Alice in Wineland: A Fairy Tale with Contracts[⋆]

Dilian Gurov[1], Christian Lidström[1], and Philipp Rümmer[2]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
[2] Uppsala University, Uppsala, Sweden

Abstract. In this tale Alice ends up in Wineland, where she tries to attend the birthday party of one of its most beloved inhabitants. In order to do so, she must learn about contracts and how important they are. She gets exposed to several contract languages, with their syntax and semantics, such as pre- and post-conditions, state machines, context-free grammars, and interval logic. She learns for what type of properties they are appropriate to use, and how to formally verify that programs meet their contracts.

1 Down the Rabbit-Hole

It was a sunny summer afternoon and Alice was once again sitting on the bank; this time she was alone, since her sister did not like much to sit in the sun. To keep herself occupied, Alice had brought the book she had found in her father's workshop the other day: it was a curious book, thick and heavy, and so worn from many years of study that it almost fell apart in her hands when she tried to open it. What had caught her attention was the picture on the cover page, showing a smiling keyhole in beautiful royal blue colour. 'I didn't even know that keyholes could smile,' thought Alice by herself, and started reading.

Even more curious than the outside was the inside of the book. Being a modern child, Alice had a solid grasp of micro-, nano-, and pico-computers, and had already written several programs herself—and a book about computers it was, as Alice had realised quickly—but the words passing by, as she was going from page to page, were so strange and unknown that she quickly started to feel all numb. She read about syntax and semantics, about functions with three or more possible values, she saw humongous trees growing to the sky, and many weird symbols that seemed to be written upside-down. She came across pages over pages filled with equations, sometimes in black and sometimes in grey colour (or maybe the print had faded over the years), and on one page she even found an egg that had the words 'Real World' written on it. Many times Alice stumbled over prose seemingly familiar, but used in a way that was utterly confusing, among those long and tedious considerations that involved banks and credit cards; and several times even mentioned contracts!

'Oh, there is no use in reading the book,' thought Alice, increasingly desperate, 'this book is perfectly idiotic!' when she suddenly noticed the White Rabbit

[⋆] We wish to thank Lewis Carroll, whose books inspired us to write this paper. Also, many thanks to Hurt Sandra Hedström for the wonderful state machine illustration.

rushing past her, mumbling its usual 'Oh dear! Oh dear! I shall be late!' Alice, knowing how fast the Rabbit used to walk, went instantly after it and asked, 'Late for what?' 'Oh dear! Why, for Reiner's birthday party, of course!' And down it went into the rabbit-hole, and down went Alice after the Rabbit.

2 Before and After the Footman

Alice was falling, and falling, and falling. That was not a very remarkable thing by itself, as Alice was already used to the journey through the rabbit hole from her previous adventures, and she was not in the least afraid—what caught Alice unprepared were rather the strange writings she could glimpse flying by on the tunnel wall, the likes of which she had never noticed before. In the beginning, Alice's fall was too fast to read any of the words, but as she started to pay closer attention, Alice could make out individual letters; and soon she recognised the sentences, symbols, and pictures she had seen in the book with the keyhole! 'Oh dear!' thought Alice, 'this will be a tricky adventure!'

Thump! As she was still contemplating about syntax, semantics, and most of all about contracts and banks, the fall came to a sudden end, and Alice found herself sitting on an open plain with a little house in it. THUMP! There was a most extraordinary noise going on within—a constant howling and sneezing, and every now and then a great crash, as if a dish or kettle had been broken to pieces. This must be it, Alice thought, the place of Reiner's birthday party!

Alice went timidly up to the door, and knocked, but there was no answer. For a minute or two she stood looking at the house, and wondering what to do next, when suddenly a footman in livery came running out of the wood. He rapped loudly at the door with his knuckles, and a second later it was opened by another footman. The first footman produced from under his arm a great letter, nearly as large as himself, and this he handed over to the other, saying in a solemn tone, 'For the Professor. An invitation to wine tasting!'

The next time Alice looked, one of the footmen was gone, and the other one was sitting on the ground near the door, staring stupidly up into the sky.

Alice really wanted to join the party, so she knocked a second time, but to no avail. 'There's no sort of use in knocking,' said the footman, idling beside her, 'and for many reasons. First, as you can see, it is me who admits people to the party, and I'm on the same side of the door as you are.'

