Sharm: A unified toolbox for Harmony, written in Swift

William Ma
Cornell University
wm274@cornell.edu

Abstract

We introduce Sharm: a toolbox for Harmony bytecode. Sharm is an interpreter, model checker, and bytecode-to-source compiler, and bytecode-to-source model checker. We implement Sharm in Swift and discuss some of the key design decisions we made ([1]). In our evaluation, we show that Sharm is performance competitive with Charm albeit slower.

Our code is publicly available at `https://github.com/whoiswillma/swift-harmony/`.

1 Introduction

Software model checking involves systematically analyzing traces of a system in order to prove properties about it ([9]). Typically these systems are specified as programs. There are two main categories of model checkers.

Specification-level model checkers such as Spin, TLC, and Charm ([8], [12], [10]). These model checkers accept input in a specification language such as PROMELA, TLA$^+$, and Harmony respectively. With tools such as TLC users specify safety properties as temporal logic formulae, and in Charm the Harmony program includes `assert` and `invariant` statements. These kinds of model checkers can be comprehensive with their model checking since the search space is typically constrained by the design of the language. However, translating the model-checked specification into an implementation can be a challenge.

Code-level model checkers such as VeriSoft and JavaPathFinder ([7], [11]). Code-level model checkers accept input in an implementation language such as C, C++ or Java. With code-level model checkers, the implementation is immediate. However, because the state space of an entire machine is so large, these model checkers usually do not search the entire state space, use heuristics, or must redundantly check some states.

Last Modified: Thursday 23rd December, 2021 17:00 UTC

	Interpreter	Compiler
Executor	CPython, CRuby	Clang, javac
Model Checker	TLC, Charm, VeriSoft, JavaPathFinder	Spin

Figure 1. An attempted categorization of some existing code tools.

Additionally, we can categorize tools along two relevant dimensions. The first is whether the tool executes or model checks its input, and the second is whether input programs are interpreted or compiled. For example, a compiling model checker accepts as input a specification in a high-level language and produces as output a program which, when executed, model checks the original input program. With little justification, in figure 1 we attempt to categorize some existing tools (including the aforementioned ones).

Sharm is able to fill in all of these gaps for Harmony. In particular, Sharm offers an interpreting executor, an interpreting model checker, a compiling executor, and a compiling model checker. This allows a user to extract an implementation after it has been model checked and shown correct.

For a user looking to model check their program, Charm is certainly the better tool. It is integrated with the Harmony frontend, supports all Harmony features, outputs better diagnostics, and has better performance than Sharm. However, from a development perspective, we hope Sharm is eventually able to offer performance competitive with Charm while being more extensible. Of course, Charm currently does not have the ability to execute Harmony code.

In summary, our contributions are as follows:

- A tool which compiles Harmony bytecode to Swift code, both for execution and model checking (§3).
- A framework for developing a interpreting executor, interpreting model checker, compiling executor, and compiling model checker together in Swift (§4).
- An evaluation of Charm versus Sharm (§5) on model

checking, and an evaluation of the interpreting executor versus compiling executor.

1.1 Why Swift

In Sharm, the compiling executor and compiling model checker output programs in Swift. Compiling to Swift has some benefits, none of which are particularly profound. First, since Swift uses LLVM for its backend, the compiled Harmony program can also leverage some of the benefits of those optimizations, especially optimizations that require knowledge about consecutive lines of bytecode. Second, Swift is a good match for Harmony in particular. Collections such as dictionaries and sets are value types in Swift which match their semantics in Harmony. These collections are also well-optimized through the use of copy-on-write, an optimization which was essentially performed manually, though with more precision, in Charm. Third, Swift has language features (admittedly not unique to it) that make it easy to reuse code between the various components of Sharm while still having good overall performance.

Writing Sharm in Swift itself has some additional benefits. Swift has enums with associated values, which are highly useful for expressing ops and their arguments in a type-safe manner ([2]). Swift also lets the programmer choose between dynamic dispatch and static dispatch, and includes `inout` parameters to express pass-by-reference for value types, similar to passing a pointer to a function in C ([5]).

