university of groningen / faculty of science and engineering

Imperative implementations of a public announcement logic model checker

Bachelor's Project Thesis

Gabriel Soicher, s3661806, g.o.soicher@student.rug.nl,
Supervisors: Dr B.R.M. Gattinger

Abstract: Contemporary model checkers for Public Announcement Logic (PAL) and Dynamic Epistemic Logic (DEL) are mostly written in Haskell, which is not usually seen as an easy to optimise language. Haskell is pure, lazy, and functional, and so this thesis investigates if there are any performance improvements to be had by writing an explicit model checker for PAL in an imperative language, and comparing that to both symbolic (SMCDEL, Gattinger (2019)) and explicit (DEMOS5, van Eijck (2014)) checkers written in Haskell. This research covers implementations in Python, Go, Java, and C++ - all of which involve parallelisation of model generation and of model solving, as well as some additional features compared to contemporary Haskell solutions. The main finding of this research is that parallelisation accounts for a large performance gain for larger models – the implementation of which is made easier due to the use of imperative languages – but that symbolic model checkers still outperform even these parallel explicit model checkers; explicit model checkers are still relevant, but only in those areas where a symbolic checker has not yet been developed.

1 Introduction

Contemporary model checkers for Public Announcement Logic (PAL) and Dynamic Epistemic Logic (DEL) are mostly written in Haskell, a pure, lazy, functional language (van Eijck, 2014) (Gattinger, 2019). Haskell syntax and semantics map very well to formal mathematical constructs, so it is often the natural choice.

The key features of Haskell that make it so appealing as a research language, and a language for logicians, can be at odds with performance oriented code. As a high-level language, it is natural to expect that some performance may be lost due to the abstraction of some implementation details. More importantly for us, however, is Haskell's lazy, pure nature. Laziness means that instead of immediately evaluating the entirety of an expression, Haskell programs are designed to only consume things as needed. This can have interesting performance implications. Likewise, the 'problem' with pureness for us is simply that it refers to a general lack of state - this can also make writing performant programs harder.

In this project, we implement a model checker for public announcement logic (PAL) in four imperative languages to determine whether or not there is any performance to be gained. All of these languages have unique features that may lead to better performance. The initial model checker was written in Python, a high-level garbage collected language that can, depending on the workload, perform quite well. This was done largely because Python is very easy to write, and so allows for quick prototyping. Implementations were then developed for Go, Java, and C++.

The main hopes for increased performance rested on the fact that memory and object access patterns can be explicitly declared in an imperative language. By definition, an imperative program is more concerned with the mechanics of the operations being performed than a declarative language like Haskell. The additional benefit of many of the implementations is easy access to shared memory concurrency - see Section 5 for more details.

This thesis is structured as follows: Section 2 introduces the logical foundations for PAL and Kripke Models, Section 3 describes the model checking process for PAL models, Section 4 details the language agnostic design for my PAL model checker, Section 5 looks at the language specific implementations of the model checker, and the results are then presented and discussed in the remaining

sections.

1.1 Software Repository

The repository containing this project's software is available on Zenodo at `https://doi.org/10.5281/zenodo.5078120` or alternatively on Github at `https://github.com/Lunrtick/palchecker/tree/master`.

2 Logical Foundations

Public announcement logic (PAL) is the main logic used in this project. PAL is a dynamic extension of epistemic logic, which is itself an extension of propositional logic. For completeness, we will include the definitions for all of these logics here.

Definition 2.1 (Propositional Logic)**.** Propositional logic is sometimes called zeroth order logic. It can be seen as a foundational logic upon which more complicated logics are built, and it contains only boolean propositions, and connections between them using logical connectives. The boolean language $\mathcal{L}_B(V)$ is given in Backus-Naur form (BNF), with a vocabulary V (where $\varphi \in V$) as

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \tag{2.1}$$

A world is a set of propositions $w \subset V$. We interpret a world as a set of true propositions - at a given world, all $p \in w$ are true, and no other propositions hold. The semantics of $\mathcal{L}_B(V)$ for these worlds are

- $w \models \top$ always holds
- $w \models p$ iff $p \in w$
- $w \models \neg p$ iff $w \not\models p$
- $w \models \varphi \wedge \psi$ iff $w \models \varphi$ and $w \models \psi$
- $w \models \varphi \vee \psi$ iff $w \models \varphi$ or $w \models \psi$. Disjunction as given here is represented explicitly in the model checker, but can be defined as $\neg(\neg\varphi \wedge \neg\psi)$.

Definition 2.2 (Epistemic Logic)**.** Epistemic logic is built on top of propositional logic, and adds the concept of agents (the set of whom is denoted I) and the knowledge operator $K_i\varphi$, where $i \in I$. This operator can be understood as saying "agent i knows that φ". A similar operator, which is used to say "agent i knows *whether* φ", is defined as $K_i\varphi \vee K_i\neg\varphi$ but often written as $K_i^?\varphi$.