'Pray, how can I enter then?' cried Alice, suddenly afraid she might have arrived too late. 'I will miss all the fun!'

'There is much you have to learn first,' said the footman, 'before Reiner will receive you. Few are deemed worthy, and you know next to nothing. At this party, everything that is done or happens has a beginning and an end. It will happen only when the beginning is possible, and it can finish only when the outcome is satisfactory.'

'This was not an encouraging opening for a lecture.' Alice replied, rather shyly, 'I—I don't understand a word of what you said.'

The footman gave a deep sigh. 'It is hard work teaching you humans even the simplest things. You will make a complete fool of yourself at the party!' He hesitated, sighed again, and then started to explain.

'The world is evolving as the result of operations or actions applied to it: opening the door, entering the house, taking a seat, or emptying a glass of wine. You might think of such operations as just names, but they carry some meaning as well. We need to have a common understanding of the operations, lest we end up emptying a bottle of wine when we really wanted to write a well-reasoned research article.

We therefore describe the meaning of operations using *contracts,* which will enable civilised people to converse without the danger of any misunderstanding. As an example that even you, my child, will understand, consider the action of sitting down:

Operation:	Take a seat
Pre:	Person is standing $\wedge$ Person was offered seat $\wedge$ Seat is free
Post:	Person is sitting

This contract gives three pieces of information: the *name* of an operation, a *pre-condition* that has to be satisfied before we can even consider starting the operation, and a *post-condition* that describes what we can expect after the operation has finished. The contract does not tell us *how* we can take a seat, which will depend on whether you are a human, a caterpillar, or a tortoise, but it will abstractly describe the assumptions and the result of sitting down.'

A question had formed in Alice's head, and at this point she managed to interrupt the monologue of the footman: 'But I much prefer to sit down whenever I want, not only when somebody has invited me to do so! What use has a contract if the pre-condition is not satisfied?'

The footman felt uneasy, confronted with ignorance of this extent, and put on an indignant face. 'A contract will not tell you anything about the outcome of an operation when you fail to take the pre-condition into account. Even though it might appear, at face value, that you could take seat on every free chair, doing so uninvited might have the most unintended side effects!

But your question has certain merits, as it highlights the slightly surprising semantics of pre-conditions, an element of confusion that can be traced through the literature. In the logic of Hoare [8], and in various methods for specifying programs (for instance [17, 11]), a post-condition applies when the pre-condition is satisfied at the point when the operation occurs; nothing is said about cases in which the pre-condition does not hold. In our case, if a person was never invited to sit down, no conclusions can be made about the effect of taking a seat. The outcome of sitting down is one of the possible effects, but neither mandated nor forbidden. In other approaches, however, pre-conditions are interpreted differently and more strictly, namely as necessary conditions for being able to embark upon an operation in the first place (for instance in the *Design by Contract* setting of Meyer [12], and in [10]).'

Sensing insecurity on the side of the footman, Alice put forward a bold suggestion: 'If that is so, wouldn't it be better to rename such conditions of the second sort to 'necessary pre-condition' or 'enabling condition'? Just imagine what a mess you will otherwise get when you associate one operation with two contracts? You will never know which pre-conditions you have to follow, and which ones you are free to ignore!'

As upset as he was about the complete lack of respect shown by this student, the footman couldn't help but secretly agree, Alice had a point. Since he was used to following rules, and not question them, he hastily changed the topic.

'You might wonder, then, how operations relate to the software programs you have read about in your book. In a program, we use *functions* or *procedures* to capture the operations (you might also call them *actions*) taking place in the real world. The effect of an operation can now, very concretely, be described using the variables in the program. The pre-condition will be a condition about the state variables, and the procedure inputs, whereas the post-condition will relate the values of state variables *before* calling the procedure with the values of the variables *after* the call (and with the inputs, and maybe the returned result). Such pre-/post-condition contracts were proposed by Meyer [12] in the context of *Design by Contract,* and represent the most common form of contract. Formal syntax for contracts of this form is provided by several programming and specification languages, including Eiffel, JML, and ACSL. The distinction between pre-conditions, which have to be ensured by the *caller* of an operation, and post-conditions, which are the responsibility of the *callee* and describe the result and effect of an operation, reflects the principle of modular design in which the different components of a program collaborate on clearly stated terms. Modular design enables us to use *Hoare logic* or *Dynamic logic* (and many other approaches) to verify rigorously that each procedure, and the program altogether, satisfies its contracts and therefore will be free of bugs!'