Another feature of Swift that became unexpectedly useful is its support for multi-line string literals with interpolated parameters ([3]). Multi-line string literals are of course helpful for storing chunks of code as strings. Over the course of development, it was handy to add an interpolated string literal within a string. In most other languages this would be done with a litany of escape characters or with string concatenation. In Swift, by prefixing and postfixing a string literal with number symbols, the normal string interpolation syntax is interpreted literally, while still allowing for string interpolation based on the number of number symbols prefixed or postfixed ([4]).

There are downsides to using Swift as well. Although the core language is platform independent, it is less mature on Windows than on macOS and Linux, and certainly less mature than C on all platforms. Moreover, one of the largest and most useful libraries `Foundation` is Apple platform specific. Of course, the trade off with using a high-level language is to give up control in exchange for ease-of-use, which places a cap on the performance that can be achieved compared to C.

2 Related Work

The Spin model checker is a model checker for PROMELA, and can verify temporal logic formulae ([8]). Their design constructs an automaton for the program, a Büchi automaton for the correctness claims, and analyzes the product of the two automata. The output of Spin is a C program. The model checker is executed by compiling and running this C program.

Spin perform state compression to reduce the memory consumption of the model checker. The entire state includes the global state, as well as many small local states. When taken as a product, these are one source of high memory consumption in prior model checkers. Spin separates global state storage and local state storage and holds references to unique local states within the global state. The Spin paper shows that this provides significant memory reduction with a slight performance overhead.

TLC is an enumerative model checker for verifying a subset of specifications written in TLA$^+$([12]). In the version presented in the paper, TLC can check safety but not liveness properties. The designers were willing to compromise on speed, but not on the size of model able to be checked. To this end, TLC explicitly manages the disk, using main memory as a mere cache. TLC also supports multi-threaded model checking. For performance reasons, TLC additionally allows any TLA$^+$module to be overridden by a Java class.

TLC has some important limitations. Since TLA$^+$is built on ZF set theory, it's clear that TLC cannot handle every TLA$^+$specification. Moreover, TLC requires that specifications be in TLA "canonical form", i.e. with a single initial condition and next-state action. TLC also requires specifications to give type invariants for variables in order to constrain the domain of existentials.

3 High-level Overview

3.1 Executors

The interpreting executor and compiling executor run Algorithm 1.

What remains is to specify when a context is terminated, what are the runnable contexts, and what a switch point is.

A context terminates when it returns from its top-level function. For `__init__` this happens when it has ran all the top-level code, and for contexts this happens when the contexts returns from the function which it was invoked on.

A context is runnable if it is atomic, or there are no atomic contexts. Equivalently, the set of runnable contexts is all contexts, unless there is an atomic context, in which case that context is the only runnable context.

Algorithm 1 Harmony program execution

```
Initialize contextBag with the initial context
while not all contexts are terminated do
    runnable ← getRunnable(contextBag)
    context ← choose(runnable)
    do
        context ← code[context.pc].run(context)
    while context is not at a switch point
    Update context in contextBag
end while
```

A switch point is the location of a context switch. This could be after every instruction, though its not done so for efficiency reasons. If two contexts modify only their local state for some substring of the trace, then any interleaving of their steps will produce the same state. Therefore, switch points are inserted at key points which could influence the final state of the program:

- The context has terminated
- The context will go from non-atomic to atomic
- The context went from atomic to non-atomic
- The context has read or written global state

An interesting design choice is to use a bag to hold contexts. A bag a.k.a. multiset is a set with multiplicity. In most cases, every element in contextBag is distinct, so it acts just as a set. There are a few situations where it can save memory. For example, if several contexts execute the same method with the same arguments, using a contextBag can save memory by having multiple identical contexts be stored only once.

3.2 Model Checkers

The interpreting model checker and compiling model checker run algorithm 2 similar to the Enumerative Reachability Algorithm given in figure 2 of [9].

In this case, some affordance must be offered to the `choose` operation. Since the `choose` operation can potentially create more than one state, it must kick back to the stateful model checker.

