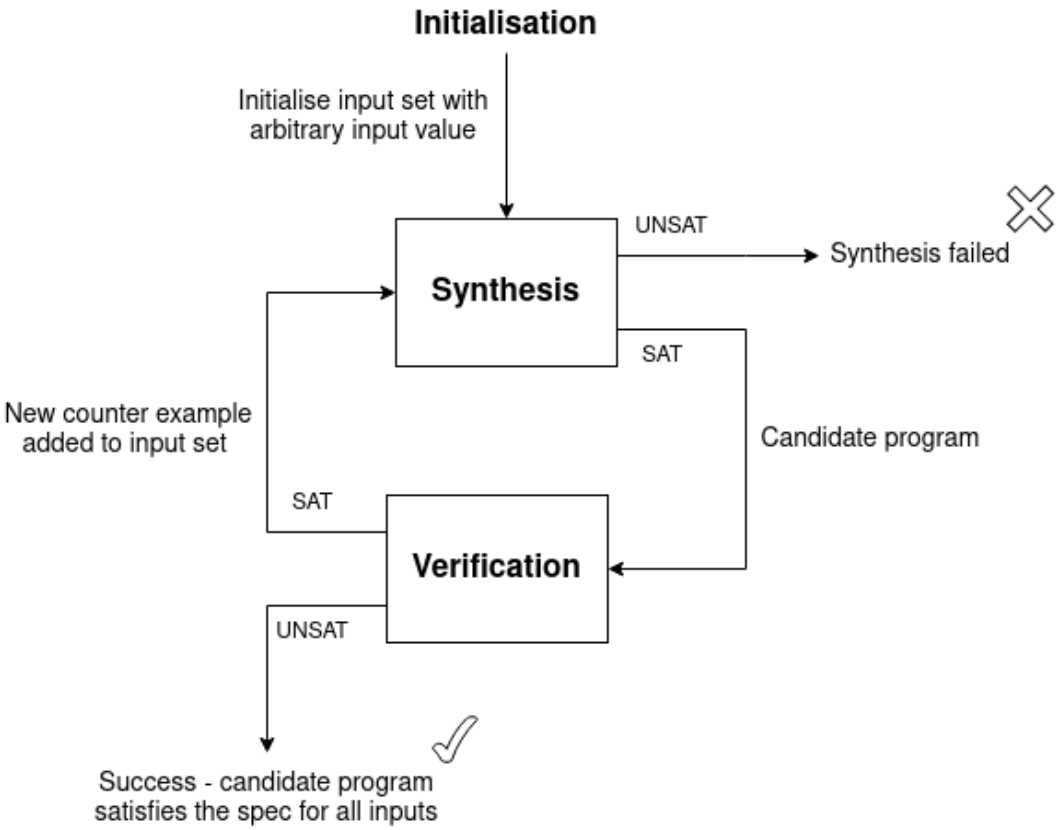

Figure 1.1: Figure showing the the CEGIS process

To generate this complete program, Solar-Lezama developed an iterative synthesis algorithm named Counter-Example Guided Iterative Synthesis (CEGIS). Although the aims of the 'Sketching' paper are not directly aligned with my project, the CEGIS algorithm which it introduces is at the core of my project. It is central to producing the magic tricks. The CEGIS process at a high level takes an input of the form:

$$\exists P. \forall I : \phi(P, I)$$

One can think of P as a program, I as a vector of inputs and $\phi()$ as a logical specification of the desired behaviour of the program with respect to its inputs. Finding some P that satisfies this formula would give us an automatically synthesised program purely from a high level specification. Solving this problem through explicit enumeration would be very expensive but the iterative nature of CEGIS makes large problem spaces feasible by exploiting certain aspects of the domain. The process is illustrated in Figure 1.1 and can be broken into three steps:

1. **Initialisation**

   A set S is initialised with one arbitrarily chosen input value I_0.

2. **Synthesis**

   Finding a P that works over all possible inputs I is very expensive, the algorithm

instead finds a program which satisfies the specification over only the inputs in the input set S. This is called the synthesis step of the algorithm, a solver finds a solution to the much more tractable formula:

$$\exists P \forall I \in S . \phi(P, I)$$

There are many ways to find the solution to this formula. One common way is to enumerate over the terms in a grammar [14]. I discuss this in more depth in Section 1.1.3. If this query is unsatisfiable then, of course, the original formulation would be unsatisfiable. If it is satisfiable then the algorithm is one step closer to a program that satisfies the specification. This P is saved (call it P') as a candidate program and the algorithm moves onto the verification stage.

3. **Verification**

   To check whether the candidate program satisfies the specification, rather than checking whether each input in the space acts accordingly on the program, a solver can be queried to search for an input I on which P' will transgress the specification. This can be formulated as so:

$$\exists I . \neg \phi(P', I)$$

   If no such I can be found then the loop is broken and the candidate program is returned as a program which is valid over all inputs, otherwise, the resulting input (call it I') is a counter-example. This counterexample will be used to further guide the synthesis of our desired program. I' is added to the set S and the algorithm returns to the synthesis stage (or breaks if some timeout threshold is reached).

1.1.1.1 CEGIS summary

The inputs added to the set S over which the candidate program is being synthesised act as constraints to the logical formula in the synthesis stage. A feature of the constraints being generated from counterexamples is that they will never be entirely redundant - it will always add some meaningful constraint since it covers a part of the search space that was not covered by the other inputs in S [16]

An intuition for why this algorithm is able to efficiently generate programs from only a small number of iterations can be found in the fact that most inputs to a program are ‘normal’, with only a handful of edge cases. A program that can handle one or two ‘normal’ inputs can usually handle the thousands of other ‘normal’ inputs.

This captures the heart of the CEGIS algorithm, though a simple function specification ϕ as is shown will not be sufficient for the purposes of this project. The formulation of further constraints will be explored in depth in the next section.

1.1.2 Component Based Synthesis

A 2010 paper Component-Based Synthesis Applied to Bitvector Programs (Gulwani et al)[8] builds on the work of Solar-Lezama. The synthesis algorithm used in this paper is an instantiation of the CEGIS process but deals with programs as a composition of components rather than as a partial 'sketch' as defined in Section 1.1.1. It also uses an SMT solver to solve a set of constraints in the synthesis stage instead of an enumerative approach which is more common. I introduce SMT solvers in Section 1.1.3. Although the synthesis of Bitvector programs seems unlinked to the synthesis of magic tricks, the power and scalability of the component-based synthesis technique is such that it can be applied to both domains.

This paper uses separate syntax, semantic and specification constraints in the CEGIS process rather than using one specification constraint ϕ as described in Section 1.1.1. This enables the synthesis and verification constraints to be more expressive. For example, the verification constraint is now formulated as

$$\exists I.\phi_{des}(P',I) \wedge \neg\phi_{spec}(P',I)$$

Where ϕ_{des} is a semantic constraint specifying the behaviour of the components and ϕ_{spec} is the specification on the final state of the program. This forces the counterexample found by the verifier to violate the specification with respect to the semantics of the program. This reduces the number of possible counterexamples and forces the counterexamples to cover a more meaningful area of the search space.

This paper also introduces syntax constraints, enforcing that the components are arranged such that the resulting assignment is a well-formed program. This constraint is conjoined to the synthesiser but not the verifier. This is because the verifier receives its components P' from the synthesiser. If the synthesiser produced P' under the syntax constraint then the verifier does not need to follow this constraint. Following this, the synthesis constraint is becomes

$$\exists P \forall I \in S.\phi_{des}(P,I) \wedge \phi_{syntax}(P) \wedge \phi_{spec}(P,I)$$

Where ϕ_{syntax} is the syntax constraint.

It is common in component based synthesis that each component in the library is only allowed to be used once [8] [9]. This reduces the number of possible combinations of components and makes it easier to control the output. To give the effect of a component being used multiple times, multiple copies of a component are added to the library.

1.1.3 The Z3 Solver

The CEGIS algorithm relies on a function which takes a logical formula and returns either UNSAT in which case the formulation is not satisfiable, or SAT in which case there is some satisfying assignment to the free variables (this satisfying assignment is also returned). This problem is known to be NP complete [11]. Fortunately, advances in

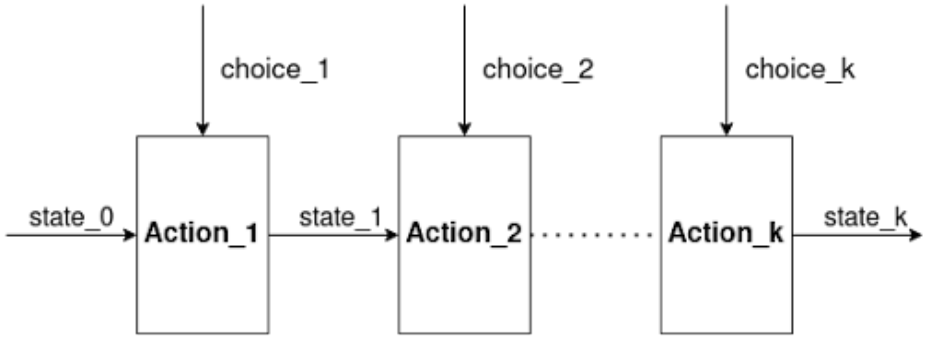

Figure 1.2: Figure showing a magic trick modelled as a sequence of actions with respect to non-deterministic audience choice. *adapted from [10]*

the field of Satisfiability Modulo Theory (SMT) solvers have produced tools which can (in many practical cases) find solutions to nontrivial problems in a reasonable amount of time [11][10]. SMT solvers apply this over theories such as integers or arrays allowing them to reason at a higher level. One such tool is the Z3 solver developed by Microsoft Research [5]. Many of the methods relied upon by Z3 are very sophisticated and beyond the scope of this project. I have chosen to use it as it is freely available, state of the art and there is evidence of its success in the context of other synthesis systems [8][7][10]

Though many aspects of Z3 will be treated as a black box, using it effectively requires some understanding of a query's representation. Z3 uses the Single Static Assignment (SSA) form in which variables are assigned one value and never change. Transitive variables, will have to be split into k variables numbered $var_1, var_2, ..., var_k$ where k is the length of the trick. Similarly, constraints on variables must be extended to constraints on every k variable with respect to the assigned value of the component. This will result in very long queries, the formulation of which I will automate.

1.1.4 Modelling magic tricks

The paper I will be reproducing in this project is a Case Study on Automated Synthesis of Magic Card Tricks [10]. This paper's most important contribution is the description of a magic trick as a $\exists\forall\exists!$ problem. This makes the observation that many card tricks can be summarised as a sequence of actions, some of which are non-deterministic (depending on random audience choice), which due to some underlying mathematical structure will always result in a predetermined final state. Figure 1.2 illustrates this model.

The original paper [10] denotes the following:

A library $\{A_1, ..., A_n\}$ is a set of actions. An action is a move in a magic trick, for example, turn the top card over or cut the deck. Since these actions change the state of the deck, each action has a corresponding state transformer function $T_i : S \times C \to S$, where C is a set of audience choices and S is a set of all possible states - a state being any configuration of the cards at a single timestep. This applies the action to a state and returns the resulting state.

An action can be deterministic or non-deterministic. For a deterministic action, the corresponding transformer function ignores the audience choice parameter C.

To allow for variable length tricks, actions $A_{n+1}...A_{n+k}$ are added to the library which we call `noop` actions. These are characterised by the fact that $T_{n+1}(s,c) = ... = T_{n+k}(s,c) = s$. In other words, they do not change the state.

A vector $i_1, i_2, ..., i_k$ corresponds to a sequence of actions. The value of each element of this vector is in the range $[0, n+k]$[1] and selects one of the actions. Let this vector be named `comps`

The vector $c_1, c_2, ...c_k$ where each $c_j \in C$ represents the sequence of non-deterministic audience choices. It is named `choices` . The vector $s_0, s_1, ..., s_k$ where $s_j \in S$ is a state of the trick.

