Converting Modest Models to Efficient Code

Leo Heyns
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
l.heyns@student.utwente.nl

ABSTRACT

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model [1]. The Modest Toolset can be used to study the reliability and performance of systems through model checking. One of the features in the Modest Toolset is converting the models to a code representation, which allows easier writing of algorithms using those models. This feature is currently used in a teaching setting, the generated code facilitates interacting with models such that the students can write their own model checking algorithms. The current implementation generates slow code and is limited to only generating Python. This paper shows methods to make the code generated by the Modest Toolset faster, which allows more complex models to be checked, and to allow it to generate models in more programming languages, which makes creating model checking algorithms accessible to a wider public; it will then compare the performance of model checking in these different programming languages.

1. INTRODUCTION

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model [1].

In many systems it is critical that they function according to their specification. Model checking is a way to guarantee that a system satisfies the given conditions or to find the cases where it fails.

Exhaustive model checking can give exact answers and guarantees about the modeled systems by exploring the entire state-space of the model. Such an approach to model checking suffers from state-space explosion due to the number of possible states growing rapidly as the model increases in complexity. Statistical model checking mitigates the problem of state-space explosion by simulating the system many times and through hypothesis testing can give similar assurances about the model, albeit without complete certainty of those assurances [8].

The Modest Toolset [4] is a toolset that can be used to perform model checking. This toolset supports the model input formats of JSON Automata Network Interface (JANI) and the Modest Language. The Modest language is a modeling language designed to be used to create models to check with the Modest Toolset. JANI [3] is a interchange format for models which is an accepted input language for many other model checking tools.

One of the features in the Modest Toolset is converting the models to a code representation, this code representation provides an interface which can be used to traverse the state space of the model. This enables the writing of algorithms on those models. The code generation is currently used in a teaching setting, where the generated code facilitates interacting with models such that the students can write their own model checking algorithms. The Modest Toolset currently generates code that is not as fast as it could be. The performance of the generated model affects the performance of algorithms using that model, which limits the size of models that can be effectively checked. Code generation is currently limited to Python code; adding a facility to generate models in more languages could have performance benefits, as more efficient programming languages than Python are available [9], as well as accessibility benefits, as many students in the current teaching setting using the Modest Toolset are not experienced in Python.

Contribution

In this paper I investigate how JANI models can be converted to an efficient code representation. I approach this by presenting a method of generating more efficient Python code from JANI models, showing a way of generalizing this code-generation for generation of different languages, and showing the performance differences of generated models in Python and C.

1.1 Related work

The Modest Toolset is not the only tool to convert models to code, the Spin model checker has an option to output C code [7]. This C code runs a model checker on the input model, it contains both the code to check the model and a representation of the model itself. The implementation of the code generation in both Spin and the Modest Toolset is not a suitable base for multi-language code generation, as their architecture hardcodes many assumptions for their single target language instead of in a more generic way for more languages.

As one of the goals is to generate models in multiple programming languages, principles from source-to-source compilers can be used in this project. In the context of converting models there is the HyST tool, which converts models other modeling languages [2]. HyST converts the source model to an intermediate representation and from that into another modeling language, the intermediate representation is similar to the internal representation of a model in the Modest Toolset, but takes no further step towards code generation.

An example where code does compile to many other languages is the Haxe cross-compiler. While it seems like the structure of the Haxe compiler could be used to generate code in multiple languages from models, the Haxe compilers' architecture cannot be effectively used in our case. This is due to the Haxe languages features being specifically designed to be compiled to many other languages and the compiler architecture being based around those language features [11]. The JANI model that should be converted to code is not specifically designed to be compiled to many programming languages and will thus not be able to work in the same way as Haxe does.

With regard to the performance of the generated code there have been studies that show that lower level programming languages produce faster programs than higher level languages [9] That research is on very general cases, thus researching this for the very specific case of models is not yet explored fully.

1.2 Methodology

Python code generation is already part of the Modest Toolset, thus the first target was to make performance improvements to that implementation. The language I added code generation for was C, since in the teaching environment where the Modest Toolset is used, students are often more familiar with C than with Python; additionally, C is a low level language compared to Python which is a high level language.