The semantics of these operators are

- $w \models K_i\varphi$ iff φ is true at every world reachable by i from w.
- $w \models K_i^?\varphi$ iff φ has the same truth value at every world reachable by i from w.

The specifics of reachability will be explained in more detail in the discussion of Kripke Models.

This is where the idea of worlds becomes interesting. Imagine two agents, Alfred and Ben, who were baking cookies. Ben leaves the kitchen, and Alfred is still in the kitchen. Let us imagine that there are two propositions, a means that the cookies have burnt, and b means that Alfred is smoking a cigar. These are the only two causes of smoke in this universe. We then have 4 possible worlds $\{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. Ben then notices that there is smoke coming from the kitchen - he also knows that Alfred is in the kitchen. Ben **knows that** Alfred **knows whether** the cookies have burnt, while Ben does not actually know what exactly has transpired himself.

This is often stated as an inability for agents to distinguish between worlds - Ben cannot see enough to know what has happened, besides knowing that there is smoke. 3 of the worlds are equally possible to him, while only the world $\emptyset$ is not possible to him. This can be visualised as a Kripke Model in Figure 2.1, where the "B" on the lines shows that it is a relationship for Ben, and not Alfred (who would have his relationships denoted by an A). The reflexive relations are omitted, and would be arrows from each world to itself for each of the agents. These are not always implied, but S5 is reflexive, and this project evaluates PAL under the S5 system. S5 is also symmetric, so these relations are not directed - a relationship from w_1 to w_2 implies a relationship from w_2 to w_1.

Definition 2.3 (Kripke Model)**.** With $\mathcal{P}$ a nonempty set of propositions, and $\mathcal{A}$ a nonempty set of agents, a Kripke Model M is a structure $\langle W, R, v\rangle$ where

- $W \neq \emptyset$ is the set of worlds

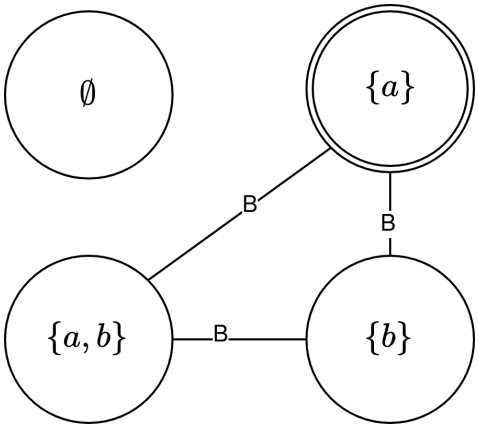


Figure 2.1: A Kripke Model for the Alfred and Ben smoke example

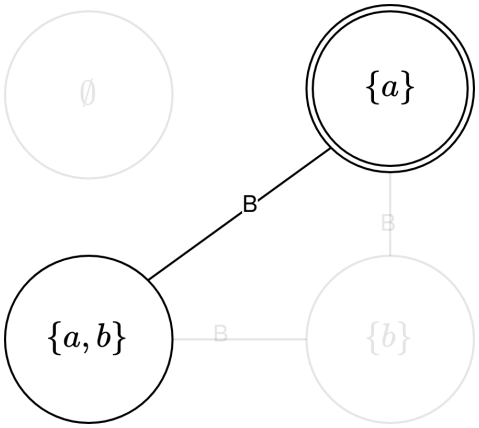


Figure 2.2: A Kripke Model for the Alfred and Ben smoke example, after the announcement by Albert that the cookies have burnt

- $R \subseteq (W \times W)$ is a binary relation for an agent between 2 worlds. This is usually written as the relation wR_ax, for agent a between worlds w and x, and shows that relative to w, a considers x to be a possible world as well.
- $v : (W \times P) \to \{0, 1\}$ is the valuation function. This is usually written as $v_w(p)$, telling us whether or not p is true at w.

The double ringed world in Figure 2.1 indicates the real world (i.e. the one that describes the objective happenings). In this scenario, we see that Ben cannot distinguish between the worlds $\{\{a\}, \{b\}, \{a, b\}\}$, he does not know what caused the smoke in the kitchen. Alfred does know, and so can distinguish between all of the worlds. Reachability is the term usually used to describe "following a line" - if the line between two worlds shows that the agent cannot tell them apart, we say that the agent can *reach* the other world by following that line. Using Definition 2.2, we can evaluate the validity of our earlier statement that Ben **knows that** Alfred **knows whether** the cookies have burnt. We first write this as $K_BK_A^?a$. Evaluating this from the inside out, $K_A^?a$ is valid in our model, because a has the same truth value at every world reachable by A from every world in our model. In all cases, the only world A can reach is the same world, so the value of a must be the same. Given this fact, we can say that $K_BK_A^?a$ is also valid, because at every world in our model, Ben cannot reach a world where Alfred is uncertain about the truthiness of a.