Finally there is a parameterised state transformer T which uses an element of the `comps` vector i_t to select a corresponding state transformer T_{i_t} which is applied to the second and third arguments which are members of the set S and C respectively. For example, the function $T(i_t, s_t, c_t)$ selects $T_{i_t}(s_t, c_t)$ which returns the state resulting from performing action A_{i_t} on state s_t with audience choice c_t.

Note that the value k is the number of timesteps in the trick and also the number of `noop` actions in the library. The original paper represents both of these with the same symbol k and it is the same value. I have done this too to be consistent with the paper.

The paper also gives a very brief description of how these components can be manipulated to produce a magic trick though it is not a clear description. Providing a clear and detailed description of this is considered part of my project and so is not contained in the introduction section.

Observe how this model of magic tricks corresponds to the program synthesis described in Section 1.1.2; `comps` corresponds to a program P and `choices` corresponds to input I. For a given `comps` and `choices` , there is a unique corresponding `states` , this is constrained in the semantic constraint ϕ_{des} introduced in Section 1.1.2. The magic tricks paper [10] uses CEGIS with component based synthesis to find new versions of magic tricks for a given specification.

1.2 Contributions

My contributions in this paper are as follows:

- My reproduction of the paper gives a clear explanation and discussion of the methods used, providing the necessary formulae for the trick. Though the paper formulates the constraints of the magic trick synthesis problem broadly, none of the syntax nor semantic constraints, which are crucial to the system, are given at any more than a high level.

[1] The paper states that the values of i are in range $[0, n]$ though, to the best of my understanding, this would give no opportunity to select a `noop` action. My discussion and implementation of the system will treat $i \in [0, n+k]$

- I successfully prevent deterministic loops from appearing in the magic tricks. This is mentioned in the original paper as a direction for further work. I explore this problem and offer solutions which work in different search spaces. I discuss related implementation challenges caused by the solution and evaluate its results.
- I conduct a user study, performing tricks generated with different parameters and constraints to an audience. I compare audience ratings of the different tricks to evaluate the importance of the deterministic loops mentioned above.

Chapter 2

Synthesising magic tricks with component-based synthesis

The original magic trick synthesis paper gives only a very high level formulation of the the task of magic trick synthesis. There are many details left out of these formulations which make it hard to reproduce. I have reproduced the component-based magic trick synthesis system described by the paper. This chapter identifies important details which were omitted by the original paper and expands upon them, discussing and giving formulae where appropriate. I also describe how I implemented the system, explaining any design decisions I have made.

To allow me to go into sufficient detail, I will focus my analysis on one trick - specifically 'The Baby Hummer' (invented by Charles Hudson [6]) which is the first trick formulated in the paper.

In 'The Baby Hummer', an audience member selects a card at random and places it at the bottom of the deck (the original trick used a deck of 4 cards). All of the cards in the deck are facing downwards. At this point, the magician claims they can find the audience selected card by giving the audience member a set of instructions without looking at the deck. Some of these instructions involve moves which depend on random choices made by the audience member, this gives the audience member a sense of control. This sense of control is undermined when, despite the randomness of the audience member's choices, the audience selected card appears to be facing the opposite way from all of the other cards in the deck. The instructions given to the audience member are a combination of the actions described in Table 2.1

2.1 Notation

The original paper uses a mixture of monospaced fonts and traditional Latex characters in it's formulae. For the sake of consistency, I have done this too. I also include fragments of pseudocode to illustrate important parts of the implementation. To avoid confusion, I state how I use these below.

- `comps` , `states` and `choices` refer to the vectors as defined in Section 1.1.4.

Action	Description
`flip_2`	flip the top two cards in the deck
`turn_top`	flip the top card in the deck
`top_2_to_bottom`	place the top two cards on the bottom of the deck
`top_to_bottom`	place the top card on the bottom of the deck
`cut`	cut the deck at random position (audience chooses) and place the bottom packet on the top

Table 2.1: Table naming and describing the actions used in the Baby Hummer trick. Notice how the `cut` action depends on the choice of the audience.

- `depth` refers to the number of cards in the deck. It is used in both implementation and formulation contexts.
- I refer to the actions (or 'moves', see Section 2.4.3) in monospaced font, for example, `flip_2` or `cut`.
- I denote a logical formula which encodes the semantics of a move with the symbol ϕ with the name of the move in subscript. For example, the semantic formula for `cut` is $\phi_{\texttt{cut}}$.
- For long formulae, I will nest sub-formulae within. The names of these sub-formulae may also use monospaced font, for example, $\phi_{\texttt{cut\_a}}$. These are also abstract formulae and their function will always be explained.
- In this chapter, I define a function `shift()` in an abstract semantic formulation. I have used monospaced font purely for readability and its use does not imply that it refers to a concrete implementation.
- Whenever I illustrate a concrete implementation with pseudocode, I signpost it clearly. In this case all text will be in monospaced font. All of the relevant functions used are either apparent from the context or have clear explanations.

2.2 Representation

2.2.1 State space

The representation of a problem is important, particularly in SMT queries where the number of possible solutions grows extremely quickly relative to the number of variables [11]. In this section, I will explore options for representing the state of a trick.

When comparing the feasibility of problem representations, it is desirable to have some formal metric. It is hard to predict runtime of an SMT query. There is not a clear correlation between runtime and the number of variables. The difficultly of a formula depends on many aspects: its structure can determine whether it is part of a decidable or undecidable fragment of a theory [2], also, different theory symbols affect problem difficulty in different ways, for example, a non-linear operator like 'multiply' typically makes a formula more hard than 'add'. Still, I have chosen to use the number of free

variables as a proxy to compare different representations of the problem since the theory symbols used remain the same.

Recall from Chapter 1.1.3 that the variables inside Z3 are immutable. For this reason, we must use SSA (single static assignment) to express values which change through time. This means that a new variable is introduced at each timestep of the trick. This is important with respect to the state representation as the size of the state space at a single timestep will be multiplied by the length of the trick k.

A simple way represent this problem could be to encode the state explicitly. In other words, I can store every variable - the unique value, position and face (face being a boolean value which is true if the card is face up) - of each card at each timestep of the trick. Such a representation would be very expressive but also very large. For example, for a trick of deck size d and length k there would be $d!$ permutations of the card values and 2^d possible face-up, face-down permutations. Extending this through every timestep results in a state space of $d! \times 2^d \times k$. Although state-of-the-art SMT solvers are able to solve queries of many variables [11], there is redundancy in this representation resulting in a needless increase in state space size.

The redundancy is in the encoding of card value. Recall the specification (the final condition) of the Baby Hummer Trick, *'the audience selected card is facing the opposite way from all of the other cards in the deck'*. The success of the trick is independent of the card values of every card in the deck other than the one selected by the audience member. It is possible, then, to successfully represent this problem, storing the face of every card and the position of only the audience selected card. This reduces the $d!$ in the explicit representation to d. The state space of this more abstract representation is then $d \times k \times 2^d$. This is how I chose to represent the search space in my implementation. To keep notation consistent with my implementation of the trick state described above, I have extended the definition of the `states` vector, introduced in Section 1.1.4. `states` now contains vector $s_0, ..., s_k$ where each s_t stores the face of every card in the deck at timestep t, along with an audience selected vector $a_0, ..., a_k$ where a_t stores the position of the audience selected card at timestep t. In the original paper, a single element of the `states` vector stored any necessary information about the state of the trick at a given time. I have made this notation more precise to allow me to clearly express specific constraints on these variables.

2.2.2 Implementation

2.2.2.1 Using Arrays

For the `choices` and `comps` vectors, I declare a symbolic integer variable for each timestep in the sequence. Initially, this is how I implemented the face state for the cards - for each timestep t, I declare variables $s_{(t,0)}, s_{(t,1)}, ..., s_{(t,\texttt{depth-1})}$ where `depth` is the number of cards in the deck. Though this seems intuitive, the state variable cannot be implemented this way. This is because, in non-deterministic components, the state must be indexed symbolically. In other words, the state variable which is being constrained depends on the value of another variable. An example of this is in the

semantic constraint for face state in the `cut` component.

$$\phi_{\texttt{cut\_s}} = \bigwedge_{j=0}^{\texttt{depth-1}} s_{(t-1),j} = s_{t,\texttt{shift}(j,c_t)}$$

where

$$\texttt{shift}(j,c) = (j-c)\%\texttt{depth}$$

Suppose each $s_{t,j}$ is an individual variable representing the face value of the jth card at timestep t. The precise function of this constraint is not important (it is explored in depth in Section 2.4). It is important, however, to notice that the value of `shift`, and thus the s varibale being constrained, depends on value of choice variable c_t. In this case, the above constraint would be impossible to encode. This is because at solving time, Z3 does not have access to how these variables have been indexed; we cannot rely on a naming system for indexing within the SMT query.

Instead, I store the state using the Z3 Array type [12]. There are two main operations I use on an Array, `select` and `store`. `(select a i)` takes an Array `a` and returns the value stored at position `i`. `(store a i v)` returns a new array which is the same as `a` only the value at position `i` is value `v`. An important feature of these operations is that the arguments `a, i` and `v` can be symbolic variables. This allows me - as is necessary in encoding the above `cut` constraint - to access the value which is being constrained (a card's face value) with respect to a symbolic variable c_t. In my case, I defined an Array of type `Array(Int, Int)` named `s_<inst>_<t>` for each environment instance `<inst>` and each timestep `<t>`.

2.2.2.2 Be careful with arrays

Z3 Arrays are unbounded [12]. This means that it is very difficult to constrain every value inside the array. This can introduce difficulties when implementing constraints. I will demonstrate how this can go wrong with an example. Suppose a formula constrains two state arrays to not be equal implemented with `array_1 != array_2`. Suppose I have constrained the values of these arrays from `0` to `depth` so that every card in the deck behaves in accordance with the trick semantics relative to the `comps` , `states` and `choices` vectors. In this scenario, it would be possible for the arrays to act as though they satisfy the constraint `array_1 != array_2` though return a trick which violates it. This is because the semantic constraints on the state have only constrained each value from `0` to `depth`, the unbounded number of array values from `depth+1` and upward have not been constrained. This is illustrated in Figure 2.1. A way to avoid this is to never directly constrain equality between two arrays. Instead, one can iterate through the positions of the array which are relevant to the trick, conjoining the non-equalities of each position. This is shown in Figure 2.2, notice how it compares to Figure 2.1.

If such a fault as this appears in a system, there is no meaningful error message, nor is there always an error - sometimes the constraint may be satisfied and sometimes it will not. This gives an idea of how SMT queries are particularly difficult to debug. Such faults delayed my progress with the project, though uncovering their mysteries has been

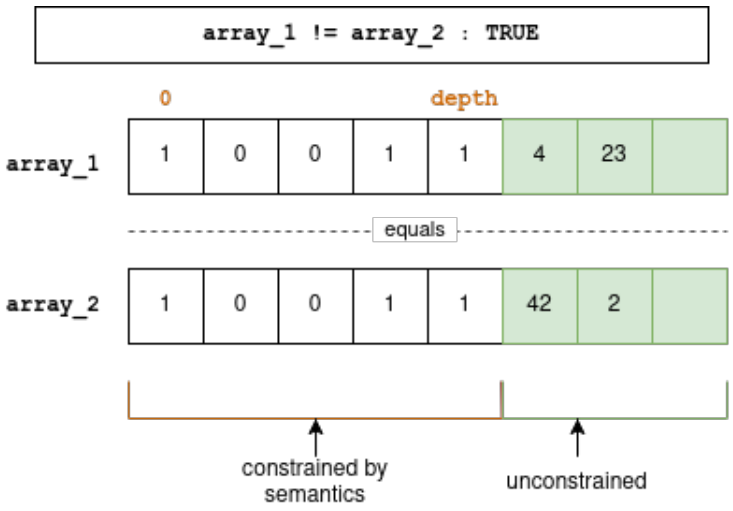

Figure 2.1: Diagram showing the problem with constraints over unbounded Array variables. The white elements are equal across both arrays, the elements in green represent those which are not constrained by semantics or specification, this means they can take any value. Though the states the arrays represent are equal, the arrays themselves are not equal.

a valuable learning experience. A description of some of the techniques that can be used to debug such faults can be found in Section 2.7.