In order to improve the performance of the generated Python code I started by analysing which aspects of the generated code have a large performance impact. Using the knowledge gained from this process I implemented an improved solution. While improving the Python code I was generalizing the code generation to be applicable to other languages as well, after which I added C code support for this now more general code generator. Generalizing the code to work for other languages allowed me to better analyze the performance difference between the generated Python an C code, as the more general solution to generating the same model in different languages would create less functional differences in the resulting generated code, preventing performance differences produced by higher quality implementations of the converter for some languages than for others.

To prevent the results of the benchmark experiments being dependent on the model used, the results are based on multiple models from the Quantitative Verification Benchmark Set [6]. This benchmark set contains models, and for each of those models one or more instances of that model in the JANI format with different parameters. I performed the experiments on all MPD and PTA model instances from the benchmark set which could be converted to functional code, as some model instances could not be compiled to code with the current implementation and some model instances would not run due to the limits of the Python interpreter and the implemented C generation. From this benchmark set 66 model instances could be converted to Python code, 80 to C code, of which 60 could be converted to both C and Python code. These model instances instances ranged from smaller to larger models, the smallest model instance having 7 states and the largest having 361 trillion states. The benchmarks were performed on a laptop with an Intel Core i7-7700HQ CPU and 16 GB of RAM running running 64-bit Ubuntu Linux 20.04. The C code was compiled using the GCC C compiler and the Python Code was run using the Python 3.7.3 interpreter.

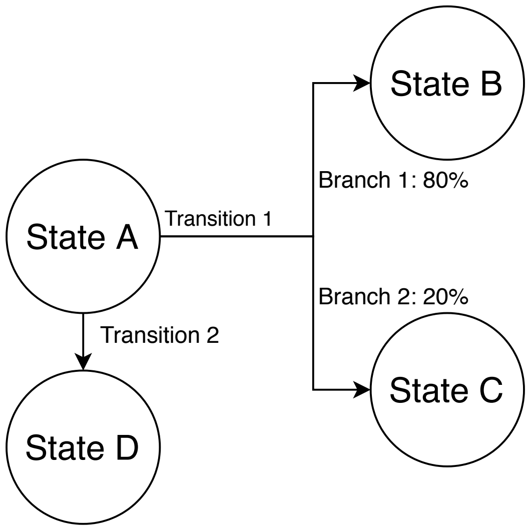

Figure 1. A small example showing the flow from state to state

To evaluate the results of research question 1 I conducted performance analysis on exhaustive and statistical model checking algorithms using the existing and the new implementation. I did a similar comparison between models generated in Python and C, and compared the efficiency of Python, C and the model checker *mcsta* in the Modest Toolset.

At the foundation of exhaustive model checking is an exploration of the state-space of the model. This exploration is the only interaction with the code representation of the model and the part of model checking which is slow, thus to evaluate the performance of the algorithm it is only necessary to benchmark the time taken to do a Breadth First Search (BFS) through the state-space of the model. As statistical model checking relies on simulations, the evaluation of the performance of the generated code was done by running Monte Carlo simulations on the models.

2. BACKGROUND

The models supported by the Modest Toolset are in essence automata, these automata have non-deterministic *transitions* and probabilistic *branches* which together form the edges between states, this is illustrated in the example in Figure 1. The Modest Toolset also supports models which consist of multiple smaller automata, these automata are synchronised together to behave like a single automata based on shared transitions between these smaller automata.

The generated code contains 3 important elements for the exploration of the state-space of the model:

- The automata classes, these represent the smaller automata, they are an interface which can be used to navigate the state-space of that specific automata by providing functions which give the *transitions*, *branches*, and next states.
- The *State* class, this class is used to store the states of all the automata in the model in on data object, it has a hash function, as some algorithms, such as BFS, require hashing the states of the model.

Algorithm 1: Monte Carlo Simulation on a model network

```
input: A model instance network, Number of trials
    n, Depth to explore d
for n times do
    state = network.get_initial_state()
    transitions = network.get_transitions(state)
    depth = 0
    while transitions is not empty AND depth < d do
        increment depth
        transition = a random choice from transitions
        branches = network.get_branches(state, transition)
        branch = a random choice from branches
            weighted by each branches probability
        state = network.jump(state, transition, branch)
        transitions = network.get_transitions(state)
    end
end
```

- The *Network* class, this class is the representation of the entire model. It synchronises the smaller automata together and is interacted with in a similar way as those smaller automata, by providing functions which give the *transitions*, *branches*, and next states.