Definition 2.4 (Public Announcement Logic)**.** Public announcement logic is an extension of epistemic logic, adding the idea of public announcements, and thus allowing the modelling of change within the system. The dynamic operator $[!\psi]\varphi$ is added to Definition 2.2, meaning that after ψ is publicly and truthfully announced, φ holds. Again, a dual operator exists, and is defined as $\neg[!\psi]\neg\varphi$. This is saying that ψ is true, and after announcing this fact, φ holds. This dual operator is often written as $\langle!\psi\rangle\varphi$.

As described in Definition 2.4, we can now make public announcements that actually change the model. As an example, consider the announcement by Albert that "the cookies have burnt". This announcement would change the model by removing the two worlds where this proposition was not true, leaving us with the model in Figure 2.2. Now, there are only two possible worlds. One, in which only the cookies have burnt, and another, where Albert is also smoking a cigar. Despite knowing that the cookies have burnt (a), Ben still does not know whether or not Albert is smoking a cigar (b), which we can write as $[!a]\neg K_B^?b$.

2.1 The Muddy Children Problem

The muddy children problem is a useful benchmark for model checkers because it has many worlds - the number of worlds is 2^n where n is the number of children - and because it involves repeated public announcements that require exhaustive checking. The problem involves n children, $n-1$ of whom have mud on their faces. Every child can observe every other child, but not themselves. The children's father arrives, and says "at least one of you has mud on your forehead". The father then repeatedly asks "do any of you know whether you have mud on your own forehead?". If we assume that all the children are perfect logicians, and are truthful, and that they must answer simultaneously, then after $n-1$ times that the father has asked the question, they will all be able to answer (Moses, Vardi, Fagin, and Halpern, 1995).

3 Model Checking

Model checking in general allows us to answer questions about a particular model. In our case, there are 3 questions we want to answer (φ is a PAL formula).

1. In what worlds from our model does φ hold?
2. Is φ valid in our model? (i.e. true at every world in our model)
3. At a given world w in our model M, does $M, w \models \varphi$ hold?

Asking whether $M, w \models \varphi$ holds is the most common question, and is also the most specific. There are two distinct ways of finding the answers to these questions. These are explicit and symbolic model checking. The model checker implemented here is explicit - we actually create (in memory) all possible worlds, explicitly check every piece of φ to answer the questions. This has the drawback of being very slow, as the number of worlds m may grow as large as $\Theta(2^{|V|})$ where V is our vocabulary. Symbolic model checking, on the other hand, does not explicitly enumerate all possible worlds, and has been shown to check models at least a few orders of magnitude faster (Burch, Clarke, McMillan, Dill, and Hwang, 1992) (Gattinger, 2018). The main drawbacks are complexity and that it is not yet possible to check models in all logics, for example neighbourhood models. It is, however, currently possible to check PAL models.

3.1 Explicit Kripke Models

The internal representation of a Kripke model is the first hurdle to overcome when designing an explicit model checker. The details are fairly implementation specific, but broadly speaking we considered two approaches.

3.1.1 Object Oriented Models

Each world is an object, which holds some true propositions, and references to related worlds for each agent in our model. This is a natural way to describe Kripke models, and it allows recursive exploration - each world is a node in a directed graph, and the edges are described by the relationships at each world.

This approach does open up room for inefficiency at implementation time though - worlds could be fragmented in memory, leading to some overhead in exploring the graph.

3.1.2 Data Arrays

A world is simply a collection of true propositions. In memory, this could look like a two dimensional array, where each index holds the set of true propositions. We would then represent the relations for each world as a separate array for each agent; again, a two dimensional array, with each index holding references to the corresponding worlds.

Mathematically, this is equivalent to the object oriented approach, but it may have performance benefits.

All of the implementations created as part of this project use the object oriented approach. The data types used to represent things like propositions and formulae are also important to consider.

3.2 Propositions

The only real requirement here is that propositions are distinguishable from one another. SMCDEL allowed only integers for propositions, whereas the design put forward here can allow arbitrary strings

for both agents and propositions. There should not be any noticeable performance differences between strings and integers at the model sizes we are using - strings are usually allocated to the heap, so we would only be dealing with pointers that are similarly sized to integers. The only place this may have an impact would be in determining whether or not a string is a member of a set, which could scale with the length of the string.

3.3 Formulae

Formulae are represented as a tree structure, where each node has a type, and contains the required information for that formula. The types cover the full range of Public Announcement Logic. The recursive representation of formulae leads to a recursive implementation of a truth checking function. Again, this has performance implications, as not all languages handle recursion equally well.