Having come to the end of this complicated explanation, the footman was very satisfied with himself, and expectantly he looked at Alice. But Alice had become confused long ago, and lost track of the many abbreviations used by the footman; all the while a new question had come into her mind and kept her occupied. Slowly, with a lot of pausing and thinking, she tried to explain her doubts.

'That is all fine, but must be an idea pursued by the academic gentry, which has never written a program longer than ten lines. A real procedure will do many things, it will read and update the values of hundreds of variables, and it can itself call many other procedures; how could one ever describe the complete effect of such a procedure using one post-condition? And what is worse, since the contract of a procedure has to capture also the effects of all internally called procedures, it will not at all be modular! If the procedure f calls procedure g, then to write the contract of f we already have to know what g does!'

The footman was taken even more aback than before by this blasphemy; and at the same time did not fully grasp what Alice meant. Rather stiffly he retorted: 'The intention of a contract is to specify the *overall effect* of an operation. Pre- and post-conditions are a means for describing how, upon a procedure call, the final values of the variables relate to their initial values. Contracts *enable* the modular design of a program (or of the world around us), since we can now talk about the effect of operations without having to consider their implementation. If a procedure calls another procedure, this is an implementation detail that the contract should abstract away from. We can build a system modularly by first introducing its operations, then equip each of the operations with a contract describing the intended behaviour, and later we can implement the operations (as procedures) independently of each other. A contract does not have to fully describe the effect of an operation either, it is enough if it mentions its essential features, and leaves away the unnecessary details.'

Alice, who had in the meanwhile ignored the rambling of the footman as she was absorbed in her own thoughts, continued her argument: 'But now imagine we want to write a contract for the task of organising Reiner's party. This will be a huge undertaking: we first have to select a good day and time, then send out all the invitations, then organise all the wine (and other less important things), then wait for all the people to arrive, then ask them to sit down, then pour wine, and so on! The party will of course contain many instances of operations like 'Take a seat,' 'Pour wine,' and 'Empty glass,', so that the pre- and post-condition of the party will repeat the pre-/post-conditions of those sub-operations many times over. The contract will be like a telescope that unfolds and becomes longer and longer, until nobody can see its beginning and end anymore! And think of all the empty bottles, how should we dispose of them?'

At this point, the footman decided that enough was enough, he had wasted already too much time with this lecture. Sitting down on the grass again, he merely mumbled: 'Go and talk to the Caterpillar! You will find it where the smoke is, and it will help you!' With this, he resumed his study of the clouds, leaving Alice alone in a rather confused state.

3 The State of the Caterpillar

As the footman had said, the Caterpillar was to be found exactly where the smoke was. It was sitting on top of a large mushroom, with its eyes closed and its arms folded, quietly smoking its long hookah. 'Good day!' Alice greeted the creature. 'I was told that you can tell me more about contracts. You see, I won't be invited to Reiner's party with only what I know now.' The Caterpillar did not move. It only released yet another cloud of smoke. 'If I want to specify,' Alice went on, 'that a computer program is to engage in certain sequences of operations, or that only certain sequences are allowed, how shall I do that? It seems to me cumbersome to use for this the pre- and post-conditions that the footman is so fond of.'

The Caterpillar finally opened its eyes. 'And because of this, little girl, you disturb my peace?' It cleared its throat and continued in a languid, sleepy voice. 'Well, let me think... This means that when executing the program, whether an operation is to be (or can be) executed next will depend on the history of operations performed up to that point. In that case one needs a notion of *state* as a means to organise the operations in the desired order, and so, it is only natural to use Finite Automata for this, or more generally, State Machines (see for instance [9] for an introduction). Due to their graphical representation, state machines can be a useful visual specification language. But there also exist other formalisms for specifying sequences of operations, such as process algebras (as used, e.g., in [14]) and grammars (see Section 4 below), and hierarchical variants of state machines such as Statecharts [7].'

Here, Alice raised her hand—upon which the Caterpillar raised its eyebrows. It was not very fond of being interrupted, but it allowed Alice to ask her question while it took a smoke. 'Can I use a state machine to specify that guests should only drink wine while seated? I wouldn't like them to get drunk and stumble over other guests, you know.' 'Oh, absolutely!' the Caterpillar replied. 'And here is how such a state machine could look.' And it drew, with the mouthpiece of the hookah, a picture on the surface of the mushroom (see Figure 1).