3. IMPLEMENTED ALGORITHMS

The generated Python code does not do anything on its own, to be able to test its performance algorithms must be used which interact with the generated code. In this research project BFS is used to approximate an exhaustive model checking algorithm and Monte Carlo simulations are used to approximate statistical model checking.

Executing these algorithms on models requires finding the next state from any given state. States have a set of possible transitions which can be taken from those states, these are fetched using the *get_transitions* function, which gets a list of possible transitions from a given state. These transitions may have multiple branches to next states, each with a probability of taking that branch, the branches are fetched using the *get_branches* function. The function *jump* gets the new state given an old state and a transition and branch from that state.

Due to some of the models in the benchmark set taking a very long time to evaluate I defined a maximum amount of time that the benchmarks may run. I set this to 10 seconds per model instance and recorded for each of the aforementioned functions the number of seconds spent executing that function per explored state.

3.1 Breadth First Search

As can be seen in the pseudocode for BFS in Algorithm 2 BFS requires keeping track of the states which have already been visited. To do this the states are placed in a visited set. The *set.contains* operation and adding to a set use the *__hash__* function of the state of the model; this is needed because explored states should not be explored again. *get_transitions* and *get_branches* are used to find all outgoing edges from the current state and the *jump* function is used to get the new states from those outgoing edges.

Algorithm 2: Breadth First Search on a model network

```
input: A model instance network
let Q be a queue
let discovered be a set of states
initial_state = network.get_initial_state()
Q.enqueue(initial_state)
add initial_state to discovered
while Q is not empty do
    state = Q.pop()
    transitions = network.get_transitions(state)
    foreach transition in transitions do
        branches = network.get_branches(state, transition)
        foreach branch in branches do
            new_state = network.jump(state, transition, branch)
            if not discovered.contains(new_state) then
                Q.enqueue(new_state)
                add new_state to discovered
            end
        end
    end
end
```

3.2 Monte Carlo Simulations

It is not necessary to use the *__hash__* function when running simulations, since having visited a state already is not relevant in a simulation as is shown in the pseudocode for Monte Carlo Simulation in Algorithm 1. Like in the BFS implementation *get_transitions*, *get_branches* and *jump* are used to navigate the state-space of the model. The probability attribute of the branches are used as the probability of exploring that branch and transitions are explored with equal probability.

There may be cases where the simulation does not halt. To prevent the benchmark from looping forever there is a maximum number of *jump*s that will be taken; during the experiments this was set to 1000 *jump*s and 1000 simulations were run per model.

4. EFFICIENT PYTHON CODE

4.1 profiling results on benchmarks

As can be seen in Table 1, and Figure 2 and 3 in both the BFS benchmark and the Monte Carlo simulations the *get_transitions* function showed the largest time usage. This is due to the synchronisation of multiple automata transitions being required to get the possible transitions of the entire model, and synchronization being a relatively complex operation. The speed of all of these functions in the generated code is dependent on the model from which the code is generated. More complex models with more complex synchronisation have slower *get_transitions* functions and models with complex states have slower *__hash__* functions. More complex synchronisation has a very large effect on the speed of the model checker, this can be seen in the figures as the percentage of time used by *get_transitions* increases as the total speed decreases.

4.1.1 BFS

The most surprising value from the result of profiling the BFS implementation is the amount of time spent in the *__hash__* function. This is due to the large number of calls that are made to this function. The current implementation in the Modest Toolset produces hashing functions

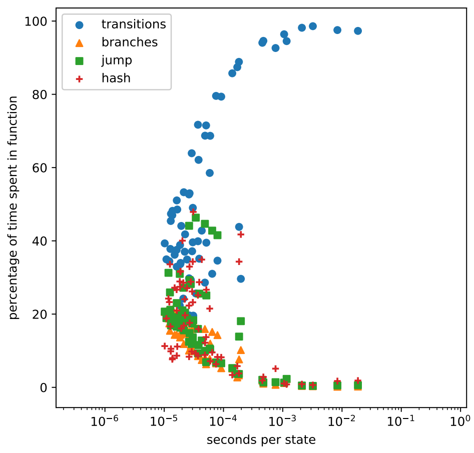

Figure 2. Performance analysis of the existing generated Python code by benchmarking seconds per state explored using BFS for multiple model instances

	percentage of time spent	
Function	BFS	Monte Carlo
get_transitions	51.5%	77.9%
jump	16.8%	7.3%
get_branches	11.6%	7.7%
__hash__	17.5%	–

Table 1. Average Time spent in function

which are much slower than the tuple hash which is in Python by default. Thus a significant performance increase can be achieved by using that instead.