4 Design

In order to facilitate implementation in multiple languages, a language agnostic design was drawn up. This is made easier by the fact that all of the implementation languages are imperative and capable of some form of object oriented programming; they do all have different idiomatic styles and benefits though.

The high level algorithm for model checking PAL is as follows

1. Parse a model definition file into a vocabulary V, state law s, observation list for each agent, and set of queries Q. Rather than invent a new format, this design uses the same one that SMCDEL does.

2. Using these values, generate a model M, with worlds as the set $\{w \mid w \in \mathcal{P}(V), w \models s\}$ explicitly created in memory, and the relationships assigned according to the given observation list.

3. Answer every query in Q using the generated explicit model.

4.1 Parsing

In order to facilitate parsing in multiple languages, and avoid having to re-implement a parser each time, ANTLR was used (Parr, Harwell, Vergnaud, Boyer, Lischke, McLaughlin, and Sisson, 2021). ANTLR is an $LL(*)$ parser that supports generating code in all the languages tested in this project, though only the Python and Java implementations were completed. The benchmark results presented in Section 6 do not include parsing however.

The grammar was constructed to match that used in SMCDEL, though some parts of this were unnecessary or more complicated than may have otherwise been required. The state law, for example, is necessary for symbolic model checkers. For explicit model checkers, it serves merely as a filter, removing any worlds where that law is invalid. It would be equivalent, for example, to simply state which worlds are allowed directly. This could of course be seen as pre-generation of the model, because the checker would read in the already constructed list of worlds. It is, however, a seemingly arbitrary choice; if the model definition included an already constructed list of worlds, then measuring the parsing time for each implementation would be more important.

4.2 Model Generation

Model generation is a fairly straightforward process. We take a set of the given vocabulary V, and generate the powerset of V, $\mathcal{P}(V)$. For each world $w \in \mathcal{P}(V)$, we need to confirm whether or not it is included according to whether the state law is true at that world. It is important to note that we generate $2^{|\mathcal{P}(V)|}$ worlds, which is part of what makes explicit model checking so expensive.

After we have the set of worlds allowed by our state law, we need to assign the relationships between all of the worlds. This is done according to the observations given in our model. These take the form

<agent>: proposition1, proposition2, ...

where each proposition in the list is a proposition that the agent can observe. In the muddy children example described in Subsection 2.1 , where the children have mud on their faces but cannot see their own face, every agent can observe every other

a: b_is_dirty , c_is_dirty ,
b: a_is_dirty , c_is_dirty ,
c: a_is_dirty , b_is_dirty ,

Figure 4.1: Observations in the muddy children problem for 3 children

```
vocabulary = set(variables)
for w in model.worlds:
  # obs are the agent's
  # observations, as in Figure 4.1
  for agent, obs in observations:
    # This is the set difference,
    i.e. {1,2} - {1} = {2}
    unobserved = vocabulary
        .difference(obs)
    observably_true =
        w.true_propositions
        .difference(unobserved)
    for w1 in model.worlds:
      if (observably_true ==
        w1.true_propositions
        .difference(unobserved)):
         w.relations[agent].append(w1)
```

Figure 4.2: Relationship assignments as written in Python

agent, but not themselves. With three children, the observations are given in Figure 4.1.

This means that we need to assign relationships for every agent in every world, where everything that is either true or false in one world and is observable to the agent also must also have the same truth value in another related world.

The process for assigning these relationships is given in Figure 4.2, as it was written in Python.

Because this process is independent for each world, it is trivial to complete in parallel. We can assign a world (or more likely, a chunk of worlds) to separate processes, which all operate using shared memory on the list of worlds.

4.3 Answering Queries

The three query types are explained in Section 3, and are referred to in the program as

1. WHERE
2. VALID
3. TRUE

respectively. TRUE is the least expensive query to answer, while VALID requires us to evaluate (in the worst case) a given formula at every world in our model. WHERE always requires us to evaluate the given formula at every world in the model.

TRUE is answered by first finding the world in our model with exactly the given true propositions. The given formula is then evaluated at that world (other worlds may be involved through relationships).

VALID is answered by evaluating the given formula at every world in the model. This can be done in parallel, and it can be short-circuited; if the formula in question is ever false at a given world, the answer will be false. No more work needs to be done in that case.

WHERE is answered by evaluating the given formula at every world, and returning all worlds where the formula is true. This can also be done in parallel, but cannot be short-circuited.

It is important to note that parallelisation is only done at the query level - the evaluation of a formula at a given world is started in a separate thread for each world, but checking of conjucts, disjuncts, and other connectives is not done in parallel. This is a deliberate design choice - connective parallelisation was attempted, but the recursive design means we quickly exhaust available cores (on the testing machines at least) and actually slow down the execution time.