2.3 Specification

Recall from Section 1.1.2 that the specification ϕ_{spec} constrains that a specified result is achieved by the program (or magic trick). The original paper does not discuss the specifics of the specification, giving only its high level formulation. This section will give a formula for the specification and discuss how it can be implemented so that it can be more easily understood and reproduced.

The logical specification of a magic trick encodes its punchline - the part of the trick which should undermine the sense of control given to the audience and inspire wonder. In the Baby Hummer, this punchline is the fact that, at the end of the trick, the audience selected card is facing the other way than all of the other cards in the deck. Recall the `states` vector which I extended in Section 2.2.1. I formulate the specification of the Baby Hummer trick, $\phi_{spec}(\texttt{states})$, as follows:

$$\phi_{spec}(\texttt{states}) = \phi_{down}(s_k, a_k) \vee \phi_{up}(s_k, a_k)$$

Recall that s_k is the final face value of the trick and a_k is the position of audience selected card at the end of the trick. This formalisation splits the the spec into two main scenarios. ϕ_{up}, a constraint which is satisfied when all of the cards are face down except the audience selected card and ϕ_{down} where the cards must be face up except the audience selected card. This captures every scenario in which the specification is satisfied. The

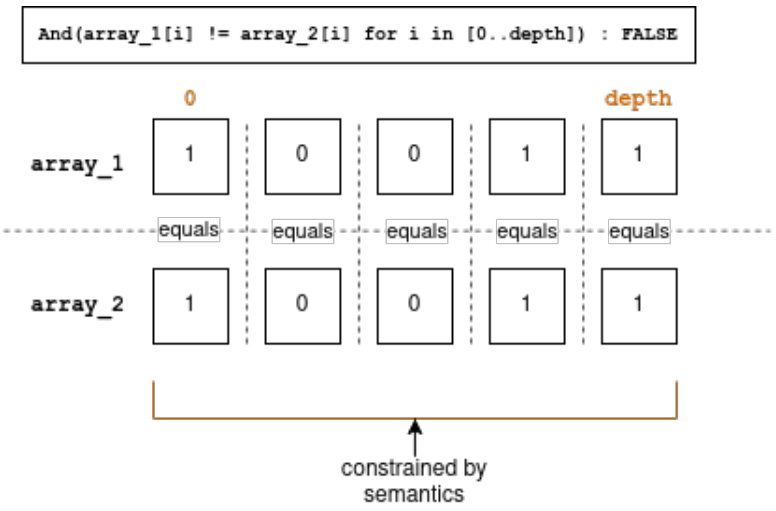

Figure 2.2: Diagram showing how the solution to the problem illustrated in Figure 2.1. The individual relevant elements of the array are iterated over and conjoined. This allows me to control the scope of the formula to only those values which are constrained by magic trick semantics and specification.

specifics of these formulae depend on the implementation and representation of search space - I will describe how I have implemented ϕ_{up} in my system which stores the state of the deck as an array of integers where (-1) represents face down and 1 represents face up:

$$\phi_{up}(s_k, a_k) = \phi_{one\_up}(s_k) \wedge \phi_{select\_up}(a_k)$$

where

- $\phi_{one\_up}(s_k) = \text{Sum}(s_k) = -(\texttt{depth} - 2)$
- $\phi_{select\_up}(a_k) = a_k = 1$

ϕ_{one_up} is satisfied when only one card is facing up. In other words, when one element in the array is equal to 1 and every other element is -1. In this case, there will be $(\texttt{depth} - 1)$ elements equal to -1 and only one element equal to 1. Adding these together gets us $-(\texttt{depth} - 2)$.

ϕ_{select_up} simply states that the audience selected card a_k must be face up.

If there is only one card facing up and the audience selected card is face up then ϕ_{up}, and thus the specification, is satisfied. From the above, the formulation of ϕ_{down} should too be clear.

It is important to note that the Sum operation in ϕ_{one_up} is one which requires care. Recall from Section 2.2.2.2 that Arrays used in Z3 are unbounded. Summing over an unbounded object is clearly problematic. To implement the desired Summing functionality, one should instead add each relevant element in the final state array (those at positions in the range $[0, \texttt{depth}]$) to a list. This enables the use of Z3's `Sum()`

function. This can be called on the list of symbolic variables which creates an expression representing their sum. This is what I did in my implementation.

2.4 Semantics

2.4.1 Overview

The semantics of the trick constrains the values of the `comps` , `states` and `choices` vectors such that they behave like cards in real life. For example, if I take a deck of cards and flip the top card, we would assume that the resulting deck of cards is the same as it was before the flip, only the face of the top card is changed. This assumption about how certain actions affect the state of the deck is an example of semantics.

Recall from Section 1.1.4 the parameterised state transformer $T(i_t, c_t, s_t, a_t)$ which for a given i_t selects a transformer function T_{i_t} and action A_{i_t} and returns the result of $T_{i_t}(c_t, s_t, a_t)$. Note that I have added the a_t variable to be consistent with the extended `states` vector, the functionality of the T has not changed. The state which used to be all stored in s_t is now distributed between s_t and a_t. Accordingly, the type of the state transformers T_{i_t} is now $S \times C \times A \to (S \times A)$.

The state transformer function T_{i_t} characterises how its corresponding action A_{i_t} changes the state. Take for example the action of turning the top card which we will name `turn_top`. The corresponding state transformer for this action will take some input state (s, a, c) (for now we will ignore c) as an argument and return the state (s', a') which would result from flipping the top card of the input state.

The function T allows us to constrain the connections between states in `states` with respect to the actions selected by `comps` and the audience choices in `choices` . The original paper formulates these constraints in a formula named ϕ_{des}, denoted as follows:

$$\phi_{des}(\texttt{comps}, \texttt{choices}, \texttt{states}) = \bigwedge_{t=1}^{k} ((s_t, a_t) = T(i_t, c_t, s_{(t-1)}, a_{(t-1)}))$$

This can be interpreted as, for every timestep t of the trick, the state of the deck (s_t, a_t) must be equal to the result of performing action A_{i_t} with audience choice c_t to the previous state of the trick $(s_{(t-1)}, a_{(t-1)})$.

2.4.2 Formulation

The paper's formulation of ϕ_{des} given above conveys the idea of how `comps` and `choices` assignments constrain the `states` through semantics. Despite this, implementing a function as complex as T inside of an SMT constraint proved to be very difficult and I could not get it to work. Details, even on a high level, on how this might have been implemented were not provided in the paper. Instead, I generated transition system similar to the one used in [3], a logical formulae encoding the semantics for each action for a given timestep t which returns true if and only if a valid transition exists.

The specifics of how any of the actions transform a state - even in the context of a function T - are not given in the original paper. I have given the logical formulae for each action in the Baby Hummer trick below.

2.4.2.1 `turn_top`

$$\phi_{\texttt{turn\_top}}(t, \texttt{states}) = (s_{(t-1),0} = -1 * s_{t,0}) \wedge (a_t = a_{(t-1)}) \wedge (\bigwedge_{j=1}^{\texttt{depth}-1} s_{(t-1),j} = s_{t,j})$$

This formula enforces the state transition of the action `turn_top` for a given timestep t. There are three conjoined expressions in the formula. The first expression enforces that the 0^{th} card at time $(t-1)$ must be the opposite to the 0th card at time t. Since the state values can be either 1 or (-1), multiplying a value by (-1) has the effect of flipping it.

The second expression enforces that the position of the audience selected card is the same at timestep $(t-1)$ as it is at timestep t. This is consistent with the `turn_top` action as the cards in the deck do not change position.

The third expression iterates over the cards in the deck from index 1 to $\texttt{depth}-1$. This covers all of the cards other than the top card. The expression asserts that these cards' face are equal across t and $(t-1)$.

2.4.2.2 `flip_2`

It may seem that the the `flip_2` action is the same as `turn_top` with an extra $s_{t-1,1} = (-1) * s_{t,1}$ term inside. The action is actually more subtle than that.

$$\phi_{\texttt{flip\_2}}(t, \texttt{states}) = \phi_{\texttt{flip\_2\_s}}(t, \texttt{states}) \wedge \phi_{\texttt{flip\_2\_a}}(t, \texttt{states})$$

Above is the high level structure of the `flip_2` action. Since it is quite complicated, I will explain its components one at a time:

$$\phi_{\texttt{flip\_2\_s}}(t, \texttt{states}) = (\phi_{same} \implies \phi_{match}) \wedge (\neg\phi_{same} \implies \phi_{opp})$$

where

- $\phi_{same} = (s_{(t-1),0} = s_{(t-1),1})$
- $\phi_{match} = (s_{(t-1),0} = -1 * s_{t,0}) \wedge (s_{(t-1),1} = -1 * s_{t,1}) \bigwedge_{j=2}^{\texttt{depth}-1} s_{(t-1),j} = s_{t,j}$
- $\phi_{opp} = \bigwedge_{j=0}^{\texttt{depth}-1} s_{(t-1),j} = s_{t,j}$

For brevity, I have omitted the arguments to the ϕ formulae. All formulae that are part of $\phi_{\texttt{flip\_2}}$, unless otherwise stated have the arguments $(t, \texttt{states})$.

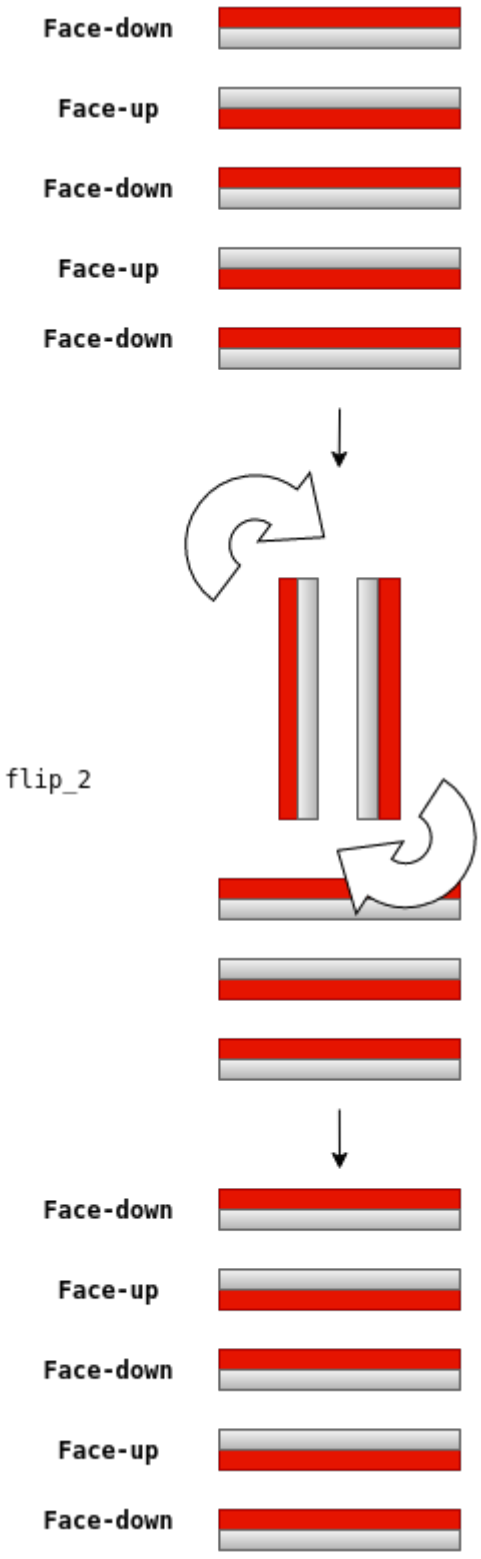

Figure 2.3: Diagram showing face state before and after a `flip_2` action. A cross-section of a deck is shown. The grey represents the front of the card and the red represents its back. Notice how the face state remains the same before and after the action.