4.1.2 Monte Carlo Simulations

Compared to the BFS profiling, Monte Carlo Simulations spend much more of the time in *get_transitions*. This is due to *jump* and *get_branches* only being called once for each call of *get_transitions* as only one of the transitions is picked to be explored, while BFS explores all transitions and branches and will thus call *get_branches* and *jump* multiple times per call of *get_transitions*. Due to *get_transitions* running such a large percentage of the time it is the only function where major performance improvements can be gained for statistical model checking.

4.2 implemented improvements

4.2.1 new hashing function

The functions generated by the old code generator to hash the model state worked by iteratively applying an operation to combine the provisional result with a hash of a variable of the state, as seen in the example in Listing 1. This is not an incorrect way to hash, but the hash function is called very often and Python is a relatively slow language.

```
def __hash__(self):
  result = 75619
  result = (((101 * result) & 0xFFFFFFFF) +
    hash(self.counter)) & 0xFFFFFFFF
  result = (((101 * result) & 0xFFFFFFFF) +
    hash(self.pc1)) & 0xFFFFFFFF
  result = (((101 * result) & 0xFFFFFFFF) +
    hash(self.coin1)) & 0xFFFFFFFF
  result = (((101 * result) & 0xFFFFFFFF) +
    hash(self.pc2)) & 0xFFFFFFFF
  result = (((101 * result) & 0xFFFFFFFF) +
    hash(self.coin2)) & 0xFFFFFFFF
  return result
```

Listing 1. Hashing function produced by the existing implementation

```
def __hash__(self):
  return hash((self.counter, self.pc1, self.
    coin1, self.pc2, self.coin2))
```

Listing 2. Hashing function produced by the new implementation

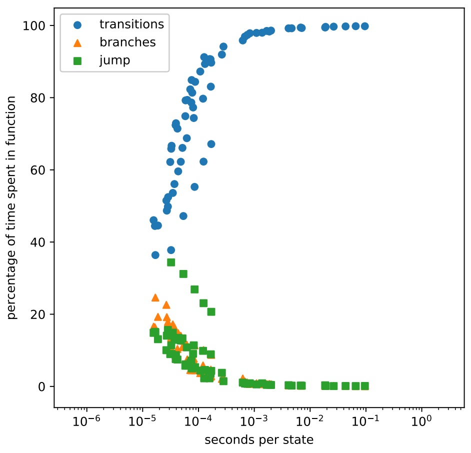

Figure 3. Performance analysis of old generated Python code by benchmarking seconds per state explored using Monte Carlo Simulations for multiple model instances

The hashing function produced by the new implementation utilises the standard Python tuple hash, as seen in the example in Listing 2, which works using a similar iterative operation as the hash produced by the old code generator.

4.2.2 memoization

Memoization is an optimisation strategy which is used to avoid recalculating the same values when not necessary. This is achieved by storing the return values of the called function and checking whether or not a value is already stored for the given inputs. This only works for idempotent functions as it cannot account for a different output given the same input, fortunately the *get_transitions* function is idempotent. I implemented this in the Python code generator by using dictionaries, as seen in the example in Listing 3. It is important to note that utilising this strategy increases the memory used when running the Monte Carlo Simulations; one of the benefits of statistical model checking is the reduced memory usage which is thus undermined by using memoization. This did not pose a problem in the tests ran, as these tests were limited in run time, which did not allow the memory usage to become a bottleneck; however this would pose a problem if a more thorough exploration of the model was performed. To prevent this becoming a problem in a proper exploration of the model a pache peplacement policy could be used to limit the size of the cache.

```
def get_transitions(self, state: State) ->
    List[Transition]:
```