5 Implementations

This project involved 4 concrete implementations of the abstract PAL model checker, in Python, Go, Java, and C++.

5.1 Python

Python is usually seen a poor choice for CPU intensive workloads, unless it is simply used as a wrapper for native code. This implementation was done in pure Python. There are two major reasons for Python's poor performance. These are the Global

Interpreter Lock (GIL) and the high overhead when calling Python functions.

The GIL is an implementation detail of the de-facto standard CPython interpreter (Python Software Foundation, 2021). It prevents two or more threads from accessing Python objects at the same time, and means that shared-memory parallelism is impossible using CPython (there are some caveats to this statement, but it is generally true). This means that it is not useful for the Python implementation to be parallelised.

The second, possibly more impactful issue, is that Python does a lot of work when a function is called (Nunez-Iglesias, 2015). This is caused largely by its dynamic nature - the Python interpreter doesn't statically analyse or optimise function calls. The interpreter must always go through steps to determine what function is being called, as even calling a function can change details about that function. This was tested empirically when an early version of the Python implementation used method jump tables to emulate the **switch** statement from C-like languages as in Figure 5.1. The Python emulation of this construct is shown in Figure 5.2, which involves calling the corresponding anonymous function according to the formula's type. Changing this implementation to an if−else chain almost halved the running time of the program. This can also be seen in Section 6, where Python is the only language to spend the vast majority of its time on solving the model, rather than generating it.

```
switch (formula.type) {
    case Proposition:
        return world
            .true_propositions
            .contains(formula.prop);
    // other cases here
}
```

Figure 5.1: A C-like switch statement for formula handling

```
return {
    Proposition: lambda f:
        f.prop in world.true_propositions,
    # other cases here
}[formula.type](formula)
```

Figure 5.2: A Python dictionary based switch construct

This overhead does not pair well with the recursive design of the model checker, as each step in the checking process results in at least one extra function call.

Python's dynamic nature makes implementation straightforward, but without dropping to native code, the execution time was simply not fast enough to compare favourably with the existing Haskell libraries.

5.2 Go

Go is a C-like language with garbage collection and builtin concurrency primitives. The promise of near-native performance with easy to use concurrency seemed to be well suited for model checking. There were three major issues holding back this implementation, however.

Goroutines, Go's lightweight threading primitive, all have growable stacks that start at a relatively small 2kb (The Go Team, 2014). Go is intended for use in environments that need to wait for IO or other asynchronous tasks, and Goroutines are intended to be very cheap to create. They are not so effective in a CPU intensive context, and especially so for recursive algorithms that likely necessitate a lot of stack growth. Each time the stack size is exceeded, a new allocation will be required, and all the existing stack data needs to be copied.

The second issue is the garbage collector. About a third of the execution time for larger amounts of children was spent on garbage collection. This is a big pain point in the model generation phase, where new **Map**s need to be created for each world.

Finally, and perhaps most importantly, Go has no native **Set** data type. It is possible to use an empty **Map**, but evaluation of performance graphs - an excerpt of which is shown in Figure 5.3 - indicated that a significant portion of the program's ex-

ecution time was spent on accessing values in these **Map**s.

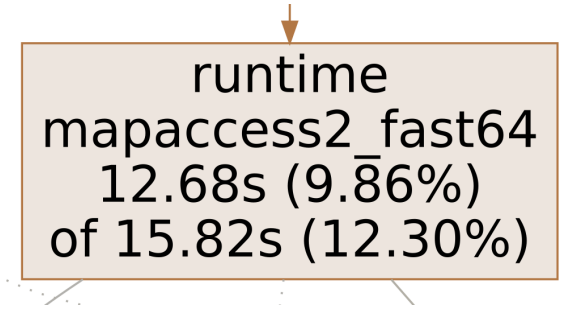


Figure 5.3: Results from a CPU profile graph showing Go's slow map access

5.3 Java

Java is usually used with the JVM, where Java is JIT (Just in Time) compiled according to runtime usage. This makes it a compelling candidate for many problems, as it is often possible to meet or exceed native code performance while being (subjectively) easier to write.

This is particularly noticeable when using Java's **Streams** API. This API exposes a **parallelStream** method, which lets you run a function in parallel over a given container's (e.g. a list) elements. This was used both for model generation and for answering WHERE and VALID queries, and basically involves the automatic creation of a thread pool in which an arbitrary function was run. This is possibly the simplest way to achieve shared-memory parallelism in any of the implementation languages.

Additionally, there are extensive libraries with very good performance. Google's Guava library provided a **SetView**, which is returned when calculating the difference between two sets (and elsewhere). A **SetView** doesn't create a new **Set**, and so saves both the Garbage Collector and some CPU time. Simply using this **SetView** instead of creating a new set resulted in a significant speedup for the model checker during the model generation phase.