The $\phi_{\texttt{flip\_2\_s}}$ formula constrains the face values in the action. At its top level, it is an if, then , else expression. The condition is whether the top two cards have equal face value at the previous timestep. If the condition is satisfied, ϕ_{match} must be satisfied which enforces that the top two positions have flipped face values while the remaining positions stay the same over the timesteps. For example, if the face values at time $(t-1)$ are `[1, 1, 0, 1, 0]`, we expect that flipping the top two cards will result in `[0, 0, 0, 1, 0]` at time t. If the condition is not satisfied, then ϕ_{opp} must be true which constrains that every face value remains the same across the timesteps. This might seem unintuitive. To verify this, refer to the example given in Figure 2.3.

$$\phi_{\texttt{flip\_2\_a}}(t, \texttt{states}) = (\phi_{ontop} \implies \phi_{top}) \wedge (\neg\phi_{ontop} \implies \phi_{noopa})$$

where

- $\phi_{ontop} = (a_{(t-1)} = 1 \vee a_{(t-1)} = 0)$
- $\phi_{top} = (a_{(t-1)} = 1 \implies a_t = 0) \wedge (a_{(t-1)} = 0 \implies a_t = 1)$
- $\phi_{noopa} = a_{(t-1)} = a_t$

The $\phi_{\texttt{flip\_2\_a}}$ formula is responsible for constraining the position of the audience selected card. This formula also takes an if, then, else structure. The condition is that the audience selected card is either at the top or the second top of the deck. In this case, the ϕ_{top} formula must be satisfied which enforces that position of the selected card becomes 1 if it was previously 0 and vice versa. This represents the swapping of positions which happens when two cards are flipped.

If the original condition is not satisfied, then ϕ_{noopa} must hold which enforces that the selected card's position remains the same as it was before. This is because only the top two cards are being flipped.

2.4.2.3 `top_2_to_bottom`

$$\phi_{\texttt{top\_2\_to\_bottom}}(t, \texttt{states}) = \phi_{\texttt{top2\_state}}(t, \texttt{states}) \wedge \phi_{\texttt{top2\_aud}}(t, \texttt{states})$$

where

- $\phi_{\texttt{top2\_state}}(t, \texttt{states}) = \bigwedge_{j=0}^{\texttt{depth}-1} s_{(t-1),j} = s_{t,\texttt{shift(j)}}$
- $\phi_{\texttt{top2\_aud}}(t, \texttt{states}) = a_t = \texttt{shift}(a_{(t-1)})$
- $\texttt{shift}(j) = (j + \texttt{depth} - 2)\%\texttt{depth}$

The formula for `top_2_to_bottom` is slightly more complicated. I have split it into two conjoined expressions, one which encodes constraints on the face of the cards and one on the position of the audience selected card. Both of these expressions make use of the newly defined $\texttt{shift}(j)$ function which given a card at position j returns the new position for that card resulting from performing the `top_2_to_bottom` action.

The $\phi_{\texttt{top2\_state}}$ iterates through every card face, enforcing that the positions of the current timestep's faces are consistent with the expected new positions as returned by $\texttt{shift}(j)$.

For the $\phi_{\texttt{top2\_aud}}$ recall that the audience selected card a_t stores only the position of the selected card. This formula enforces that the previous timestep's position is shifted according to `shift` applied to the previous position.

2.4.2.4 `top_to_bottom`

I have not given a full formal description of the `top_to_bottom` action as it is very similar to `top_2_to_bottom`. Both of these actions shift the position of the face and audience selected card by a constant amount. The amount by which the cards are shifting is encoded in the `shift` function. `top_to_bottom` is exactly the same as its counterpart other than the `shift` function which, for `top_to_bottom`, is defined as so:

$$\texttt{shift}(j) = (j + \texttt{depth} - 1)\%\texttt{depth}$$

2.4.2.5 `cut`

The `cut` action is perhaps the most important, it is the only action which gives the audience control of the trick. In this move the audience selects a random point in the deck and places all of the cards below that point at the top of the deck. The previously described actions have all been deterministic, which is to say, given a state (s,a) the action's resulting state (s',a') will always be the same. `cut` is a non-deterministic action as for a given state (s,a) the resulting state (s',a') depends on the audience choice and is not pre-determined.

Recall that audience choices are represented in a vector $c_1,...,c_k$ named `choices` . Although the audience choices are only used in the `cut` action, we store one at each timestep. In deterministic actions, the transformer function ignores the c value, though in `cut`, we use this value to determine the semantically correct proceeding state. The formula for `cut` is denoted below:

$$\phi_{\texttt{cut}}(t,\texttt{states},\texttt{choices}) = \phi_{\texttt{cut\_s}}(t,\texttt{states},c_t) \wedge \phi_{\texttt{cut\_a}}(t,\texttt{states},c_t)$$

where

- $\phi_{\texttt{cut\_s}} = \bigwedge_{j=0}^{\texttt{depth-1}} s_{(t-1),j} = s_{t,\texttt{shift}(j,c_t)}$
- $\phi_{\texttt{cut\_a}} = a_{(t-1)} = \texttt{shift}(a_t, c_t)$
- $\texttt{shift}(j,c) = (j-c)\%\texttt{depth}$

The above constraint consists of two conjoined expressions, $\phi_{\texttt{cut\_s}}$ and $\phi_{\texttt{cut\_a}}$, which constrain the semantics for the card faces and audience selected card respectively. $\phi_{\texttt{cut\_s}}$ works much like the the the constraints in `top_2_to_bottom`, iterating through each

position at the given timestep and enforcing that the previous face value position is equal to the position according to the shifting function `shift`.

$\phi_{\texttt{cut\_a}}$ enforces that the audience selected card's position at at the previous timestep is shifted according to `shift`.

As well as the previous position j, this new version of `shift` takes also the audience choice parameter c which governs how much the positions are shifted by. In this way, the cut action can be thought of as a parameterised `top_to_bottom` action.

2.4.3 Structuring Semantics

The individual action semantic constraints formulated above only constrain the state for a given timestep t. The order of the actions in the trick correspond to the assignment to the `comps` vector which selects these actions. Recall from the introduction that the actions are stored in a library $\{A_1,...,A_{n+k}\}$. An assignment to `comps` where $i_1 = 3, i_2 = 5, i_3 = 2$ would represent a trick where the action at timestep 1 is A_3, the action at timestep 2 is A_5 and the action at timestep 3 is A_2. To encode this functionality, the semantic constraints must be structured such that whether their effect is active depends on the assignment to `comps` .

It is possible that any action can be selected at any timestep. For this reason I encode the semantics for every action at every timestep conditioned on the assignment to each timestep's corresponding component i_t. For example, for timestep $t = 1$, If $i_1 = 1$, I enforce the semantic constraint corresponding to action A_1 on the state at timestep 1, if $i_1 = 2$, I do the same for action A_2. Iterating this structure over all of the timesteps can be formulated in the high level constraint ϕ_{des} as follows:

$$\phi_{des}(\texttt{comps},\texttt{choices},\texttt{states}) = \bigwedge_{t=1}^{k}\bigwedge_{i'=1}^{n+k} i_t = i' \implies ((\phi_{i'}(t,\texttt{states},\texttt{choices}))$$

Where $n+k$ is the size of the library (and thus the maximum component value) and $\phi_{i'}$ is the semantic constraint corresponding to action $A_{i'}$ which is enforced on the states at time t if the tth value of `comps` (i_t) is equal to i'. This encodes the effect of the assignments to `comps` selecting the actions.

The above formulation of ϕ_{des} requires that each possible value of i has a unique corresponding semantic constraint ϕ_i which is encoded explicitly at every timestep. Recall that an action can only be used once; to create the effect of repeated actions in a trick, multiple copies of an action must be added to the library. In such a case where the library contains multiple copies of an action where, for example, A_i and $A_{i'}$ represent the same move, ϕ_{des} will contain redundant, repeated semantic constraints ϕ_i and $\phi_{i'}$. To avoid these repetitions, I do the following.

I define a move m as a reference to a unique semantic constraint γ_m on `states` and `choices` . The set M is the set of references to every unique semantic constraint referred to by an action in the library. I say an action A_i corresponds to a move m if the action's

corresponding semantic constraint ϕ_i is logically identical to γ_m. The set lib_m is a set of all indices i which select and action A_i corresponding to move m

With these definitions, I can now reduce the size of the ϕ_{des} constraint by not including redundant copies of semantically identical constraints. The new formulation is below:

$$\phi_{des}(\texttt{comps}, \texttt{choices}, \texttt{states}) = \bigwedge_{t=1}^{k} \bigwedge_{m=1}^{|M|} i_t \in lib_m \implies \gamma_m(t, \texttt{states}, \texttt{choices})$$

This enforces that if a component at a given timestep t selects an action which corresponds to a move m, then that move's corresponding semantic constraint γ_m must hold with respect to the timestep t.

Encoding set or list membership in Z3 is difficult. A simple way of implementing the antecedent of the above formula would be to iterate through the elements of lib_m, disjoining whether they are equal to the given component value. I have given an example below in pseudocode:

```
Or([i == comp for comp in lib])
```

where `i` is a value of the component vector and `lib` is the sub-library of a move.

2.4.3.1 Value range

It is important to constrain the values given to the variables such that the semantic constraints work as expected. For example, I enforce that the values of s for every timestep from position 0 up to position $\texttt{depth} - 1$ are either 1 or (-1). If this constraint was not enforced explicitly, a value outside of this range could be produced which 'gets around' a semantic constraint leading to a false trick. I also enforce that the values of `choices` are in the range $[0, \texttt{depth} - 1]$. This is so that each value corresponds to a point in the deck at which the user can cut. These are extra semantic constraints which must be conjoined to ϕ_{des} to ensure that the solver engages with the other semantic constraints appropriately.

2.5 Syntax and other constraints

Though the original paper gave a high level formulation of how trick semantics were constrained in the system, the formulation of syntax in the magic trick is never mentioned directly, also, there is no mention of where syntax should be used in the CEGIS process. This section discusses these issues.

2.5.1 Unique action

The syntax of a magic trick encodes which arrangements of components correspond to a valid trick. To enforce that an action in the library can only be used once, I impose a constraint which forces the values of `comps` to be unique. I name this constraint ϕ_{uniq}. This constraint is very simple and I have omitted its explicit formulation for brevity.

2.5.2 Trivialities

The paper states that initially, the tricks resulting from their system were 'non-interesting'. They identify three properties as making a trick 'non-interesting' and enforce constraints to rule them out. Since these constraints encode how components can be combined and only depend on the components themselves, I view them as syntax constraints. Though the paper identifies these properties, it does not provide the constraints themselves. I include constraints to rule out these properties above as follows.

I name the conjunction of all these trivialities constraints ϕ_{triv}.

2.5.2.1 The `cut` action is not used

The cut action is the only action which introduces non-determinism into the trick. If we do not force this action to appear at least once, the solver is likely to produce entirely deterministic tricks since they are much simpler and easier to synthesise. Such tricks do not give any control to the audience and so are trivial. I rule out such tricks with the following constraint:

$$\bigvee_{t=2}^{k-1} i_t \in cutlib$$

Where *cutlib* is the set of indices which select actions corresponding to the `cut` move. This disjunction is over every timestep t in the trick other than the first and last. The set membership operation would be implemented as is described in Section 2.4.3

2.5.2.2 `cut` appears at the start or end

In the above constraint, I did not iterate over the first and last components for a reason. The paper mentions that tricks where `cut` is the first or last component also appear 'non-interesting'. Such tricks can be ruled out with the very simple constraint

$$i_1 \notin cutlib \wedge i_k \notin cutlib$$

.