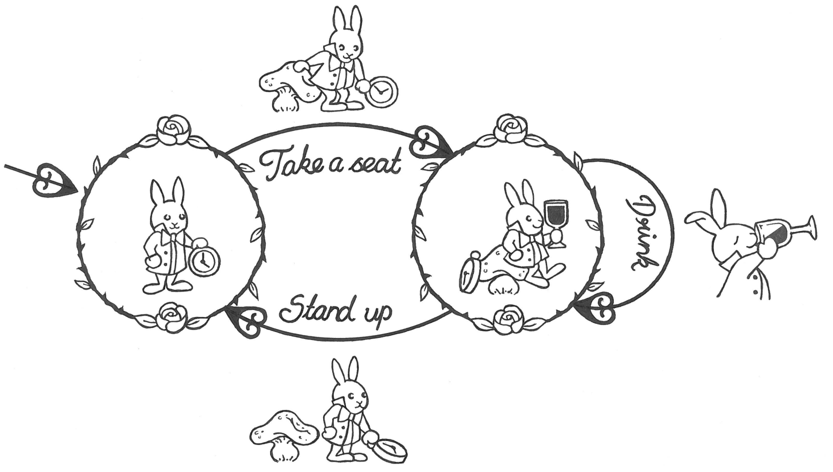

Fig. 1. A contract presented as a state machine, specifying that guests shall be sitting while drinking.

'How marvellous,' exclaimed Alice, 'that is so beautiful!' (Maybe even more so than the drawings of John Tenniel, she thought.) 'But can't this also be achieved with pre- and post-conditions as the footman wanted, instead of drawing circles and arrows?' 'One surely can, my child, one surely can,' replied the Caterpillar.

Here, the Caterpillar made a long pause, exhaling a cloud of smoke. 'You see, a picture is just so much easier to understand. But once you have drawn it, and you wish to use pre- and post-conditions, you simply have to introduce a new global variable[3] to represent the state of the state machine. This *state* variable is initialised at the start of program execution to the initial state of

[3] Or a so-called *ghost* variable, if your specification system supports them: these are variables that are not part of the program, but are used for specification purposes.

the state machine. And then, you have to encode every transition of the state machine into the contract of the procedure that implements the operation with which the transition is labelled. You can achieve this by adding a conjunct to the pre-condition, to capture when the operation is enabled, and a conjunct to the post-condition, to capture how the state variable is changed.'

'Hm,' Alice mumbled, writing something in her notebook. 'Would this then be a good contract for the drink operation?' And she showed her notes to the Caterpillar.

Operation:	Drink wine
Pre:	Person is sitting $\wedge$ Glass has wine
Post:	Person is sitting

'Yes, indeed,' replied the Caterpillar. But Alice was still wondering. 'If I understand correctly,' she said, 'the state machine is for the whole program and not just local to one procedure. And we also seem to assume that procedures and operations are the same thing. But how about *procedure contracts* that prescribe many operations to be performed, and in certain orders? Can I also use state machines to write such contracts?'

'Hm, hm.' The Caterpillar was pleased to have such an astute student, but was at the same time annoyed by her many questions. 'Yes. Such a situation would arise, for instance, when we have a *main* procedure, which is responsible to order the operations, and we want to write a contract for this procedure. You could then indeed use a state machine like the one in Figure 1 to formulate the contract. Or, if you prefer to write the contract in textual form, you could present it in some contract specification language such as ConSpec [1] (see Figure 2).'

```
CONTRACT STATE
   bool sitting = false;

AFTER take_a_seat()
PERFORM
   !sitting  -->  { sitting = true; }

AFTER drink()
PERFORM
   sitting  -->  { skip; }

AFTER stand_up()
PERFORM
   sitting  -->  { sitting = false; }
```

Fig. 2. A state machine in textual form.

'I see,' said Alice. (But the picture was indeed much nicer, she thought.) 'So `sitting` is the state variable, and is initialised to false in the beginning. And to take a seat, `sitting` must be false, but as a result of the operation it will become true?' 'Yes, indeed,' the Caterpillar replied. 'But what if the operation is attempted when `sitting` is true?' . 'Well, then the contract will be violated, Alice.' 'Oh, yes, I remember! So then, drinking is only allowed when `sitting` is true, and the operation does not change this.'