5.4 C++

C++ is the only language used here which doesn't have a garbage collector. It is often regarded as the fastest language for general purpose computing. This of course comes at the cost of being (again, subjectively) more time consuming to write, and also with the danger of writing non-idiomatic code that doesn't get optimised as one might expect. I should disclose that I am not an extremely experienced C++ developer, and so it is quite likely that I have not done the language justice in these benchmarks.

I used the OpenMP compiler extension for parallelisation, which allows for very specific descriptions of exactly what memory is shared between threads, and also how work is allocated across those threads. Overall, the C++ implementation took by far the longest to write.

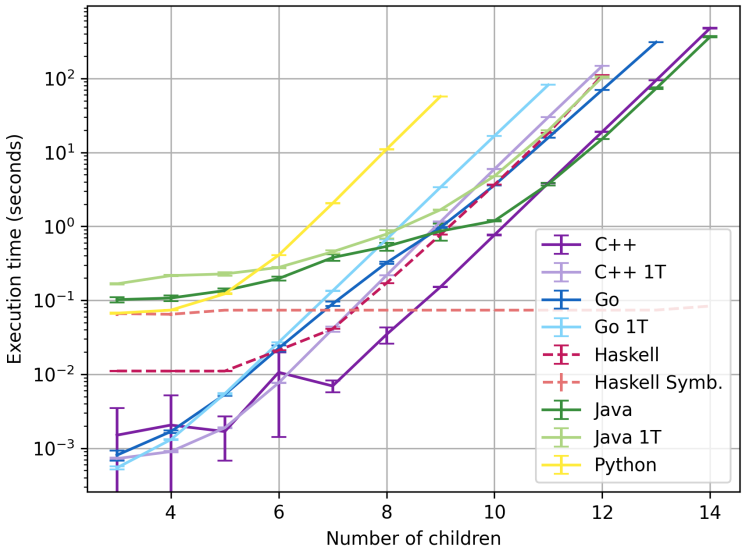


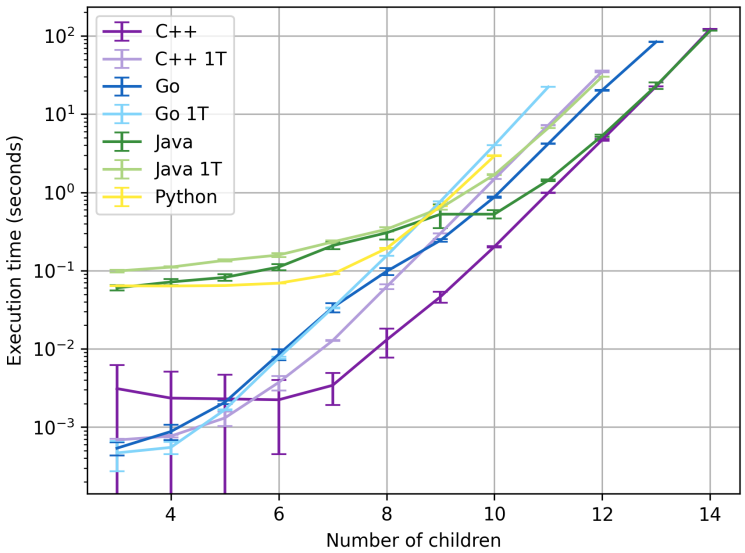
Figure 6.1: Execution time of different implementations on the muddy children benchmark (error bars show std. dev.)

6 Results

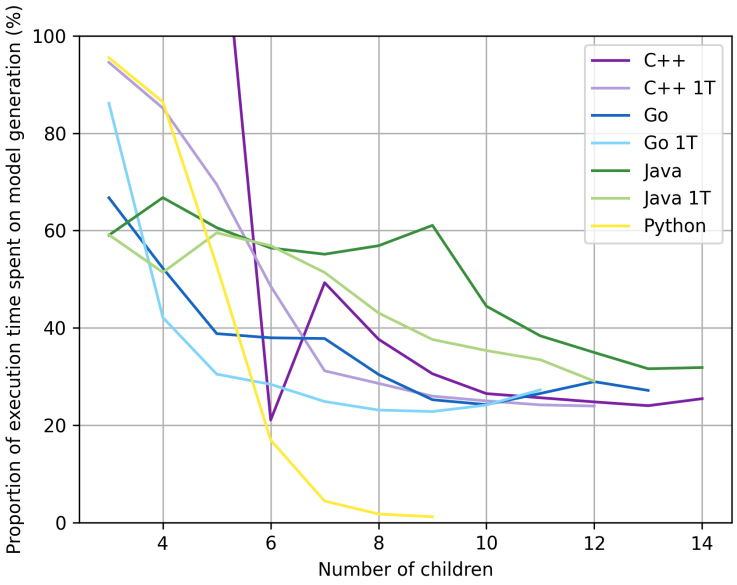
All benchmarks were run on a Ryzen 3600 (12 thread CPU) with 16Gb of DDR4 3600MHz RAM, using the muddy children example described in Subsection 2.1. Each implementation was packaged in a Docker container, and benchmarked inside the container using Hyperfine (to avoid measuring container startup time) (Peter, 2020). The single threaded benchmarks (and the Python one, which is implicitly single threaded) were all conducted using exactly the same programs, but used Docker's cpu limiting features to run on only a single core. They also utilised CPU pinning, to prevent the Docker runtime from scheduling work equivalent to a single core across multiple actual cores. This was done using the flags −−cpus 1 −−cpuset−cpus 1. Without pinning the CPU, the running time could be nearly double for single threaded programs.