Switch points are inserted identically to as they are in the interpreter. In the context of model checking, *not* inserting switch points after every instruction is a form of symmetry reduction ([9]): instead of checking every possible interleaving, the model checker iterates over a representative interleaving, which reduces potentially several equivalent interleavings to checking a single interleaving.

In the same vein, storing contexts in a contextBag is also a form of symmetry reduction. This is an innovation borrowed from Charm. Compared to its use in the executor

Algorithm 2 Stateful Model Checker

```
visited ← $\emptyset$
boundary ← { (initial state, initial context) }
while $\exists$ (st, ctx) $\in$ boundary do
    if (st, ctx) $\in$ visited then
        continue
    end if
    visited ← visited $\cup$ { (st, ctx) }
    assert ctx $\in$ st.contextBag

    do
        st, ctx ← code[context.pc].run(st, ctx)
    while context is not (after a switch point or before choose)

    if context is before choose then
        for value $\in$ possible choices do
            ctx' ← choose(ctx, value)
            boundary ← boundary $\cup$ { (st, ctx') }
        end for
    end if

    if context is after a switch point then
        Update st.contextBag with ctx
        for nextCtx $\in$ runnable(st) do
            boundary ← boundary $\cup$ { (st, nextCtx) }
        end for
    end if
end while
```

(§3.1), the contextBag is much more important in the model checker. Consider a Harmony program which spawns n contexts which immediately wait on an n-context barrier. Giving each context an identity such as its index in an array would require checking on the order of $n!$ number of states, one per permutation. Storing contexts in a bag reduces this to a polynomial number of states, which turns out to be quadratic in n.

4 Implementation Details

The interpreting executor and interpreting model checker were straightforward to implement. In this section we cover more important design decisions we made with regards to Sharm.

4.1 Ops

The interpreting executor, interpreting model checker, compiling executor, and compiling model checker all depend on implementations of core Harmony ops. Every Harmony op implicitly takes as input the current context, and some additional arguments specified in the Harmony bytecode. Broadly speaking, there are three ways to implement these operators:

Functional. Static methods which take immutable arguments and return the updated context. In Swift, the function signature for the implementation of `Push` would be

```
static func push(
    context: Context,
    value: Value
) -> Context
```

Since `structs` are value types and `Context` is a `struct`, the Swift compiler may have to copy the context at runtime for ops that mutate the context.

Object-oriented. Since every op takes a context as input, an object-oriented approach would be to implement the operators on the Context structure.

```
struct Context {
    ...
    mutating func push(value: Value)
    ...
}
```

Swift requires that the programmer annotate mutating methods on `structs` as a way of precisely specifying the `struct`'s interface, though the Swift compiler is capable of inferring whether a function is mutating from its implementation.

The object-oriented approach has some limitation. If we implement `Context` as a struct, then `push` would be statically dispatched. However, it would be more difficult to customize the behavior of a `Context` since `structs` cannot be subclassed. If we implement `Context` as a class, then we can customize its behavior in subclasses, but we could incur dynamic dispatch overhead.

Imperative. Pass a reference to the context as an `inout` argument.

```
static func push(
    context: inout Context,
    value: Value
)
```

We chose to implement ops imperatively. An imperative implementation makes minimal assumptions about what context state is needed before and after the call. If the caller must retain the old value of the context, it can duplicate it itself. Moreover, we can gain back the advantage of code-reuse and customizability from the object-oriented approach through *protocol-oriented programming* ([6]).

In Swift, protocols define interfaces that are implemented by classes or structs. Protocols can additionally define default implementations. In Sharm we define a protocol `OpVisitor` which implements the visitor pattern over operators. The protocol-oriented part is introduced with the `DefaultOpImplVisitor`, which provides default implementations of the methods in `OpVisitor` by requiring mutating access to a context.

In this way, the implementation of the interpreting executor and interpreting model checker is made simple. Each defines its own visitor which override methods in `DefaultOpImplVisitor` to detect when the context is at key instructions.