'And still' Alice kept wondering, 'instead of using a state machine to specify which other procedures are called and in what order, can't I simply specify the *overall effect* of calling the main procedure in terms of pre- and post-conditions? The footman was insisting on this, you see!'

'Hm, hm,' mumbled the Caterpillar. 'Yes, you can—and typically you would do exactly this. But sometimes you want your contract to be more abstract, as for instance when the very *purpose* of the procedure is to call certain other procedures in a certain order, regardless of what exactly they do (i.e., how they change the state). Another case is when the overall effect of calling the procedure simply cannot be captured in terms of changing the values of the variables—for example, when the effect concerns (interaction with) external entities.'

'Right. But I do have one last question,' said Alice. (Thank God, thought the Caterpillar, as it was beginning to get tired from all the profound questions that Alice was asking.) 'Will the pictures of the state machines always be small enough that we can draw them on a mushroom?' 'No, not always,' the Caterpillar replied. 'Sometimes the state machine may have a large or even an infinite state space. Such state machines are best represented *symbolically*, e.g., as a symbolic Kripke structure (see for instance [5]), by means of two predicates over states: a unary predicate $\mathsf{I}(s)$ that captures the *initial states*, and a binary predicate $\mathsf{T}(s, s')$ that captures the *transition relation.* This is for instance how one represents programs in the TLA framework [10] and the NuSMV model checker [4].

'And finally,' the Caterpillar continued after a brief pause, 'there is the question of how to formally *verify* that a given procedure meets its contract, when the latter is stated as a state machine.' Alice, who really wanted to make sure that Reiner's guests won't drink excessively and make fools of themselves at the party, looked very interested. 'Yes!' she exclaimed, 'I suppose, if we encoded the state machine with pre- and post-conditions, we could simply use procedure-modular deductive verification?'

'Indeed,' replied the Caterpillar. 'Another possibility would be to check statically that the state machine *simulates* the program's execution. Essentially, this amounts to conjoining the program with the state machine (i.e., forming their automata-theoretic product) and checking that the state machine never blocks (i.e., never has to take an operation that is not offered from its current state).' Here, Alice looked puzzled. 'Is this also a procedure-modular method?' she asked. But the Caterpillar could take no more. 'End of class!' it shouted grumpily, and with this it turned its back to Alice.

4 The March Hare Rules

After the lecture by the Caterpillar, Alice thought that, surely, she now knows almost everything there is to know about contracts. Satisfied, she started heading back to the party, but realised that it would probably be improper to show up without some sort of gift—a bottle of wine, or perhaps two. She was reminded of the March Hare, and the many parties with the Mad Hatter and the Dormouse. Maybe he would be able to help, Alice thought, and started heading towards his house to pay another visit.

When she arrived, she was immediately greeted by the March Hare, who invited her in. ‘My dear friend’, Alice started, ‘I am attending a party, and I am in need of wine, in haste.’ The March Hare led her down into the cellar, where, sure enough, there was a giant wine rack on one of the walls. ‘I have enough wine for any party’, the March Hare said, ‘but I cannot just give it away.’ ‘Please, I promise you, for any bottle I take, I will put another one back, as soon as the party is over,’ Alice replied.

Not convinced, the March Hare insisted that she make the promise formal, and started lecturing Alice about *context-free grammars* (CFGs). ‘Context-free languages are defined by context-free grammars. Such grammars consist of *production rules*, which are sentences over terminating and non-terminating symbols. The terminating symbols are symbols representing the basic units that we want to reason about. This could be the program states, for example, or some set of abstract actions. The non-terminating symbols refer to other production rules, possibly even recursively, and are substituted when evaluating what possible sequences may occur. For this reason, context-free grammars can express many properties that cannot be expressed in the other formalisms, some of which you have already seen.’

As the March Hare went on and on, Alice grew impatient. Eventually, she interrupted the monologue, and although not entirely sure she understood, presented to the March Hare the following contract, in the form of a CFG:

$$C \rightarrow \varepsilon \mid \text{TakeBottle}\ C\ \text{PutNewBottle}$$

She explained her reasoning. The contract is denoted by the production rule C. To make sure the production stops, either because party guests have had just enough wine, or, perhaps more likely, because the shelf has run out of bottles, we use the symbol ε to say that no action is taken. Alice’s contract to the March Hare, C, then says that either she doesn’t take any bottle at all, or at first she takes one bottle, in the end replaces it with a new bottle, and in between she again fulfils the same contract, resulting in the same amount of bottles being put back as was initially taken out.