2.5.2.3 Flipping actions are not mixed with cut

If all of the actions which change the face of a card appear before the first `cut` action, then there is no non-determinism contributing to the changing of the face. In a situation such as this, the audience's choices will not affect the face of any of the cards in the deck. Since the trick specification depends on the face of the card on the final state, the audience will not feel as though they have control, this is an undesirable property for a trick to have. I rule out such tricks as follows:

$$\bigwedge_{t=2}^{k-1} (i_t \in cutlib) \implies \bigvee_{t'=t}^{k} i_{t'} \in (flip2lib \cup turntoplib)$$

Where $flip2lib$ and $turntoplib$ are the sub-libraries for `flip_2` and `turn_top` respectively; these are all of the flipping moves in the trick. The conjunction in the formula above iterates over every timestep t at which a `cut` move can appear. If the component at timestep t is a `cut` move then there must exist at least one component at timestep $t' > t$ which corresponds to a flipping action.

2.6 CEGIS

A high level introduction to how CEGIS works in general is given in Section 1.1.1. The original paper gives a brief overview of the synthesis of magic tricks providing some high level formulations. This overview does not make clear the meaning of the formulations given, nor does it discuss important details about them. This section will expand upon this, giving a detailed formulation and discussion of how CEGIS can be used to synthesis magic tricks.

2.6.1 Synthesis

Recall that the synthesis stage of CEGIS should produce a semantically and syntactically correct magic trick which satisfies the specification for a subset of input examples which are, in our case, audience choice examples. I denote the set of audience choice examples as $\{\texttt{choices}_1, \texttt{choices}_2, ...\texttt{choices}_L\}$. This is consistent with the original paper. Note that for a given `comps` and $\texttt{choices}_l$ there is a unique corresponding $\texttt{states}_l$.

The paper denotes the synthesis query given to the SMT solver at a high level as follows:

$$\mathcal{F}_{synthesiser} : \exists \texttt{comps} \exists \texttt{states}_1 ... \exists \texttt{states}_L . \psi_{synth}(\texttt{comps})$$

where

$$\psi_{synth}(\texttt{comps}) = \bigwedge_{l=1}^{L} \phi_{des}(\texttt{comps}, \texttt{choices}_l, \texttt{states}_l) \wedge \phi_{spec}(\texttt{states}_l)$$

Where ψ_{synth} is the synthesis constraint. Recall that in the context of a trick `comps` , for each $\texttt{choices}_l$ there is a unique corresponding $\texttt{states}_l$.

The syntax constraint ψ_{synth} enforces that for a trick `comps` to be valid, it must be both semantically correct and satisfy the specification as constrained by ϕ_{des} and ϕ_{spec} for every audience choice example in `choices` .

The synthesiser searches for an assignment to `comps` which satisfies $\mathcal{F}_{synthesiser}$ (and the `states` vectors which uniquely correspond with `choices`) which satisfies ψ_{synth}. If there exists a satisfying assignment for $\mathcal{F}_{synthesiser}$ then the resulting `comps` is passed to the verifier. Otherwise, there does not exist a trick which can satisfy the specification.

2.6.2 Verification

The verification stage of CEGIS should take a trick produced by the synthesiser and find an audience choice which, according to the semantics of a magic trick, violate the

specification. The original paper explains this in a convoluted way, I provide a clear verification formula below.

$$\mathcal{F}_{verifier} : \exists \texttt{choices} \exists \texttt{states}.\phi_{des}(\texttt{comps}, \texttt{choices}, \texttt{states}) \wedge \neg \phi_{spec}(\texttt{states})$$

ϕ_{des} ensures that the choices affect the state in a way that corresponds to magic trick semantics. ϕ_{spec} ensures that the resulting state violates the trick specification.

If an SMT solver queried with $\mathcal{F}_{verifier}$ results in valid counter example, then this counter example is added to the set $\{\texttt{choices}_1, \texttt{choices}_2, ..., \texttt{choices}_L\}$. The synthesiser then searches for a new trick, taking the new set of audience choices into account. If this query is unsatisfiable then there exist no counter-examples. In other words, the sequence of actions selected by `comps` is a valid magic trick.

2.6.3 What about syntax?

The original paper does not mention where the syntax of a trick should belong in the CEGIS process. In the paper's high level formulation of the synthesis and verification constraints, only the ϕ_{des} and ϕ_{spec} constraints are shown. Since the syntax relates to whether a trick is valid, I originally added the syntax constraints ϕ_{triv} and ϕ_{uniq} introduced in Section 2.5 to ϕ_{des}, treating their conjunction as one constraint so that it could influence the tricks in the CEGIS process according to the paper's formulation.

This technique produced semantically and syntactically correct magic tricks. Despite this, I noticed there was a redundancy in the verification constraint. Recall from Section 1.1.2, if the synthesiser which produces a program P (in our case `comps`) does so under a syntax constraint, then I can assume that any P (`comps` vector) which is passed to the verifier always obeys the syntax. This separation of syntax and semantics in CEGIS can be found in [8] which uses a well-formed-program constraint φ_{wfp} in the synthesiser though not in the verifier.

For this reason, I can enforce the syntax constraints only in the synthesis phase and be sure that the resulting magic tricks will be syntactically correct. I give the syntax constraints mentioned above a name ϕ_{syntax} and conjoin it with the synthesis constraint ψ_{synth} introduced in section 2.6.1. This also produces correct magic tricks and it can reduce the synthesis time for certain problems shorter time (Section 4.2)

2.6.4 Implementation

Though I have discussed the CEGIS framework (Section 1.1.1) as well as the formulation of the synthesis and verification constraints (Section 2.6), the implementation of such a system is not trivial. This section will discuss some of the CEGIS implementation specifics of this system, providing details on a number of design challenges and their solutions.

2.6.4.1 Parameterised variables

Recall that the synthesis query $\mathcal{F}_{synthesiser}$ enforces the semantic and specification constraints over every audience choice vector $\texttt{choices}_1, ..., \texttt{choices}_L$. A way one might try to implement this is to generate all of the Z3 variables that will be used in the trick (namely the `comps` , `states` , and `choices` vectors) call them `vars` for example, and generate ϕ_{des} and ϕ_{spec} on these variables for each audience choice in the input set like so:

```
And[And(des(vars), spec(vars), to_constraint(input_list[0])), ...
    And(des(vars), spec(vars), to_constraint(input_list[L-1]))]
```

Where `to_constraint(input_list[0])` generates Z3 formula constraining that the audience choice vector `choices` is equal to the 0^{th} element of the input list. This wouldn't work since this is enforcing contradictions, cheifly, that `choices` is equal to `input_list[0]` and also `input_list[L-1]`. In order to enforce that a given trick is valid over a series of choice examples, I need to define a whole environment of variables (`choices` and `states`) for each choice example in the input list. I define a variable `instance` which refers to the current iteration of the CEGIS loop. This corresponds to the value l which selects the $\texttt{states}_l$ and $\texttt{choices}_l$ introduced in Section 2.6.

I created a function which generates Z3 trick variables and constraints named

```
initialise_env(k, depth, instance)
```

where `k` is the length of the trick, `depth` is the depth and `instance` is a number by which the name of the `choices` and `states` vectors are to be parameterised. I do not parameterise the `comps` vector since I want to synthesise a single `comps` . the parameterised `states` and `choices` vectors represent the effect that this single `comps` has on the list of different inputs.

This function returns `Variables, des, spec` where `Variables` is an object from which the generated Z3 variables can be accessed and `des` and `spec` are the semantic and synthesis constraints over the variables generated by the function.

To give an example of this, suppose I call the function

```
initialise_env(k=15, depth=4, instance=2)
```

The `choices` variables initialised by this function would have the names `c_2_1, c_2_2, ..., c_2_15` where `c` indicates that it is a related to the audience choice, the number after the first underscore is the instance and after the second is the timestep. The fact that variables for different instances have systematically different names means that I can constrain the different choice examples in the input list without unnecessary contradictions, for example:

```
And[And(des_1, spec_1, to_constraint(input_list[0])), ...
    And(des_L, spec_L, to_constraint(input_list[L-1]))]
```

Where `des_1` and `spec_1` are the semantic and specification constraint on only the variables parameterised by instance 1. For now I assume that the variables in the input list are parameterised using the same naming system.

In the improved version of this system where the syntax constraints are removed from ϕ_{des} and are enforced only in the synthesiser (Section 2.6.3), I conjoin to each instance a corresponding syntax constraint. This is generated by a function `to_syntax()` which imposes the syntax constraint on a `Variables` object. I call `to_syntax()` on the variables returned from `initialise_env()` parameterised with each relevant instance.

2.6.4.2 Instances in the verifier

When a satisfying `comps` vector is found, it is passed to the verifier. Recall that the $\mathcal{F}_{verifier}$ query searches for a semantically correct choices vector `choices` (and unique corresponding `states`) for a given candidate `comps` vector, where `comps` has been produced by the synthesiser.

The instance by which the variables representing the counter-example are parameterised by is important. This is because the assignment to these variables will be constrained in the synthesiser with respect to other parameterised variables. When a new counter-example is passed to the synthesiser, the synthesiser constrains the new counter-example with respect to variables parameterised by the current instance. To preempt this, the verifier searches for a counterexample which satisfies the verification constraint on variables parameterised by `instance+1`

2.6.4.3 Re-using constraints

Although the variables and constraints for a given instance are generated automatically by `initialise_env()`, re-generating all of the necessary constraints at every instance is inefficient. This is because both the constraints and the inputs accumulate over the instances. I can reduce the number of calls to `initialise_env()` by storing the synthesis constraint for each instance, appending to it at every iteration. I call this list `synth_list`.

Using this technique, the system only needs to call `initialise_env()` once for each instance. In the synthesis stage, once the current instance's constraint is added to `synth_list`, the entire synthesis constraint can be implemented as

```
And(And(synth_list), And(input_constraints))
```

Where `input_constraints` is a list where every element is the result of `to_constraint()` applied to the input list.

2.7 Debugging

In the process of implementing this system, as with any system, I made mistakes. Debugging these mistakes proved to be difficult. Though I interact with Z3 through a Python API, an error in the verifier, for example, is only visible once an invalid trick has been output. In this case, it is possible to examine the constraints passed to the solver directly, though this is in the form of a large SMT query which is not human-readable. There is also the problem that since the verifier is essentially an automatic test, automatically testing the verifier would result in a copy of itself, which defeats the

timestep	4	5	6	7
move	flip_2	turn_top	flip_2	cut
choice	2	1	0	2
selected card position	0	0	1	2
state	0 0 1	1 0 1	0 1 1	1 1 0

Figure 2.4: Figure representing assignments to `comps` , `choices` and `states` in a way which can be read as a magic trick. Timestep 7 has been highlighted. This is because the states at timestep 7 violates the expected behaviour of the `cut` action with audience choice 2. Such a sequence being produced by the verifier indicates that there is a fault in the semantics for the `cut` action.

purpose. These difficulties slowed my progress in the project. In this section, I discuss some techniques that I used to find faults in this system.

2.7.1 Readability

Though the solver's output can be difficult to read, a function can be defined to translate it into a human readable format to improve debugging. In the synthesis phase, it is not only a `comps` vector representing a trick which is produced but also assignments to the `states` and `choices` vectors. Though these are internally represented as a sequence of arrays and integers, I implemented a function which converts this into a sequence which can be read as a magic trick in a format shown in Figure 2.4. This representation enabled me to find areas where the expected behaviour of the trick is violated. For example, in Figure 2.4, the transition of state between timestep 6 and 7 is incorrect. This example suggests that there is a fault in the semantics of the `cut` action. Without the perspective given by this representation, it would not be clear that the problem with the trick lies in the `cut` action.

2.7.2 Truncating constraints

In situations where a known valid trick is not being accepted by the verifier, it is possible to test this trick on the verifier multiple times, each iteration, removing a constraint. With this technique, one can pinpoint a constraint which is preventing the valid trick from being verified. Finding this point-of-failure does not always give an immediate solution to the bug as the unsatisfiability of a valid trick is usually the result of the interdependance of many constraints. Finding different points-of-failure for the same trick, however, can uncover patterns of failure and give valuable insight into the cause

of the problem.