We would argue that protocol-oriented programming is not novel. As an example, abstract classes in Java serve a similar role. The strength of protocol-oriented programming comes from the fact that protocols serve as a means to give dynamic-dispatch-like behavior to structs, while still preserving their stack allocation (*citation needed*).

4.2 Compiling Executor

The Harmony-to-Swift Compiler (H2SCompiler) converts a sequence of Harmony bytecode ops into a Swift program. There are a few possible approaches to compiling Harmony into Swift code:

Strawman. Embed an array of Ops into a Swift file and run the executing interpreter in §3.1.

Compared to the interpreter, this approach only eliminates the time it takes to deserialize ops.

Single step. Generate a `step` function which takes as input a context and makes a static call to the op implementation based on the context's PC value, and embed the ops as control flow within the Swift code.

Compared to the strawman approach, this converts all calls to use static dispatch, but still incurs a function call overhead for every step performed.

Basic block. Identify spans of straight-line execution (basic blocks) within a Harmony program and embed those into the `step` function.

This approach has the most potential for speedup. Ops are statically compiled, which gives the Swift compiler the most opportunity to inline methods and perform inter-function optimization. As much code is possible is executed straight-line, which could reduce the overhead required to call the `step` function.

A basic block is composed of a start PC and a sequence of ops. To construct the basic blocks we must first identify PC values at which a basic block could start. We can conservatively overestimate the set of starting PC values, since starting PCs which cannot be reached are simply dead code in the final Swift file. In fact, we could simply mark all PC values as potential basic block starts. However, this would bloat the generated Swift file, so to reduce the amount of generated Swift code, we use the following conservative heuristics.

`AtomicInc`, `AtomicDec`, `Load`, `Store`. These ops define switch points in Harmony, so a context which resumes might begin executing at any of these ops.

`Frame`, `Apply`. These relate to function invocation. Control can of course flow to a `Frame` op, but we must care to add the PC *after* the `Apply` op, since control will return there after the function completes. Since `Apply` is also used for dictionary lookups, we will incidentally add more PC values than necessary.

`JumpCond(pc, cond)`, `Jump(pc)`. For `Jump` and `JumpCond`, we must add the destination PC as a potential start. For `JumpCond` we must also consider the case when the condition does not hold and control flow falls through to the next instruction.

After we have identified the starts of a basic block, enumerating its instructions is straight forward. We start at one of the PC values we identified above and append instructions to a list until we pass a `Jump` or `JumpCond`, `Apply`, or `Ret`, or until the next instruction is a `Load`, `Store`, `AtomicInc`, or `AtomicDec`.

4.2.1 Future Work

There are a few ways in which we would have liked to extend the Sharm compiler.

More control flow analysis. For simplicity, we chose not to follow the `Jump(pc)` op in our basic block analysis. A more fine-grained analysis could trace unconditional jumps to their destination in the `step` function and continue execution. This may be most profitable for the `__init__` context, which starts at `pc = 0` and typically performs many unconditional jumps before terminating at the end of the file.

Moreover, it may be possible to statically distinguish when `Apply` ops are for function calls by looking at previous instructions for whether a PC value is pushed onto the stack.

User input. `choice` can be used to model user input. A future goal is to allow interpreted Harmony programs to prompt the user for the result of choices.

4.3 Compiling Model Checker

The Harmony-to-Swift Model Checker (H2SModelChecker) takes as input a Harmony program and produces a Swift program that, once compiled and executed, model checks the original Harmony program. The H2SModelChecker performs the same basic-block optimization that was performed in the Harmony-to-Swift compiler within its step function.

One significant difference between the compiling model checker and the interpreting model checker is that the compiling model checker performs lazy atomic optimizations which are also performed in Charm, whereas the interpreting model checker does not. An atomic step can be performed lazily if none of its operations generate a switch point, such as a load or store to global state. In this case, an `AtomicInc`/`AtomicDec` need not generate a switch point when the context becomes atomic/non-atomic.

Ideally, we would have liked to implement lazy atomic in the interpreting model checker as well. It was done only in the compiling model checker for time reasons.