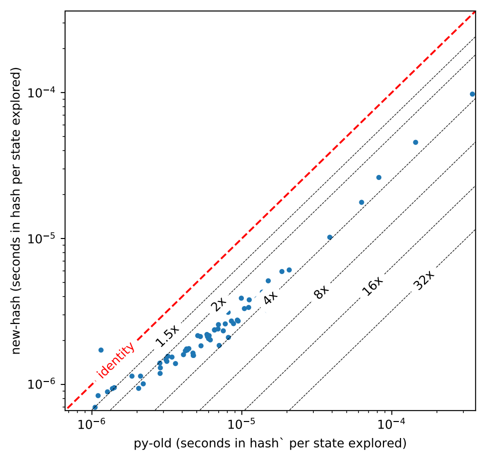

Figure 4. Performance comparison between the hash functions produced by `new-hash` and `py-old` using BFS as a benchmark.

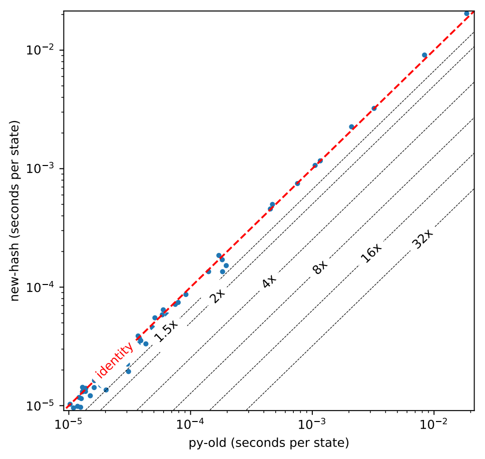

Figure 5. Performance comparison between `py-old` and `new-hash` using BFS as a benchmark.

```
  if state in self.transitions_table:
    return self.transitions_table[state]

  #operations generating the result in the
    transitions variable

  self.transitions_table[state] = transitions
  return transitions
```

Listing 3. An example of generated code using memoization

4.3 Results

I will consider the following implementations:

- `py-old`: The old implementation for Python.
- `new-hash`: The old implementation for Python, but with the new hashing function.
- `memoized`: The old implementation for Python, but using memoization.
- `both`: An implementation for Python using both memoization and the new hashing function.

4.3.1 *new hashing function*

As seen in Figure 4 the new generated hashing functions are much faster than the old hashing functions. This leads to a noticeable increase in performance in BFS, but only for less complex models, as seen in Figure 5. The new hash function has very little influence on the results of the more complex models, as these spend the vast majority of the time in *get_transitions*, as seen in Figure 2.

4.3.2 *memoization*

As seen in Figure 6 the *get_transitions* functions generated by `memoized` are substantially faster than the old *get_transitions* functions. This leads to a significant increase in performance in Monte Carlo Simulations as seen in Figure 7. This method does not offer any benefits for BFS as in BFS every state is only visited once and thus the *get_transitions* function is never called with the same parameters twice.

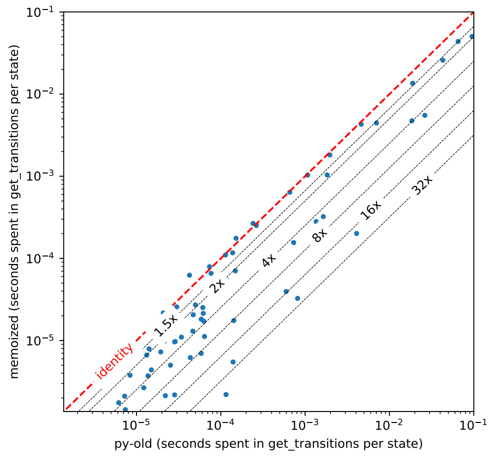

Figure 6. Performance comparison between the *get_transitions* functions produced by `memoized` and `py-old` using Monte Carlo Simulations as a benchmark.

4.3.3 *new hash function and memoization programming*

Due to memoization programming making use of dictionaries in *get_transitions*, and dictionaries using the state hash function, both optimisations can be combined to speed up the Monte Carlo Simulations further. Figure 8 shows that using the new hashing functions makes it noticeably faster than only using memoization. This results in the total performance increase to be substantial compared to the old generated code as shown in Figure 9.