Chapter 3

Trivial Pursuit: Ruling Out Deterministic Loops

In the paper's conclusion, the writers state: *'We also plan to improve the quality of our results by adding constraints to rule out subsequences being identity operations without user-input. Efficient encoding of such constraints will also be useful in excluding dead code in program synthesis [10].'* This chapter will explore the problem of preventing these identity operations (or deterministic loops as I will occasionally refer to them). As this was regarded as further work by the original paper, it had not been done before and presented a conceptual challenge

I am using the word 'subsequences' to describe what are traditionally referred to as 'substrings' in which all elements must be consecutive. This is to remain consistent with the original paper [10].

3.1 Motivation

Identity subsequences lead to non-interesting tricks. There are two main reasons why this is. I demonstrate these reasons with examples below:

- **Obvious subsequences:** Suppose a trick contained two consecutive `turntop` components. Any audience memeber asked to flip the top card twice would notice that the state has stayed the same and no *magic* shuffling was happening.
- **Undermining other constraints:** The original paper includes constraints to *'mix the card-turning actions with cut'* to avoid *'non-interesting tricks'*. As discussed in section 2.5.2, this is achieved by ensuring that a flip operation takes place after a cut operation. Observe the following trick reported by the original paper.

  ```
  top_to_bottom, top_to_bottom, top_to_bottom, turn_top,
  cut, top_2_to_bottom, cut, flip_2, flip_2, cut,
  turn_top, turn_top
  ```

  Not only are the consecutive `turn_top_2` operations obvious, there is a greater problem with this trick. Although the constraint which forces a flip action to take

place after a cut is satisfied, its *mixing with cut* effect is undermined; as soon as a flip happens, it is immediately reversed! For this reason I make the claim that deterministic loops lead to non-interesting magic tricks.

A naive solution to these problems is to directly prevent flip components from appearing consecutively. I experimented with this though it does not solve the problem. This is, in part, a result of the `noop` operations discussed in section 1.1.4. Though I could prevent against `[flip, flip]`, the system may produce `[flip, noop, flip]` or `[flip, noop, noop, flip]` etc. which would all behave the same way. Even if I was to rule out a `flip` with any number of `noop` in between (which would be a very large constraint), the problem still would not have been solved. As the size of the deck changes, different trivial subsequences emerge. For example, with a deck of size 4, the combination of `[top_2_to_bottom, top_2_to_bottom]` is an identity subsequence though that is not the case in a deck of size 5.

It should be clear that adding constraints to rule out these trivial subsequences on a case by case basis is neither a scalable nor a robust approach. I need instead a general solution. This is what I have found.

3.2 Formulation

Central to ruling out deterministic identity subsequences is the ϕ_{ndl} formula that I created (ndl stands for 'not a deterministic loop').

$$\phi_{ndl}(q, \texttt{states}, \texttt{comps}) = \phi_{looping}(q, \texttt{states}) \implies \phi_{non\_det}(q, \texttt{comps})$$

where

- $\phi_{looping}(q, \texttt{states}) = (s_{(u-1)} = s_v)$
- $\phi_{non\_det}(q, \texttt{comps}) = \bigvee_{j=u}^{v} i_j \in nondetlib$

A subsequence $q = q_1, q_2, ..., q_d$ of list `[1..k]` where k is the trick length, represents a window of consecutive timesteps in a magic trick. Q is a set of these subsequences. To allow for `noop` operations, Q contains only subsequences where d (the length of the subsequence) is greater than 1. Every other subsequence of length k is included in Q.

For brevity, I use the symbols $u = q_1$ and $v = q_d$ to denote the values of the first and last elements in a subsequence q.

Note that for a string of length n, there are $n(n+1)/2$ non-empty substrings. Remember that I am using the term *subsequences* to mean what are usually known as *substrings*.

The helper formula $\phi_{looping}$ evaluates to true if and only if the state of a model at the start of the subsequence is equal to its state at the end of the subsequence. This captures the idea of an identity subsequence. Note how the state at the start of the subsequence

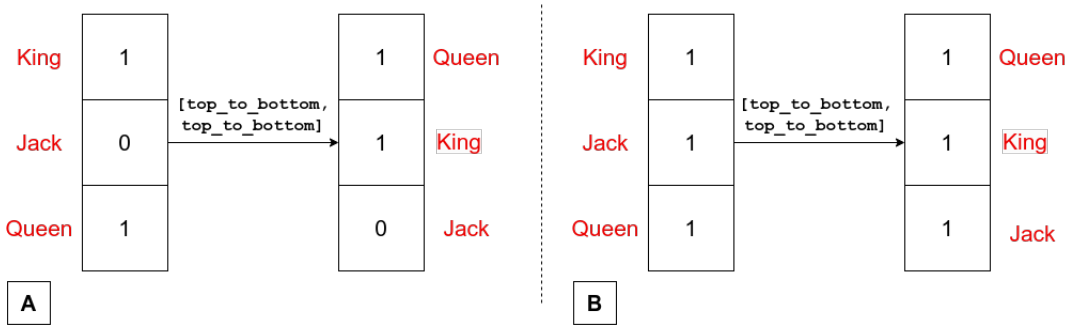

Figure 3.1: Diagram showing how deterministic subsequences behave differently depending on their state. Observe how the two subsequences consist of the same components. Though they are functionally the same, in the abstract search space, model B would be considered a deterministic loop and model A would not.

is $s_{(u-1)}$. This is because s_t, the state at timestep t, is defined as the result of performing the operator c_t on $s_{(t-1)}$.

ϕ_{non_det} evaluates to true if and only if at least one of the model's components at any point in the subsequence is a non-deterministic component. The *nondetlib* used here is a sub-library of all component values which select non-deterministic actions.

the ϕ_{ndl} formula is now quite simple, it returns true if q is not a deterministic loop. The reason ϕ_{ndl} allows for loops which contain a non-deterministic component is to enable situations such as a the zero-cut where a user's choice for a cut operation is 0. This is a valid move but does not change the state of the deck.

For now, for ease of explanation, let's assume that my state representation explicitly encodes the position and value of each unique card (I will later relax this assumption). In this case, I can extend my semantic constraint ϕ_{des} to include

$$\bigwedge_{q \in Q} \phi_{ndl}(q, \texttt{states}, \texttt{comps})$$

which would rule out any trick containing a deterministic subsequence.

As well as the discussed, *double flip* subsequences, this constraint also catches trivialities which emerge only in particular environments. For example, the constraint rules out a trick containing `[top_2_to_bottom, top_2_to_bottom]` where the deck size is 4, though accepts such a combination in tricks of deck size 5.

3.3 Problems in the abstract search space

I have shown that the above constraint rules out deterministic loops in an explicit search space, though the problem is not yet solved. Our ϕ_{ndl} constraints run into serious problems when searching in an abstract space. In a search space as described in section 2.2.1, certain subsequences may satisfy the $\phi_{looping}$ formula even though they consist of valid, non-looping moves. An example of such a subsequence is shown in Figure 3.1.

The subsequence shown in Figure 3.1 has the property that it satisfies $\phi_{looping}$ for *some* `choices` though not for *all* `choices` . I will call such a subsequence a **false loop**

False loops prevent perfectly valid non-looping tricks from being created in the synthesiser. This is because any choice vector in the input set which would cause a model to (falsely) appear as though it looped would seem to the synthesiser like a syntactically incorect program and would not be synthesised.

As well as this, false loops cause a more serious problem. Suppose the syntax contstraints in ϕ_{des} contained the disjunction of ϕ_{ndl} over subsequences defined in 3.2. Now suppose a trick `comps` has been synthesised according to these constraints and passed to the verifier. Suppose this trick contained a false loop. The verifier would search for a counter-example by solving the CEGIS verification constraint (Section 1.1.1):

$$\exists \texttt{choices}.\phi_{des}(\texttt{comps},\texttt{choices},\texttt{states}) \wedge \neg\phi_{spec}(\texttt{states})$$

Since there exists a false loop in `comps` , there exist 'valid' (in terms of real world magic trick syntax) assignments to the choice vector which would violate the ϕ_{ndl} constraint.

To verify this, consider the example in Figure 3.1 - recall that in the verifier, the components are concrete, therefore, any assignment to the choice vector which causes the state to be as it is in model B would introduce a deterministic loop, violating ϕ_{ndl}.

As can be seen from the verification constraint, a valid counter-example must satisfy the constraint ϕ_{des}. It follows that the ϕ_{ndl} constraint has not only constrained the value of the components (as intended), it has also indirectly constrained the choice values.

As a result of this, there arise situations in which the verifier returns a trick because it can not find any counter-examples, even though there exist valid audience choices which violate the trick specification. These are false magic tricks which undermine the integrity of the whole system.

3.4 Exile the false loops

In Section 3.3 I described the two main problems that false loops cause.

1. Some invalid tricks are incorrectly verified
2. Some valid tricks are not created in the synthesiser.

This section will formulate the solution I have found for both of these problems.

3.4.1 Forbidding invalid tricks in the verifier

Recall that in the verification step, A component vector is given and the system searches for a counter-example in the form of a choice vector - in this step, there is no assignment that can be made to the choice vector which would change the value of the component vector. If I assume that the given component vector does not contain any deterministic loops then the verifier has no way to create them. For this reason, I am able to exclude

the ϕ_{ndl} constraint from the verifier though (to ensure that the synthesised models do not contain deterministic loops) keep it in the synthesiser.

Suppose a model is passed to the verifier which contains a false loop. Even if the only counter-example to this model was a choice vector which would cause the false loop subsequence to behave like a loop (as in Figure 3.1 model B), the verifier would find the counter-example as it is not constrained by ϕ_{ndl}. I have shown that a verifier as described here will never return a magic trick if there exists counter-example.

Excluding ϕ_{ndl} from the verifier introduces a difference between the the synthesiser and the verifier. This creates a further problem: The verifier can now add counter-examples to the input set which contradict the ϕ_{ndl} constraints in the synthesiser which will lead to unsatisfiable magic tricks. This problem is addressed in the following section.

3.4.2 Including valid tricks in the synthesiser

When ϕ_{ndl} constraints are embedded in ϕ_{ndl}, the synthesiser will not generate a trick unless, for all $\texttt{choices}_l$ in the input set, the state doesn't exhibit loop-like behaviour. This means that many 'valid' tricks containing false loops may not be synthesised.

The goal of the ϕ_{ndl} constraint is to allow for tricks containing false loops but to rule out tricks containing *real* deterministic-loops. Recall the definition of a false loop: 'a trick that satisfies $\phi_{looping}$ for *some* `choices` though not for *all* `choices` ' (Section 3.3). I define a *real* looping trick as a trick that satisfies $\phi_{looping}$ for all `choices` . It follows that for a given subsequence q and trick `comps` , if there exists any possible valid assignment to `states` (dependent on the synthesis constraint ψ_{synth}) that satisfies $\phi_{ndl}(q, \texttt{states}, \texttt{comps})$ then that subsequence does not contain a deterministic loop.

Of course, an existential quantification over the possible values of `states` is very expensive. Looking at the problem intuitively, however, it becomes apparent that `choices` vectors which undermine a false loop (if there is a false loop) are very dense. In other words, most audience choices will make a false-looping subsequence appear as though it doesn't loop. To verify this, look again at figure 3.1, the only initial states which make this subsequence appear to loop are `[1, 1, 1]` and `[0, 0, 0]`, every other state undermines this false loop. Following this intuition I found a way to practically implement this constraint without using a very large quantifier.