5 Evaluation

We wrote three Harmony programs for our benchmark suite: BinarySearch, Quicksort, and SelectionSort. Each program checks that the computed result is sensible using Harmony `assert` statements.

Program	Interpreted None	Interpreted Opt	Compiled None	Compiled Opt
BinarySearch-Exec	3.43	0.12	2.96	**0.10**
Quicksort-Exec	0.72	0.03	0.24	**0.02**
SelectionSort-Exec	15.82	0.38	2.85	**0.10**

Figure 2. Executor Benchmarks measuring execution time. Columns subtitled "None" mean the Swift compiler was not passed any optimization flags. Columns subtitled "Opt" means the Swift compiler was passed the `-O -wmo` flags.

Program	Compilation Time None	Compilation Time Opt
BinarySearch-Exec	4.08	13.17
Quicksort-Exec	2.63	9.23
SelectionSort-Exec	2.72	9.15

Figure 3. Swift compilation times for executable programs

Program	Interpreted None	Interpreted Opt	Compiled None	Compiled Opt
BinarySearch-Exec	3.43	**0.12**	7.04	13.27
Quicksort-Exec	0.72	**0.03**	2.87	9.25
SelectionSort-Exec	15.82	**0.38**	5.57	9.25

Figure 4. Executor Benchmarks measuring total time, which is the sum of compilation time and execution time. The compilation time for interpreters is zero. We see the optimized interpreter is significantly faster than the optimizing compiler because Swift compilation adds significant overhead.

We frame our evaluation around three questions that apply to both executors and model checkers.

- How effective is compiling as opposed to interpreting?

  We execute and model check our benchmark programs with both the interpreter and compiler.

- How effective are Swift compiler optimizations?

  For the interpreters, we compile the interpreting executor and interpreting model checker using the release configuration. For the compilers, we compile the generated Swift program by passing the `-O -wmo` flags to the Swift compiler. We are unsure which exact compiler optimizations the Swift compiler performs, but as far as we understand these flags are the way to have the compiler produce well-optimized executables.

- How much latency does Swift compilation add?

  Compiling Swift programs, especially with optimization, adds significant overhead. We compare the cost of running the interpreter versus the overhead of Swift compilation.

We perform our evaluations on a MacBook Pro 14" with an M1 Pro processor. Our benchmark script is available as `benchmark.sh` in our repository.

5.1 Executors

We benchmark the interpreting and compiling executors, with and without Swift optimization in figure 2. The compiled executable benchmarks only include the time to execute the generated executable, and do not include Swift compilation time which we evaluate next. We report the real time from the macOS `time` command in seconds and highlight the fastest time for each program.

A majority of the speedup comes from Swift compiler optimizations. For SelectionSort, starting from Interpreted/None, moving to Compiled/None results in a $6\times$ speedup, but moving instead to Interpreted/Opt results in a $42\times$ speedup.

In figure 3 we present just the Swift compilation times for the above programs, with and without optimization. In figure 4 we report the total time to compile and execute each program. If we measure the total time from when we invoke Sharm to when execution completes, then the optimized interpreting executor is the fastest since Swift compilation does add significant overhead.

5.2 Model Checkers

We benchmark Sharm's model checkers and Harmony on the above programs as well as a fixed version of the Dining philosophers. Our results for just execution time (without Swift compilation) are shown in figure 5. We reduce the size of inputs to make model checking tractable.

Once again, a majority of the speedup comes from optimized Swift code. For SelectionSort, starting from Interpreted/None, moving to Compiled/None results in a $1.5\times$ speedup, whereas moving instead to Interpreted/Opt results in a $19\times$ speedup.

The comparison with Harmony is unfair for Harmony. First, we report the time required to run the entirety of Harmony including the frontend, whereas for Sharm the time is measured from when we start the interpreting model checker or the compiled model checker, after the Harmony frontend has already generated bytecode and the Swift com-

Benchmark	Harmony	Interpreted		Compiled	
		None	Opt	None	Opt
BinarySearch-Mc	**0.14**	7.61	0.37	7.22	0.33
QuickSort-Mc	**0.98**	40.94	2.45	28.31	1.44
SelectionSort-Mc	**1.86**	67.25	3.56	43.47	2.01
DinersFixed	0.84	>300	70.91	0.2	**0.03**

Figure 5. Model Checker Benchmarks measuring execution time. Running the non-optimized interpreting model checker on DinersFixed timed out after taking longer than 300s.