5. GENERALISING CODE GENERATION

In order to make it easier to add more languages I planned on expressing the model in abstract language structures. Programming languages are often built up of similar statements and expressions. Expressing the generation of models using code generation units which only generate one particular language structure can help making adding more languages easier as only the implementation of these units

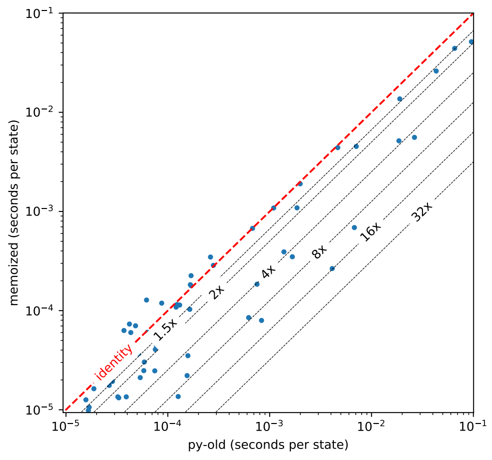

Figure 7. Performance comparison between `py-old` and `memoized` using Monte Carlo Simulations as a benchmark.

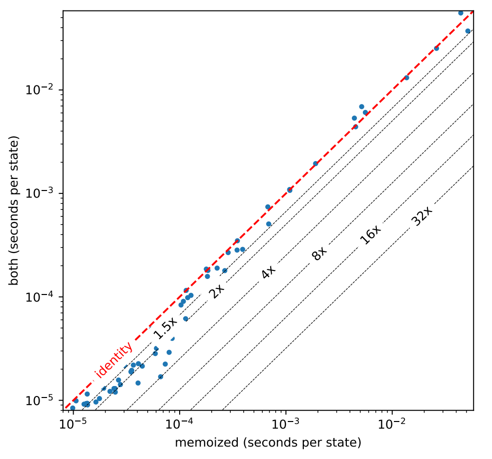

Figure 8. Performance comparison between `memoized` and `both` using Monte Carlo Simulations as a benchmark.

must be changed to add a new language, and not the way these units are structured in the code generation. This approach is similar to the use of an intermediate language. If some of these language structures are available in one language, but not in another, then more effort is needed to produce a working code generator, but not more than would have been needed if this strategy was not deployed.

5.1 Templating engines

To achieve these code generation units I initially planned on using the StringTemplate library [10], this library has great tools for creating subtemplates. These subtemplates could then be used to express a single language structure like the code generation units I described above, which could use more subtemplates of more detailed language structures. Unfortunately the StringTemplate library is not maintained well enough for C# to use currently. The T4 templating engine was considered, but it did not have the same advantages StringTemplate offered and thus a stringbuilder was used, like it was in the original implementation.

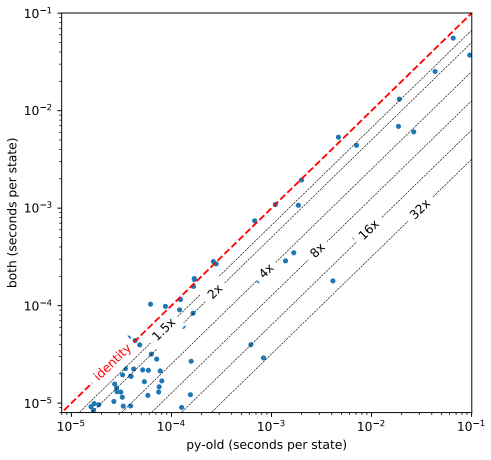

Figure 9. Performance comparison between `py-old` and `both` using Monte Carlo Simulations as a benchmark.