‘That looks good,’ said the March Hare, ‘and I can agree to those terms. But before you go, let me first tell you a bit more about such contracts.’

The March Hare continued: ‘It is often the case when specifying sequences of operations, that contracts become unwieldy. Consider a procedure which performs its task by calling several other procedures, which all have their own contracts specifying the sequences of operations they will produce. The contract for the top-level procedure will then, naturally, consist of some combination of all those sequences produced by the called procedures. Instead of restating the full sequences, or the formulas representing them, it would be preferable if we had some way of directly referring to the contracts of the called procedures.

Now, for contracting purposes, the non-terminating symbols of the context-free grammars may also serve the purpose of referring to the contracts of other parties involved. For example, for two mutually recursive procedures a and b, with contracts given by non-terminating symbols A and B, respectively. Modelling only the events of calling and terminating the other procedure, we could give their contracts by the following grammar:

$$\begin{aligned} A &\to \varepsilon \mid \mathrm{call}(b)\ B\ \mathrm{term}(b) \\ B &\to \varepsilon \mid \mathrm{call}(a)\ A\ \mathrm{term}(a) \end{aligned}$$

We are thus able to specify contracts more concisely, when they depend on results of other procedures. You may already be familiar with the concept of function modularity for verification purposes (see Section 2), and by using CFGs in this way we achieve a similar modularity in the contracts themselves.'

'Okay,' said Alice.

'From an assume-guarantee viewpoint, the possible productions of terminating symbols of a contract are what is guaranteed, under the assumption that the other procedures whose contracts are referred to, produce what is specified by the grammar. In this way, assumptions are made explicit, whereas in other formalisms, such as pre- and post-conditions, similar assumptions implicitly exist on called procedures, but are never explicitly stated. A drawback, however, of directly referring to other contracts in this way, is that they are no longer independent of each other, and changes in one contract, as may happen regularly during development, will affect all contracts directly or indirectly referring to that contract.'

'Does this not also mean, then,' Alice remarked, 'that verifying the correctness of such contracts is not an easy task?'

'Why, yes...' the March Hare said, 'but I have some ideas.' Listening to the explanation, Alice learned of the concept of undecidability. Some problems are in general infeasible to solve, and for two languages defined by CFGs, deciding whether one is included in the other is precisely such a problem [9]. This did not stop the March Hare from fleshing out the idea. 'Let us say, that we are only interested in certain actions. If we ignore the data of the program, then the program could be translated, statement by statement, into a CFG producing all those sequences of actions which the program could possibly produce, and then some.' Alice looked confused, but the March Hare continued. 'You see, since we do not take any data into account, our analysis of the control-flow will be an *overapproximation*. But if the sequences produced by this grammar can also be produced by the CFG of the contract, then we can still be sure that the program satisfies the contract.'

‘But,’ objected Alice, ‘you just said a moment ago that we cannot decide whether one such language is a subset of another.’

‘While this will not be possible to verify every time, if we are lucky, often it will.’ The March Hare went on to explain that if we do not limit ourselves to automatic proofs, but allow humans (or other creatures) to interact with the prover, many more possibilities arise, and that there already are ideas proposed for solving the problem at hand in this way [16]. ‘Or...’ The March Hare looked deflated. ‘Or we restrict ourselves to simpler classes of languagues.’ The March Hare suggested the use of visibly pushdown languages [2]. For such languages, Alice soon learned, inclusion is always decidable in exponential time. ‘Then you will have your answer in the end, although it might take a very long time. Let’s just not restrict ourselves too much, or we will end up back at finite state-machines!’ The March Hare laughed, and ran away, leaving Alice alone in the wine cellar.

5 The Logic of the Mad Hatter

Alice, equipped with both knowledge and wine, again started making her way back to the party. As she was walking along a path in the forest, she happened upon the Mad Hatter. Alice started telling him about everything she had learned. The Mad Hatter soon stopped her, and said, ‘Yes, yes. That all seems very useful. But how would you go about specifying that if a bottle of wine has been emptied, eventually a new one will be brought to the table? Such a contract seems to me to be of utmost importance!’ Alice thought for a minute, but could not find an obvious answer in the logics she had gotten to know.