Benchmark	Harmony	Interpreted		Compiled	
		None	Opt	None	Opt
BinarySearch-Mc	**0.14**	7.61	0.37	10.02	9.67
QuickSort-Mc	**0.98**	40.94	2.45	31.27	11.78
SelectionSort-Mc	**1.86**	67.25	3.56	46.61	13.13
DinersFixed	**0.84**	>300	70.91	4.73	14.36

Figure 6. Model Checker Benchmarks measuring total time. The compilation time for interpreting model checkers is zero.

Program	Compilation Time	
	None	Opt
BinarySearch-Mc	2.80	9.34
Quicksort-Mc	2.96	10.34
SelectionSort-Mc	3.14	11.12
DinersFixed	4.53	14.33

Figure 7. Swift compilation times for model checked programs

piler has potentially performed extensive optimization. Second Sharm is missing features from Charm such as infinite loop detection, data race checking, and diagnostics. Despite all that, Charm is still faster than all all variants of Sharm on most benchmarks.

In figure 7 we include Swift compilation times for our model checking benchmarks. In figure 6 we report the total time to compile and model check each program. Once again, if we measure execution time from when Sharm is invoked until model checking completes, the fastest Sharm model checker on all programs except for DinersFixed is the optimized interpreting model checker. More investigation is needed to determine why interpreting model checking is slow on DinersFixed.

6 Conclusion

Sharm is a toolbox for Harmony bytecode: it implements an interpreting executor, a compiling executor, an interpreting model checker, and compiling model checker. Our key optimizations were to use static dispatch where possible, to use protocol-oriented programming to gain back the advantages of object-oriented programming, and to perform basic block optimizations in compiled code.

Our evaluation shows that, despite our best efforts, Charm is still faster than Sharm at model checking. However, the model checking performance of the optimized interpreting model checker is within reach of Charm. With more optimization and symmetry reduction techniques, we may eventually be able to bring Sharm's performance in line with Charm.

Ignoring Swift compilation time, the optimized compiling executor / model checker produce slightly faster executables than their interpreting counterparts. However, since Swift compilation with optimization adds significant overhead, the fastest end-to-end Sharm executor and model checker were the optimized interpreting versions, which shows that much of the benefit of implementation-language-level optimization can be applied with just an interpreter. The developers of Charm are likely justified in their implementation of an interpreting model checker over a compiling model checker.

7 Acknowledgements

Thanks to Robbert Van Renesse for working closely with me on Harmony. Thanks to the reviewers who provided early feedback on drafts of this paper.

References

[1] Swift. `https://www.swift.org/`.

[2] Swift enumerations. `https://docs.swift.org/swift-book/LanguageGuide/Enumerations.html`.

[3] Swift strings. `https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID286`.

[4] Swift strings. `https://docs.swift.org/swift-book/LanguageGuide/StringsAndCharacters.html#ID292`.

[5] Writing high-performance swift code. `https://github.com/apple/swift/blob/main/docs/OptimizationTips.rst`.

[6] Protocol-oriented programming in swift. `https://developer.apple.com/videos/play/wwdc2015/408/`, 2015.

[7] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 174–186, New York, NY, USA, 1997. Association for Computing Machinery.

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

[9] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), Oct. 2009.

[10] R. van Renesse. Harmony. `https://harmony.cs.cornell.edu/`.

[11] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pages 3–11, 2000.

[12] Y. Yu, P. Manolios, and L. Lamport. Model checking tla+ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods*, pages 54–66, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

8 Appendix: Reviewer Concerns

Review #4A

Weakness: "... it is not clear what advantages SHARM provides over CHARM from a user perspective."

I added a paragraph in the introduction discussing when a user should choose Sharm or Charm. Charm is better for model checking and is better integrated with Harmony, but Sharm has the ability to execute as opposed to model check.