Figure 6.1 shows the execution time for the various implementations in seconds, according to the number of children in the benchmark. The "1T" in the legend denotes the single threaded editions of the programs. It should be noted again that the model size grows as 2^n where n is the number of children. The first point to note is how quickly each implementation runs for small models. The start-up time for C++ and Go is very small, and Haskell (which also compiles to native code) also starts relatively fast. Python and Java, which run as bytecode through an interpreter and virtual machine respectively, start the slowest out of all the models. Symbolic SMCDEL ("Haskell Symb." on the graph) is also quite slow to start. Python is clearly the slowest of all the implementations, taking nearly $10^2 s$ for 9 children. The reference Haskell implementation ("Haskell" on the graph) is quite fast considering that it is single threaded, eventually running in about the same time as the single threaded Java implementation, which was the fastest of all the imperative single threaded programs. Java and C++ were the fastest imperative implementations, with C++ taking the crown for models smaller than 11 children, and Java for larger models (but only just). Symbolic SMCDEL ("Haskell Symb." on the graph) is clearly the fastest for larger models, as it runs in almost the same time for all tested model sizes.

Figure 6.2a shows the execution time in seconds, but doesn't measure the time taken to solve the model. I implemented a flag in all of the impera-



(a) Execution time of different implementations on the muddy children benchmark, measuring only the time taken to generate the model (error bars show std. dev.)



(b) Relative time spent generating the model versus solving the model (each measured over separate runs)

Figure 6.2: Model Generation Results

tive programs so that they simply ceased execution after the model was generated, but I did not do this for either of the Haskell programs (this also is not a meaningful measurement for SMCDEL, as it is a symbolic checker). This meant that I could measure the results using Hyperfine, but did also mean that these results are from different runs than those shown in Figure 6.1. The trends here in Figure 6.2a are broadly similar to those seen in Figure 6.1, Java and C++ are the two fasted programs for larger models, C++ and Go start the fasted, and Java and Python start the slowest. Python does actually generate the model in a fairly competitive time, seemingly running slightly faster than single threaded Go.

Figure 6.2b shows the relative time spent generating the model versus solving it for increasing model sizes. Again, the measurements for model generation and solving times were done on separate runs - the high standard deviation for C++ for small models evident in both Figures 6.1 and 6.2a accounts for the greater than 100% proportion seen in Figure 6.2b. Most of the implementations trend towards spending about 30% of their time on model generation. Python is the exception, and ends up spending almost all of its time on answering queries.

7 Discussion

It is clear from the results in Figure 6.1 that there are two main contenders for the fastest explicit model checker. These are the multi-threaded C++ and Java implementations. For smaller models, the relatively faster start-up times of C++ may be more desirable (though both run in much less than 1 second). As the models increase in size, however, the Java implementation supersedes C++.

Without the benefit of multiple threads, the C++ version isn't quite as fast as the reference implementation in Haskell. The Java version is about as fast for larger models. This may be somewhat surprising, as it was expected that there would be some performance gains to be had simply by switching to an imperative language. I suspect that this performance deficit for the imperative languages was largely due to the fact that they were not explicitly single-threaded programs. All the code to handle thread management and synchronisation is not free, but this setup cost would usually be a small price to pay for the speed increase that multiple threads provide. By leaving this multi-threading capability out, I expect that these imperative programs would be more competitive against Haskell.

The recursive design of this solver may also be a factor in Haskell's excellent single-threaded performance. Haskell is very well optimized for recursive solutions because there isn't really another way of expressing programs in Haskell. Languages like Java and C++ may not be able to perform some optimizations, like tail-call elimination, which prevents the creation of another stack frame when the recursive call is the last call in a function (Madsen, Zarifi, and Lhoták, 2018).

Multi-threading support does however appear to be more accessible to Java, Go, and C++, at least compared to Haskell. The benefits are clear - Java and C++ can solve a given model almost an order of magnitude faster than DemoS5 (the explicit Haskell program).