This method depends on the fact that the synthesiser is always used in the context of the CEGIS loop; a trick is only ever synthesised in the context of the input set. I am able to exploit this. Instead of quantifying over every possible `states` assignment, I can approximate it by quantifying over only the ones which arise from the $\texttt{choices}_l$ assignments in the input set (recall that there is a unique $\texttt{states}_l$ vector for each $\texttt{choices}_l$ vector). As the number of iterations in the CEGIS loop are far fewer than the size of the input space, this greatly reduces the size of the formula. The formula for the synthesiser is then written as follows:

$$\mathcal{F}_{synthesiser} : \exists \texttt{comps} \exists \texttt{states}_1 ... \exists \texttt{states}_L$$
$$\psi_{synth}(\texttt{comps}) \wedge \forall q \in Q. \bigvee_{l=1}^{L} \phi_{ndl}(q, \texttt{states}_l, \texttt{comps})$$

where ψ_{synth} is the synthesis constraint as defined in Section 1.1.1 (this doesn't contain the ϕ_{ndl} constraint) and Q is the set of subsequences.

Reducing the quantification over all possible `states` to only the ones resulting from the input set means that valid tricks which contain false loops may be rejected by the synthesiser. However, the synthesiser will never produce a trick which contains a deterministic loop. The reasoning for this is as follows: suppose there existed a trick `comps` which satisfied the ψ_{synth} constraint and contained a *real* deterministic loop. By definition, there would exist some subsequence q which for every valid `choices` vector, $\phi_{ndl}(q, \texttt{states}, \texttt{comps})$ is violated (`states` is the unique corresponding vector to `choices`). It follows that $\mathcal{F}_{synthesiser}$ would always reject such a trick as it depends on finding at least one `choices` vector for every subsequence which satisfies $\phi_{ndl}(q, \texttt{states}, \texttt{comps})$ which, as we have shown, it will never do.

Although I cannot guarantee that the synthesiser will consider all valid tricks, there are properties of the input set that mean that it is less likely that a valid trick specification returns UNSAT. The intuition behind this is that when the input set is small, Even though the ϕ_{ndl} constraint must be satisfied over one of the small number of `choices` vectors, this is counterbalanced by the fact that satisfying tricks are very dense in the search space. This is because the specification constraint only needs to hold for a small number of user inputs. As the input set grows, valid tricks become more sparse in the search space as the specification must hold for a larger number of inputs, at this point the ϕ_{ndl} constraint can consider a large amount of `choices` and it is more likely that if a trick contains a false loop, a `choices` vector is found for which the false loop doesn't behave like a loop.

As well as the counterbalancing mentioned above, since the input set consists of counter-examples from previous tricks, each input will cover a different area of the state space [16]. For this reason, the types of `states` vectors resulting from the input set are more likely to undermine a false loop than a random sampling of `states` vectors.

Though the constraint introduced in this chapter enforces the ordering of components, I do not treat it purely as a syntax constraint. This is because the way in which it is constrained depends on the semantics of the trick; off the shelf CEGIS solvers such as CVC5 do not allow syntax constraints which depend on anything other than the output components [13]. Looking through the literature, I was unable to find a paper which approximates the CEGIS process by disjoining constraints over the input set as I have done in this chapter. This was a novel approach which solves a problem regarded by the authors of the original paper as further work.

3.5 Implementation

Although the the augmentation of the sythesiser described above introduces a large number of constraints. I store these constraints in such a way that minimises the amount of time spent generating them. This approach, similar to the one discussed in Section 2.6.4.3, takes advantage of the fact that the constraints across iterations are largely the same - in the conjunction/disjunction alternation: $\bigwedge_{q\in Q}\bigvee_{l}^{L}$, the number of subsequences q is constant and the number of examples in the input set L increase by only 1. This means that across iterations, there are $|Q|\times(L-1)$ constraints which stay the same.

To avoid recomputing these constraints at every iteration, I store a 2-d array of size $|Q|\times L$ called `conjunction`. `conjunction[q][l]` points to the Z3 representation of $\phi_{ndl}(q,\texttt{states}_l,\texttt{comps})$. At every iteration in the synthesis step, the input set will have increased in size by 1. This is updated in `conjunction` as so:

```
i = 0
for dis in conjunction:
    dis.append(phi_ndl(q[i], states[L-1], comps))
    i ++
```

where `phi_ndl()` is a function which returns $\phi_{ndl}(q,\texttt{states},\texttt{comps})$, in Z3 form, `q` is a list of all subsequences and `states` is a list of Z3 `states` variables indexed such that `states[L-1]` is $\texttt{states}_L$

To add these constraints to a solver object `Solver`, I do the following

```
for dis in conjunction:
    Solver.add(Or(dis))
```

The `Or` function of Z3 takes a list of Z3 objects and returns the disjunction of them. Adding these constraints sequentially has the effect of conjoining them. The structure of this array is illustrated in Figure 3.2.

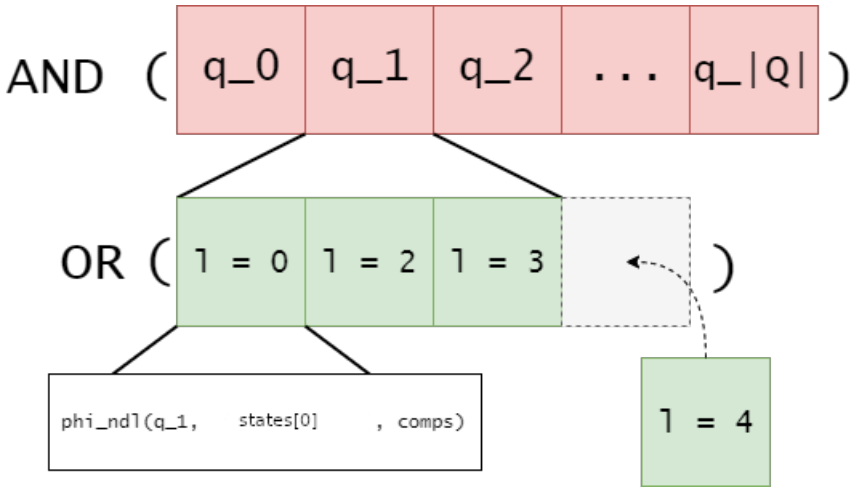

Figure 3.2: Diagram showing how the ϕ_{ndl} constraint is implemented in the synthesiser with a 2-d array. This can be stored and reused across iterations, only an extra input must be added for each subsequence q. The addition of an extra input is represented by the `l = 4` block.

Chapter 4

Evaluation and conclusion

4.1 User study

In this section I evaluate the system with a user study comparing reported audience engagement for different magic tricks. Magic tricks take place in the mind of the audience and so the must be performed if they are to be evaluated. The original paper does not evaluate its magic tricks, it only reports the time it takes to synthesise them. In the original paper's conclusion, the authors state that automated synthesis of magic tricks can be used as an interesting example in computer science courses. Since engagement is important in the classroom [4], I have used the participants report of how engaging a trick was as a metric with which to compare the magic tricks.

To evaluate user engagement with the trick, I gathered consenting students at a number of sessions and performed 3 different types of 'The Baby Hummer' trick at each session. The different types are as follows:

1. **baseline-4** This is the category of tricks that were reported by the original magic trick paper that contain a deterministic loop as described in Chapter 3. These tricks have a deck depth of 4. For example:

   ```
   flip_2, top_to_bottom, top_to_bottom, turn_top, cut,
   top_to_bottom, cut, flip_2, flip_2, cut, turn_top, turn_top
   ```

2. **ndl-4** The tricks produced by my system with the same parameters (length, depth) as baseline-4, except deterministic loops are forbidden. For example:

   ```
   flip_2, turn_top, top_to_bottom, top_2_to_bottom, straight_cut,
   flip_2, straight_cut, flip_2, top_2_to_bottom, straight_cut,
   turn_top, top_2_to_bottom, turn_top
   ```

3. **ndl-6** A trick produced by my system which forbids deterministic loops and has depth of 6. For example:

   ```
   flip_2, top_2_to_bottom, turn_top, top_to_bottom, flip_2,
   top_2_to_bottom, cut, turn_top, flip_2, turn_top,
   top_2_to_bottom
   ```

After each trick, I asked the participants to rate how engaging the trick was out of 10.

For each category, there are a number of tricks. I select a random trick from each category at each session to make the comparison more fair.

A comparison of the mean ratings for each trick can be seen in Figure 4.1. Observe how the tricks which forbid deterministic loops receive a higher rating than the tricks produced by the original paper. This is consistent with the claim made in Section 3 that deterministic loops result in non-interesting tricks. It also demonstrates the effectiveness of my constraint which forbids these loops.

Comparing the two categories which have forbidden deterministic loops shows that the greater depth had a only a small positive effect on trick engagement. Since a trick of length 6 takes a much greater amount of time to synthesise (Figure 4.2), a trick in the category **ndl-4** would likely be the best candidate for use as a classroom example.

4.2 Runtime

Throughout this project, I describe different ways of implementing the system. In this section, I compare the runtime of some of these implementations over a series different of trick depths. The different implementations are as follows:

1. **baseline** This category contains implementations of the system which enforce only the constraints as described by the original paper.
2. **base + ndl** This category contains implementations which have all of the properties of the baseline but also has the constraints to prevent deterministic loops as described in Section 3.
3. **triv not in verifier** This category contains implementations which have all of the properties of **base + ndl** except the syntax constraints are separated out from the semantic constraints and only enforced in the synthesiser as described in Section 2.6.3. I report some of the tricks produced in this category in Section 4.3.

For each implementation, I synthesise 6 tricks and calculate the mean synthesis time. I repeat this for different trick depths. To ensure that different tricks are being synthesised each time, I set the initial input example to be different for each synthesis run.

The results from this experiment can be seen in Figure 4.2. Observe how for all of the depth values, the baseline is the quickest and appears to be negligibly affected by the increase in depth. To explain why this happens, recall from Section 3 that the deterministic loops can undermine the ‘flip after cut’ constraint by having a double flip e.g. `[flip_2, flip_2]` which prevents the ‘mixing with cut’ effect. If the cards are not mixed after a cut move (the only non-deterministic move), then the trick effectively becomes deterministic. In this case, the depth of the deck will have only a minor affect on the runtime.

Observe how the runtime for the implementations which contain ndl constraints grows quickly as the depth increases. The **triv not in verifier** is faster than **base + ndl** for the depth of 6. This is simply because the query is smaller in the former. Despite

this, for depth = 5, **base + ndl** is faster. This was not what I expected since there are fewer constraints and an otherwise identical model. However, the runtime of an SMT query cannot be garnered from query size alone [11]. In future work I would run this experiment again with more data to get a better picture of how this implementation affects performance.

It is worth mentioning that the quickest synthesis time for 'The Baby Hummer' reported by the original paper was 11m 11sec. recall that the original paper did not enforce any deterministic loop constraints. The longest synthesis time for this trick in my implementation was 25sec. As this paper gave only a high level description of the system, it is hard to know what contributed to this difference in runtime. However, the original paper was written in 2016 which makes it very likely that a great part of the difference in performance has come from the huge improvement of SMT solvers in the last decade [11].

4.3 Magic tricks

In this section I report some of the magic tricks produced by the system with different depths along with their runtimes. These tricks were produced with constraints that ruled out deterministic loops, as described in Section 3. The syntax constraints were enforced only in the synthesiser as described in Section 2.6.3. These tricks belong to the **triv not in verifier** category described in Section 4.2.

- **depth = 3**

  `flip_2, top_2_to_bottom, top_2_to_bottom, straight_cut, turn_top, flip_2, turn_top, flip_2, top_to_bottom, top_to_bottom, turn_top`

  Runtime: 30.5 seconds

- **depth = 4**

  `flip_2, turn_top, straight_cut, straight_cut, top_2_to_bottom, top_to_bottom, turn_top, top_to_bottom, top_to_bottom, turn_top, top_2_to_bottom`

  Runtime: 14.9 seconds

- **depth = 5**

  `flip_2, top_2_to_bottom, flip_2, straight_cut, straight_cut, top_to_bottom, straight_cut, flip_2, turn_top, top_to_bottom, turn_top, top_to_bottom`

  Runtime: 123.6 seconds

- **depth = 6**

  `turn_top, top_to_bottom, flip_2, top_2_to_bottom, flip_2, straight_cut, top_2_to_bottom, top_to_bottom, turn_top, flip_2, turn_top`

Runtime: 899.1 seconds

4.4 Contributions

Throughout this paper, I have given a clear and detailed discussion of the system, providing formulae and implementation details where appropriate. This has made the original magic trick paper, as well as synthesis systems for different magic tricks, easier to reproduce.