Comment: "I don't think it's very necessary to go into Harmony/Swift language features."

I removed this section.

Review #4B

Weakness: "... it would be nice to reframe the project (title and abstract) to not just be about the model checker. . . "

I redid the introduction to discuss the design space of code tools and include model checking as one half of the design space. The first dimension is executors vs. model checkers and the second dimension is interpreters vs. compilers. I include a section about how I intend for Sharm touch all four quadrants of the design space.

Comment: "I think you can talk less about specific Harmony features (section 3) and just redirect people to the Harmony book/documentation."

I removed this section.

Comment: "I think it would also be easier to read if the implementation and high level design choices (section 4) can be separated into different sections."

I separated the Implementation section into High-level Overview and Implementation Details sections.

Review #4C

Weakness: "One weakness is that the SHARM tool is not well motivated. What do you hope to achieve using this work?"

I added a paragraph in the introduction discussing my motivation for Sharm to be a general-purpose toolbox for Harmony code, and for it to be easier to extend and develop than Charm with comparable performance.

Weakness: ”Furthermore, you could describe what you learned in the process of building SHARM, the advantages/disadvantages of using Swift and the difficulties faced in each of the focus areas of SHARM.”

I added a subsection of the introduction discussing some of the features of Swift which I used in the project.

Weakness: ”It seems the development of SHARM has an experiential aspect to it so including it in the paper would improve it.”

I agree, though I will not be able to accomplish this before the camera-ready deadline.

Comment: ”The distinction between stateful and stateless model checking is not made clear...”

I am frankly not well read up on this space. The rest of the comment includes several terms which I am unfamiliar with. I am unsure how to address this comment.

Review #4D

Weakness: ”A weakness of the author’s work is the absence of a well documented artifact of their implementations – this is fixable and I am sure the author intends to do it later on.”

I added a README to the GitHub repo with instructions on how to build the project and walk the user through how to run two demo programs.

Weakness: ”Overall, as previously mentioned, the paper does a great job of describing implementation decisions and motivations for each of the individual technical contributions. However, there are a few exceptions that appear to be fixable but would probably require some effort, as described in the following comments.”

I address the reviewer’s comments in the next few points.

Comment: ”In Section 4.1, the author should provide a more detailed description of Harmony values and their Swift interpretations.”

I removed this section because it was more of a diversion and not necessary for the core ideas of the paper.

Comment: ”In Section 4.2, the reader wonders why the protocols in the Nondeterminism and OpVisitor modules are not described in more detail.”

I added a section discussing protocol-oriented programming and how it was used in Sharm.

Comment: ”It is mentioned that the stateless model checker is neither sound nor complete; I suggest that the author describe what they mean more specifically in their write-up when the implementation is complete.”

I included this as an off-handed comment which I cannot back up factually. I removed this sentence.

Comment: ”Regarding the compiler, how does the author intend to validate their compiled programs? ... Could it be possible to have a compiler that doesn’t follow the Harmony language specification (supposing that there were formally such a thing) but compiles to a few Swift programs that model check just fine, and in fact the model checker performs much better than both Charm and the stateful model checker? That would be concerning.”

This is a valid concern which I frankly cannot address.

Comment: ”Can the author provide more clarity on the “-O” flag: what optimizations does this entail?”

I intend to fully flesh out this section when I do the evaluation. Also, since I am not a Swift compiler expert, I’m not sure what optimizations it entails either.

Comment: ”Please update your GitHub repo with a good description of build instructions and dependencies, plus some well documented examples. I am 100% sure you intend to do this, but I couldn’t not mention it!”

As mentioned above in the weakness, I added a README with a brief tutorial that should help users get up and running.

Comment: ”Latex quotes should be fixed throughout the document.”

I think the reviewer means citations? I redid all the citations to all be in parentheses at the end of the sentence that cites the resource.

Comment: ”In the bulleted points summarizing the contributions of the work (at the end of the Introduction) can the author include the sections in the paper where each of the contributions are described?”

I did just this.