7.1 Problems and Challenges

Though every software project is mostly filled with challenges and solutions, these are some of the more serious ones that I encountered.

The parsing solution used in SMCDEL allowed for left-recursive structures. This was an issue specifically around disjunction and conjunction, where a model description file containing $a\&b\&c$ was parsed correctly in SMCDEL, but not so easily using ANTLR. Circumventing this limitation is possible, but as parsing was not a core piece of the project I simply allowed for only binary relations with the & and | characters - three or more con- or disjunctions were only allowed to be written as $AND(a, b, c)$ or $OR(a, b, c)$.

Another challenge was the elimination of worlds due to public announcements. This involved restricting the worlds in the model, but could not involve mutation of those worlds directly in memory (at least without copying) due to the multiple parallel threads that may be accessing those worlds at any given time. The solution I used was to pass a set containing the eliminated worlds (either as pointers to those worlds, or as the id of the eliminated world) as a parameter to my recursive functions. The stack-like nature of recursive calls meant

that these eliminated worlds only lived as long as necessary, but did also mean that whenever a world was accessed I needed to determine whether or not it had been eliminated.

The other problem that I encountered was around consistent benchmarking and easy reproducability of my findings. Using 5 different languages, each with unique build tools and dependency requirements, meant that the usual process of cloning a repository and following some steps or running a script was not likely to work. To circumvent this, I used Docker containers, which allow for a declarative setup of arbitrary environments. Initially I ran these containers as if they were my executables, but the start-up time of a Docker container is non-negligible. This meant that I had to put my benchmarking program inside each of the containers, and run the benchmarks there. The other benefit of this approach was Docker's tools for restricting CPU access, which made it easy to test constrained single threaded versions of my programs. The process for reproducing this research now only depends on Docker and Make (and Python + Matplotlib if plotting is required).

Finally, there are the broader conceptual problems. PAL is a very simplified version of DEL, so supporting more complicated operations like group announcements could have have implications for the performance rankings obtained here. The muddy children problem, as benchmarked here, also focuses on a particular worst case scenario where every agent can only observe all others - problems like the drinking logicians, where each can only observe themselves, might also have vastly different performance characteristics.

7.2 Further Research

Now that the benefits of parallelisation have been established, further research can be done into running DEMOS5 in parallel. This is traditionally a strong area for functional languages, which by design do not allow for shared mutable state. Haskell's performance on a single thread shows a lot of promise.

It would also be interesting to examine the performance of symbolic model checkers written in other languages than Haskell.

8 Conclusions

Two of the four imperative implementations (Java and C++) of the PAL model checker described in Section 4 showed large performance gains over the incumbent DEMOS5 Haskell implementation, and so it is possible to conclude that the DEMOS5 implementation is not optimal for larger models. It is also clear that a large part of this performance gain is due to parallelisation - future research into parallelising this problem in Haskell is certainly warranted. Overall, however, the clearest finding is that symbolic model checkers like SMCDEL are so much faster than explicit model checkers; explicit model checkers are still relevant, but only in those areas where a symbolic checker has not yet been developed.

References

Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

Malvin Gattinger. *New Directions in Model Checking Dynamic Epistemic Logic*. PhD thesis, University of Amsterdam, 2018. URL `https://malv.in/phdthesis`.

Malvin Gattinger. jrclogic/smcdel: v1.1.0, December 2019. URL `https://doi.org/10.5281/zenodo.3568325`.

Magnus Madsen, Ramin Zarifi, and Ondřej Lhoták. Tail call elimination and data representation for functional languages on the java virtual machine. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 139–150, 2018.

Yoram Moses, M Vardi, Ronald Fagin, and J Halpern. *Reasoning About Knowledge*, volume 4. MIT Press Cambridge, MA, 1995. ISBN 9780262061629.

Juan Nunez-Iglesias. Big data in little laptop: a streaming story in python, August 2015. URL `https://www.euroscipy.org/2015/schedule/presentation/5/`.

Terence Parr, Sam Harwell, Eric Vergnaud, Peter Boyer, Mike Lischke, Dan McLaughlin, and David Sisson. antlr/antlr4: v4.9.2, March 2021. URL https://github.com/antlr/antlr4/tree/4.9.2.

David Peter. sharkdp/hyperfine: v1.11.0, October 2020. URL https://github.com/sharkdp/hyperfine/tree/v1.11.0.

Python Software Foundation. Initialization, finalization, and threads. *Python Documentation*, June 2021. URL https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock.

The Go Team. Go 1.4 release notes, December 2014. URL https://golang.org/doc/go1.4#runtime.

Jan van Eijck. DEMO S5, March 2014. URL https://staff.fnwi.uva.nl/d.j.n.vaneijck2/software/demo_s5/.