‘Or how about: there shall never be fewer than five full bottles of wine at each table?’ the Mad Hatter continued. ‘An even more important property!’

The Mad Hatter started rambling about this thing called Interval Temporal Logic (ITL) [6], and how it could be used to specify properties over intervals, or finite sequences of states. ‘At the *heart* of this, is an operation called *chop*, and much like when the Queen of Hearts applies it to heads and bodies, it separates an interval into two!’

The Mad Hatter continued ‘Now, if E represents the fact that a bottle has been emptied, and N that a new bottle is brought to the table, then the logical formula $\neg E \vee (E^\frown N)$ means that either no bottle is emptied, or it is, and if so, a new bottle is eventually brought to the table.’

Alice had to stop to think. She understood the connectives of negation and disjunction, since their meaning here was similar to how she had seen them used before. She recalled what the Mad Hatter had said about the chop operator—that it was a way of concatenating two separate intervals, such that the first interval ends where the second begins. She also recalled that atomic formulas are evaluated in the first state of an interval, and are true for the entire interval if they hold in this state.

‘I see!’ Alice exclaimed. ‘If the interval starts with an emptied bottle, we want there to be a second interval starting with a new bottle being brought, and this should happen when the first interval ends.' She thought a bit more. 'And since the intervals are finite, in particular the first one, the state containing the new bottle must eventually occur.'

'Correct. Let's now say, the state of our table consists of the number of wine bottles on it. We can represent this by the variable *bottles*. At any point in time, we thus want to assert that $bottles \geq 5$.'

Alice also soon understood, that by using the operator *, she could specify that an assertion shall repeatedly hold. 'So by writing F^*, I say that it should be possible to divide the interval into smaller parts, such that each subinterval ends in the state where the next one begins, and the formula F holds in each part, right?'

'That is correct. Now, let's see if can use this to specify what we want.'

'Oh, I know—I know!' Alice exclaimed. 'We simply write $(bottles \geq 5)^*$. That was too easy!'

'Aha, not quite,' the Mad Hatter grinned. 'Would your formula not hold for an interval where there are at least five bottles only in the first state?'

Alice thought for a bit, and realised her mistake. 'You are right, since one possible way to split the interval is into a single part.'

'You see, there is one more operator that we need to talk about, called *next*—and by the way, we will also need a formula that holds for all possible intervals, let's call it true. Now, about *next*, it is true if the formula that comes after holds in the second state. We can use this to reason about the length of intervals...' Alice was again becoming impatient—and it must have showed, because the Mad Hatter interrupted himself mid-sentence. 'But enough of that for now. In this case, all we need is a simple little trick. Instead of saying something shall always hold, we state that its negation must never hold. Using this equivalence, you should be able to specify the property with what you already know.'

'That is neat. How about this, then: $\neg(\mathsf{true}^\frown\neg(bottles \geq 5))$?' Alice asked. 'It should not be the case that eventually, there is fewer than five bottles of wine.'

The Mad Hatter nodded. 'Let me tell you one more thing, before you head off,' he said. 'Since you know about pre- and post-conditions, this might be familiar to you,' (see Section 2). The Mad Hatter explained that ITL could be used in a similar way, to get something more closely resembling a contract. 'As with other types of assertions, we would often like to specify that for our formula to apply, some pre-condition must be met. But for this pre-condition, we are not so interested in the full sequence of preceding states. Thus, in interval-based contracts, the pre-condition can be an assertion over singleton states, whereas the post-condition specifies the full traces to be produced.' The Mad Hatter explained that much like Hoare logic has been used to prove correctness of contracts in the style of pre- and post-conditions, extensions of it can be used to prove the correctness of contracts based on ITL [13].

As the Mad Hatter noticed that Alice was barely listening anymore, they said their farewells, and went their separate ways.

'And now it is high time for me to go to Reiner's party,' thought Alice. 'Enough of all these contract languages. And I really want to finally meet Reiner!'

She then saw the White Rabbit rush past her. 'Wait!' she shouted after it, 'I can't run as fast as you!' And they disappeared one after the other in the forest.

6 Epilogue

Alice was sound asleep on the bank, with the book [3] still in her hands, when her sister came. 'Alice, Alice, wake up, I brought you a marshmallow!' Alice promptly jumped on her feet. She then recalled her dream. 'Sister, dear—you will never believe what I just dreamt!' And she told her sister about Wineland and all the creatures she had met, about Reiner and the party she never got to.