5.2 Implemented generalisation

I generalised only a small number of these structures: switch cases, for loops and scopes, as these features were used often and most easily generalised. While much of the code generation was not generalised, the parts which were made implementing these parts for C almost trivial.

```
class SwitchCase
{
    private IndentedTextStream s;
    private String switchVar;
    private bool needEl = false;
    public SwitchCase(IndentedTextStream s, String switchVar){
      //the textstream to write the generated code to
      this.s = s;
      // the variable to be switched on
      this.switchVar = switchVar;
    }
    public Disposer Open(){
      return new Disposer();
    }

    public Disposer AddCase(String option){
      if (needElse){
        s.Write("el");
      } else {
        needElse = true;
      }
      s.WriteLine("if " + switchVar + " == " + option + ":");
      return s.Indent();
    }
}
```

Listing 4. Switch case generator for Python

```
class SwitchCase{
    private IndentedTextStream s;
    private String switchVar;
    public SwitchCase(IndentedTextStream s, String switchVar){
        //the textstream to write the generated code to
        this.s = s;
        // the variable to be switched on
        this.switchVar = switchVar;
    }
    public Disposer Open(){
        s.WriteLine("switch(" + switchVar + "
    )");
        return OpenScope(s);
    }
    public Disposer AddCase(String option){
        s.WriteLine("case " + option + ":");
        Disposer d = s.Indent();
        return new Disposer(() =>
        {
            d.Dispose();
            s.WriteLine("break;");
        });
    }
}
```

Listing 5. Switch case generator for C

As can be seen in Listing 4 and 5, C supports switch cases, while Python does not, however the same behaviour as a switch case can be attained by using repeated if-elif statements. The implementations of these classes are different, but they are interacted with in the same way: A switch-case is opened, multiple cases are added, after which it is closed[1]. and can thus be used in exactly the same way during generation.

5.3 C implementation

The largest difference between C and Python is Python having much much more abstract features, whereas C requires writing these abstract features yourself. However as Python and C are both imperative programming languages they share many of the same control structures. Thus I made the C generator produce the same control structures in as in Python. These control structures are therefore also the best candidates to generalise for many languages.

Python and C do typing in a fundamentally different way, therefore this caused the most difficulty in generalising the Python generation to C. In Python, new data types are defined as classes, which encapsulate data and behaviour, and in C data types are defined as structs, which only encapsulate data. Thus the behaviour that was originally in the Python classes needs to be moved to functions outside of the structs with the same behaviour. Additionally Python does its typechecking at run-time through duck typing, whereas C does its typechecking at compile-time and requires strict declarations of types. This leads to the problem of the Python generation not needing information on the types used, while C does require it, as a consequence this was the area of code generation that was most difficult to change in a generic way from Python to C; in this case I added the types by interpreting myself which types had to be used, for example needing to specify that a variable is in fact an integer, instead of just assigning an integer value to an untyped variable and deferring typechecks to the Python interpreter at runtime. Python also supports datastructures which are not available in C by default, such as lists, sets and dictionaries, which thus needed to be added to the generated C code. As memoization requires dictionaries, this was not implemented in C. These problems would not be as apparent if the initial implementation would have generated C and would have been changed to also generate Python; because it is less difficult to convert the more verbose C code to the more concise Python code, as the C code needs to contain more information than the Python code and it is easier to remove information than to add it.

[1]C# has a *using* statement, which calls a method, then opens a block and when the block is closed calls the *Dispose()* method on the returned disposer, which is used to close both the switch-case and each case