I have improved the system described by the original paper, encoding constraints which rule out deterministic subsequences. This was an improvement suggested by the authors as further work. I have explored the problem on a high level and also discussed implementation details and challenges with different search spaces.

I have evaluated these tricks with a user study and demonstrated that my improvement to the system leads to more engaging magic tricks.

4.5 Further work

The original paper mentions that the efficient encoding of constraints which rule out deterministic subsequences could be used to exclude dead code in program synthesis [10]. Although the solution I proposed works well in the domain of magic tricks, applying this technique to program synthesis would be an interesting direction for further work as there is a wider range of applications for program synthesis than for magic trick synthesis. Specifically, I would like to investigate whether the space of solutions for practical program specifications is sufficiently dense such that my technique does not prune all valid programs.

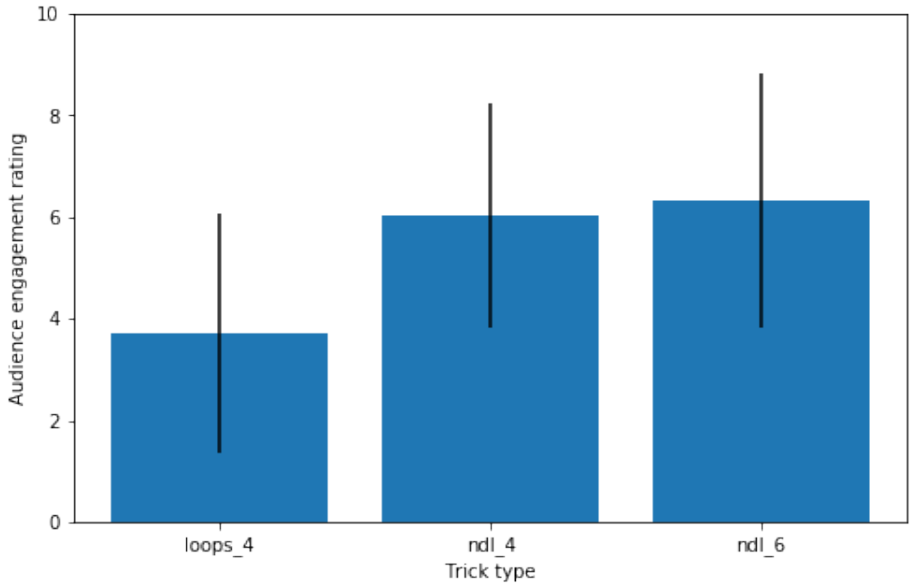

Figure 4.1: Plot comparing the reported audience engagement for different trick types. The error bars represent the standard deviation. This plot was created from 27 data points gathered from 9 participants.

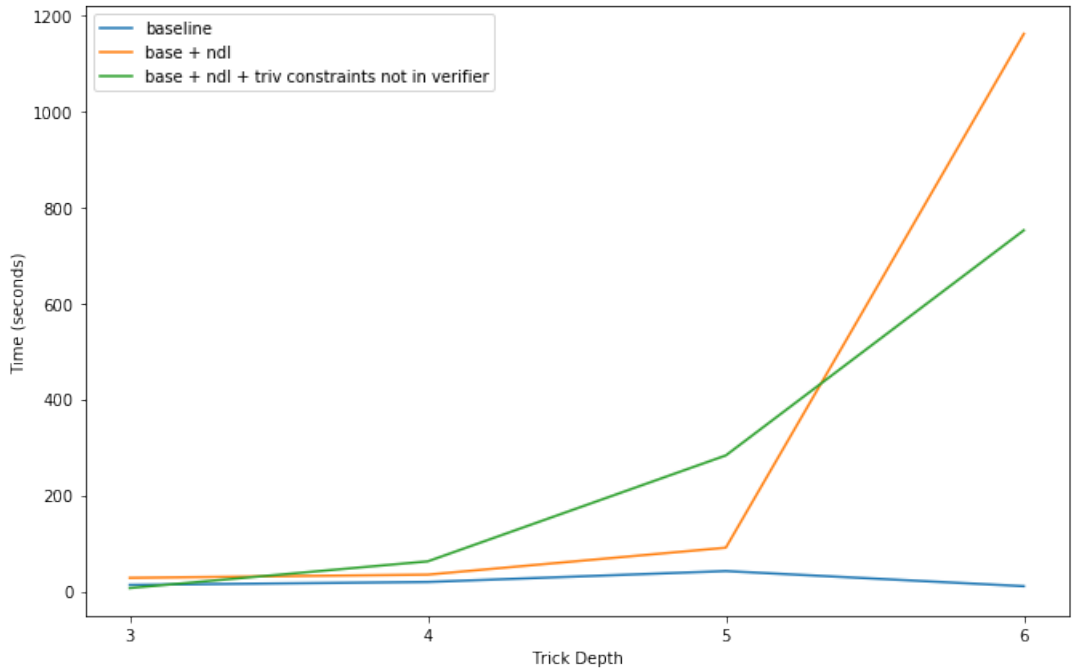

Figure 4.2: Plot showing the how the synthesis runtime for different implementations of the system vary with the depth of the trick. Tricks are all synthesised on a student DICE account

Bibliography

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 270–288, Cham, 2018. Springer International Publishing.

[2] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 427–442, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[4] HAMISH COATES. The value of student engagement for higher education quality assurance. *Quality in Higher Education*, 11(1):25–36, 2005.

[5] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[6] Persi. Diaconis. *Magical mathematics : the mathematical ideas that animate great magic tricks / Persi Diaconis, Ron Graham ; with a foreword by Martin Gardner.* Princeton University Press, Princeton, 2012.

[7] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Component-based synthesis applied to bitvector programs. 04 2011.

[8] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI'11, June 4-8, 2011, San Jose, California, USA*, June 2011.

[9] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224, 2010.

[10] Susmit Jha, Vasumathi Raman, and Sanjit A. Seshia. On exists forall unique exists solving: A case study on automated synthesis of magic card tricks. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 81–84, 2016.

[11] Daniel Kroening. *Decision procedures : an algorithmic point of view / Daniel Kroening, Ofer Strichman ; foreword by Randal E Bryant.* Texts in theoretical computer science. Springer, Berlin, second edition. edition, 2016.

[12] Microsoft. Arrays: Microsoft online z3 guide, https://microsoft.github.io/z3guide/docs/theories/arrays/.

[13] Elizabeth Polgreen, Saswat Padhi, Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. The sygus language standard v 2.1. 2021.

[14] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 74–83, Cham, 2019. Springer International Publishing.

[15] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[16] Armando Solar-Lezama. The sketching approach to program synthesis. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pages 7–8, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

Appendix A

Participants' information sheet

Participant Information Sheet

Project title:	The automated synthesis of card magic tricks
Principal investigator:	Elizabeth Polgreen
Researcher collecting data:	Rory Fotheringham
Funder (if applicable):	

This study was certified according to the Informatics Research Ethics Process, RT number **7241** Please take time to read the following information carefully. You should keep this page for your records.

Who are the researchers?

Rory Fotheringham Elizabeth Polgreen

What is the purpose of the study?

The aim of the study is to identify a relationship between the attributes of a magic trick (number of cards, number of moves etc.) and how engaging the trick is. The study will also record the effect that learning the trick oneself has on engagement. The data collected from this study will feed into the design of more engaging tricks which can be used in the classroom as a teaching resource.

Why have I been asked to take part?

You have been asked to take part because you are a university student in a STEM degree. This group is most likely to benefit from the results of this study as it will lead to the development of an improved STEM resource.

Do I have to take part?

No – participation in this study is entirely up to you. You can withdraw from the study at any time, up until you hand in your answer form without giving a reason. After this point, personal data will be deleted and anonymised data will be combined such that it is impossible to remove individual information from the analysis. Your rights will not be affected. If you wish to withdraw, contact the PI. We will keep copies of your original consent, and of your withdrawal request.

What will happen if I decide to take part?

Specify:

After a short introduction, Rory (the researcher) will show you a series of short magic tricks. You will then be split into small groups where you will each learn a magic trick from a list of simple instructions and show it to the others in your group. After each trick you will be asked to assign a number between 0 and 4 representing how engaged you were by the trick. The session will last 45 minutes there will be no audio or video recording of the session.

Are there any risks associated with taking part?

There are no significant risks associated with participation.

Are there any benefits associated with taking part?

There will be an array of gluten free and vegan baked goods at the session

What will happen to the results of this study?

The results of this study may be summarised in published articles, reports and presentations. Key findings will be anonymized: We will remove any information that could, in our assessment, allow anyone to identify you. With your consent, information can also be used for future research. Your data may be archived for a maximum of 4 years. All potentially identifiable data will be deleted within this timeframe if it has not already been deleted as part of anonymization.

Data protection and confidentiality.

Your data will be processed in accordance with Data Protection Law. All information collected about you will be kept strictly confidential. Your data will be referred to by a unique participant number rather than by name. Your data will only be viewed by the researcher/research team [Elizabeth Polgreen, Rory Fotheringham].

All electronic data will be stored on a password-protected encrypted computer, on the School of Informatics' secure file servers, or on the University's secure encrypted cloud storage services (DataShare, ownCloud, or Sharepoint) and all paper records will be stored in a locked filing cabinet in the PI's office. Your consent information will be kept separately from your responses in order to minimise risk.

What are my data protection rights?

The University of Edinburgh is a Data Controller for the information you provide. You have the right to access information held about you. Your right of access can be exercised in accordance Data Protection Law. You also have other rights including rights of correction, erasure and objection. For more details, including the right to lodge a complaint with the Information Commissioner's Office, please visit www.ico.org.uk. Questions, comments and requests about your personal data can also be sent to the University Data Protection Officer at dpo@ed.ac.uk.

Who can I contact?

If you have any further questions about the study, please contact the lead researcher, Rory Fotheringham at s1849475@ed.ac.uk

If you wish to make a complaint about the study, please contact inf-ethics@inf.ed.ac.uk. When you contact us, please provide the study title and detail the nature of your complaint.

Updated information.

If the research project changes in any way, an updated Participant Information Sheet will be made available on http://web.inf.ed.ac.uk/infweb/research/study-updates.

Alternative formats.

To request this document in an alternative format, such as large print or on coloured paper, please contact Rory Fotheringham at s1849475@ed.ac.uk

General information.

For general information about how we use your data, go to: edin.ac/privacy-research

Appendix B

Participants' consent form

Participant number:_____

Participant Consent Form

Project title:	The Automated Synthesis of Card Magic Tricks
Principal investigator (PI):	Elizabeth Polgreen
Researcher:	Rory Fotheringham
PI contact details:	Elizabeth.polgreen@ed.ac.uk

By participating in the study you agree that: I have read and understood the Participant Information Sheet for the above study, that I have had the opportunity to ask questions, and that any questions I had were answered to my satisfaction.

- My participation is voluntary, and that I can withdraw at any time without giving a reason. Withdrawing will not affect any of my rights.
- I consent to my anonymised data being used in academic publications and presentations.
- I understand that my anonymised data will be stored for the duration outlined in the Participant Information Sheet.

Please tick yes or no for each of these statements.

1. I allow my data to be used in future ethically approved research.

Yes	**No**

2. I agree to take part in this study.

Yes	**No**

Name of person giving consent	Date dd/mm/yy	Signature

Name of person taking consent	Date dd/mm/yy	Signature

THE UNIVERSITY of EDINBURGH informatics