'First, there was this footman, you know,' Alice started her story. 'He knew an awful lot about doing things. He said that to do something properly, you first need to know when you can do it (he called it a pre-condition), and then you need to know how things would change afterwards (he called it a post-condition). These two conditions he called a Contract. We talked a lot about these contracts. But I thought them rather limited. Then there was the grumpy Caterpillar, who was sitting on a mushroom and smoking a long hookah. We talked about contracts with states. It drew funny pictures on the mushroom, with circles and arrows, and called them State Machines. And then there was the March Hare. He told me about contracts written as grammars. But not the English grammar we learn about in school, you know. These grammars are called context-free, and with them you could talk about taking bottles and then returning them, one-for-one. And finally, there was the Mad Hatter, who was really quite mad, indeed! He told me about something very strange, he called it Interval Logic. He used some funny letters to write contracts. With them you could make sure that the guests at Reiner's party will always have enough to drink.'

Alice's sister looked rather perplexed. She didn't understand a word of what Alice was saying. And then she suddenly got very agitated. 'This is all very, very strange,' she said. 'While you were asleep, Alice, I heard this funny song on the wireless. It must have been about your dream! It went on like this:'

And if you go chasing rabbits
And you know you're going to fall,
Tell 'em a hookah smoking caterpillar
Has given you the call.
Call Alice
When she was just small.

'Yes, this is indeed very strange!' Alice exclaimed. 'Maybe I heard it too, from a distance, and that's why I dreamt all this? But after all,' she continued, thoughtfully, 'I am really happy that you woke me up. I should think that *applying* all these contracts for Reiner's party would have made a complete mess!'

And still, Alice wondered, in Wineland they all assumed that everything you do, you do in a sequence. But how would the contracts be if we also considered things that we do simultaneously? This, and many other questions crossed

Alice's mind. She had still to learn a lot about contracts, and such mysterious languages as Separation Logic [15], in which one can write contracts for concurrent programs. Alice had even heard rumours about contracts that were *smart,* although that was surely an exaggeration that the mad creatures in Wineland must have come up with.

References

1. Aktug, I., Naliuka, K.: ConSpec - A formal language for policy specification. Science of Computer Programming **74**(1-2), 2–12 (2008). https://doi.org/10.1016/j.scico.2008.09.004
2. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of ACM Symposium on Theory of Computing (STOC'04). p. 202–211. Association for Computing Machinery (2004). https://doi.org/10.1145/1007352.1007390
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach - Foreword by K. Rustan M. Leino, Lecture Notes in Computer Science, vol. 4334. Springer (2007). https://doi.org/10.1007/978-3-540-69061-0
4. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model verifier. In: Proceedings of Computer Aided Verification (CAV'99). Lecture Notes in Computer Science, vol. 1633, pp. 495–499. Springer (1999). https://doi.org/10.1007/3-540-48683-6_44
5. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
6. Della Monica, D., Goranko, V., Montanari, A., Sciavicco, G.: Interval temporal logics: a journey. Bulletin of the European Association for Theoretical Computer Science EATCS **105** (01 2011)
7. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259
9. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation, 3rd Edition. Pearson international edition, Addison-Wesley (2007)
10. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems **16**(3), 872–923 (1994). https://doi.org/10.1145/177492.177726
11. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: Jml reference manual (2008)
12. Meyer, B.: Applying "Design by Contract". IEEE Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
13. Nakata, K., Uustalu, T.: A Hoare logic for the coinductive trace-based big-step semantics of while. In: Programming Languages and Systems. pp. 488–506. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_26
14. Oortwijn, W., Gurov, D., Huisman, M.: Practical abstractions for automated verification of shared-memory concurrency. In: Proceedings of VMCAI 2020. Lecture Notes in Computer Science, vol. 11990, pp. 401–425. Springer (2020). https://doi.org/10.1007/978-3-030-39322-9_19

15. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of Logic in Computer Science (LICS'02). pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
16. Rot, J., Bonsangue, M., Rutten, J.: Proving language inclusion and equivalence by coinduction. Information and Computation **246**, 62–76 (2016). https://doi.org/10.1016/j.ic.2015.11.009
17. Wing, J.M.: A Two-Tiered Approach to Specifying Programs. Ph.D. thesis (1983), Technical Report TR-299