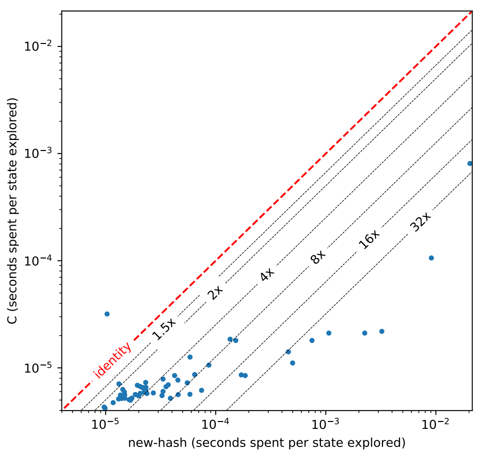

Figure 10. newhash vs c using BFS

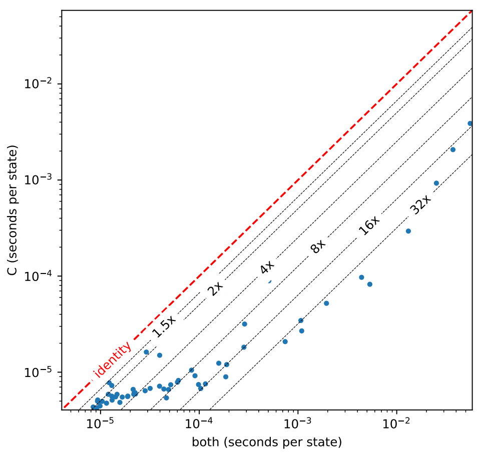

Figure 11. both vs C using Monte Carlo

6. EFFICIENCY ACROSS LANGUAGES

I applied the BFS and Monte Carlo benchmarks to the generated C code in the same way as on the generated Python code, also producing results in seconds spent per explored state. The benchmarks showed that the generated C code was much faster than `new-hash`, the fastest Python implementation for BFS, when using BFS as a benchmark as seen in Figure 10.

Figure 11 shows that the generated C code is also much faster than `both` when using Monte Carlo simulations as a benchmark.

The Python and C implementations can also be compared to, the Modest Toolsets *mcsta* tool, which will compile the model to a representation in .NET bytecode [5].

Table 2 shows the speed of BFS on several models in different languages and tells us that in these few cases Python is much slower than both C and the .Net bytecode. Whether or not the C code is faster than the .Net bytecode depends

model instance	`new-hash`	C	.NET bytecode
echoring	17.1k	79.6k	145k
rabin.5	73.7k	180k	170k
elevators.b-11-9	30.1k	127k	96.9k

Table 2. States explored per second using BFS for various model instances in different languages.

on the model.

7. CONCLUSION

In this research project I set out to improve the performance of the Python code generated by the Modest Toolset, to simplify adding support for generating more programming languages and to compare the performance of model checking in different programming languages. The existing generation of Python code was modestly improved for exhaustive model checking by improving the generated hash function. Its performance for statistical model checking was significantly increased by using memoization and the new hash function together. However this was achieved by caching all values, which would lead to problems in memory usage, thus this should be improved by using a cache replacement policy, which policy is best for the specific case of statistical model checking is not part of this research, but could be explored further in future research.

In order to make adding support for more programming languages less difficult I changed the code generation to make use of more general language structures. This is currently limited to a few control structures, but could be expanded in the future, but to a limited extent as many details of programming languages differ enough to make generalising all concepts difficult.

I added support for C code generation, which was much faster than both the existing and improved Python implementations C is much much faster and scales better.

As a possible extension Haskell code generation could be implemented as well, as it is a language from a different paradigm it could provide contrast to C and Python. As a further work the benchmarks could be made more accurate by not only using the exploration stage of model checking, but by using a full model checker and memoization could be applied to C to further increase its performance for statistical model checking.

8. REFERENCES

[1] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[2] S. Bak, S. Bogomolov, and T. T. Johnson. HYST: A source transformation and translation tool for hybrid automaton models. In A. Girard and S. Sankaranarayanan, editors, *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC'15, Seattle, WA, USA, April 14-16, 2015*, pages 128–133. ACM, 2015.

[3] C. E. Budde, C. Dehnert, E. M. Hahn, A. Hartmanns, S. Junges, and A. Turrini. JANI: Quantitative model and tool interaction. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 151–168, 2017.

[4] A. Hartmanns and H. Hermanns. The modest toolset: An integrated environment for quantitative modelling and verification. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 593–598. Springer, 2014.

[5] A. Hartmanns and H. Hermanns. *A Modest Markov Automata Tutorial*, pages 250–276. Springer International Publishing, Cham, 2019.

[6] A. Hartmanns, M. Klauck, D. Parker, T. Quatmann, and E. Ruijters. The quantitative verification benchmark set. In T. Vojnar and L. Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, volume 11427 of *Lecture Notes in Computer Science*, pages 344–350. Springer, 2019.

[7] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[8] A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer, 2010.

[9] S. Nanz and C. A. Furia. A comparative study of programming languages in rosetta code. In A. Bertolino, G. Canfora, and S. G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 778–788. IEEE Computer Society, 2015.

[10] T. Parr. StringTemplate. `https://www.stringtemplate.org/`, June 2020.

[11] D. Štrekelj, H. Leventić, and I. Galić. Performance Overhead of Haxe Programming Language for Cross-Platform Game Development. *International journal of electrical and computer engineering systems*, 6(1):9–13, May 2015.