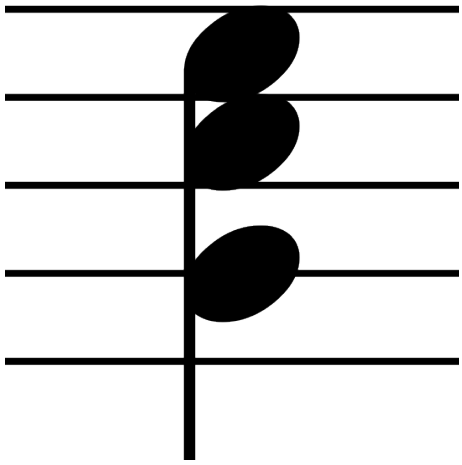
How Harmony is Composed



Robbert van Renesse

Contents

Chapter 1

Introduction

This document describes the Harmony code base. In a nutshell, when you "execute" a Harmony program, the following steps are taken:

- Using a Python3 embedding of Antlr4, the Harmony program is parsed into an Abstract Syntax Tree (AST);
- Using Python, the AST is used to generate Harmony Virtual Machine (HVM) code in the form of a JSON file with the extension `.hvm`. The Python code is capable of executing HVM code for the purpose of constant folding;
- The JSON file is passed to "Charm" (for C Harmony), a model checker built specifically for HVM in C.
- Charm builds a "Kripke structure," a directed graph whose nodes represent reachable states and whose edges represent executing HVM instructions (which change the state).
- Assuming no errors were found along the way (assertion failures, divide by zero, uninitialized variables, etc.), Charm also analyzes the Kripke structure upon completion to look for deadlock ("non-terminating states"), race conditions, and busy waiting.
- If some problematic issue was found, Charm outputs the shortest path to such an issue. If not, but there was **print** output, Charm outputs the entire Kripke structure.
- The output of Charm is handled in Python. It can generate visualization of problematic issues. If there are no issues, it interprets the Kripke structure as a Nondeterministic Finite Automaton (NFA), with the transitions labeled using zero or more output values from Harmony **print** statements. This part is mostly used to generate "Harmony Finite Automata" (HFA) files, which can be used as an input to the model checker to compare **print** behaviors of two different Harmony programs.

1.1 Value Representation in JSON

Many of the intermediate representations are in JSON format, and in particular Harmony values it is useful to know how Harmony values are represented in JSON. Each Harmony value is encoded as a JSON dictionary with a "type" and a "value" field. The "type" field is one of the following:

"bool": a boolean.

"int": a integer.

"atom": a string.

"set": a set.

"list": a list.

"dict": a dictionary.

"address": an address.

"pc": a program counter.

"context": a context.

The "value" field describes the type-specific value of the Harmony value.

"bool": "False" or "True".

"int": a integer.

"atom": a string.

"set": a sorted list of the values in the set (recursive).

"list": a list, recursively containing other Harmony values. eitem[] "dict": a list of mappings sorted by key. Each mapping is a dictionary containing a "key" and a "value".

"address": an address consists of a function "func" and a list of arguments "args". **None** is an address without a function or arguments. The function `PC(-1)` represents the function that maps global variable names to values.

"pc": an integer program counter.

"context": a dictionary describing the context (see below).

A context has the following fields (each are Harmony values themselves):

"pc": the program counter value.

"sp": the integer stack pointer (size of the stack).

"vars": a dictionary containing the local variables of the current method.

"atomic": the "atomic counter" of the context. Usually only included if larger than 0, when the context is in atomic mode.

"readonly": the "readonly counter" of the context. Usually only included if larger than 0, when the context is in readonly mode.

"atomicFlag": whether the context is in atomic mode. Usually only included when the context is in atomic mode.

"interruptlevel": the integer interrupt level of the context. Usually only included if larger than 0, when the context is handling an interrupt.

"stopped": a boolean indicating if the context is currently stopped. Usually only included if so.

"terminated": a boolean indicating if the context has terminated. Usually only included if so.

"failed": a boolean indicating if the context has failed. Usually only included if so.

"eternal": a boolean indicating if the context is eternal. Usually only included if so.

"trap_pc": the program counter of the trap handler if any.

"trap_arg": the argument to the trap handler.

Chapter 2

Parser and Code Generator

The parser is responsible for processing the Harmony input files and producing Harmony Virtual Machine (HVM) code that the model checker can evaluate. The input files are a collection of Harmony modules. One of those is given to the parser and it is given the name *__main__*. The module may import other modules, and this may go one recursively. Each module is incorporated at most once.

The Harmony grammar is given as an ANTLR4 file "Harmony.g4" in the top of the Harmony directory hierarchy. Another important file is "harmony_model_checker/parser/antlr_rule_visitor.py". This file generates the Abstract Syntax Tree (AST) that is the input to the code generator. Essentially, there is an entry for each rule in the grammar that describes how to represent that rule.

The input is first divided into tokens, which are four-tuples (*lexeme*, *file*, *line*, *column*), where *lexeme* is a string containing the text representation of the token, *file* is the module, and *line*, *column* its location in the module.

An AST represents the syntactic structure of the Harmony program. For example, the program $x = y + z$ might have a root node that represents *assignment*. The left branch would point to a leaf node representing the variable x. The right branch would point to a node representing addition. That node has branches for the variables y and z.

The AST data structure, along with the code for code generation, is defined in file "harmony_model_checker/harmony/ast.py". **class** `AST` is the base Python class for all the nodes in the AST. Each node in the AST represents a sequence of tokens, represented by *token*, *endtoken*, which are the first and the last token (which are guaranteed to be in the same module). In the case of leaf nodes in the AST, *token* == *endtoken*. A node also has a boolean field **atomically** that indicates that the node is (part of) an atomic operation.

Then for each type of rule, there is a child class. For example, **class** *NameAST* represents an identifier. Most of the types will define specialized versions of the following methods:

- *compile*(*self*, *scope*, *code*, *stmt*): this is the method that generates HVM code for this rule. The HVM code is added to the the *code* argument, which is of type **class** *Code* and defined in file "harmony_model_checker/harmony/code.py". Context for this, like the types of the identifiers in the scope, is defined in the *scope* argument which is of type **class** *Scope*, defined in file "harmony_model_checker/harmony/scope.py". *stmt* is a four-tuple *line1*, *column1*, *line2*, *column2* indicating what part of the Harmony source code is being parsed.

- *gencode*(*self*, *scope*, *code*, *stmt*): same as *compile*(), but does not do constant folding. (Note: constant folding is currently disabled in any case until we can find a way to invoke Charm to evaluate constant expressions.)
- *address*(*self*, *scope*, *code*, *stmt*): like *compile*(), but computes the address of the expression (which cannot involve an operator), rather than the value.
- *ph1*(*self*, *scope*, *code*, *stmt*): assignment statements have a left-hand side and a right-hand side, and are evaluated in three steps. First, *ph1*() pushes the address for the left-hand side onto the stack. (Sometimes, there may be multiple addresses, as in $x, y = 1, 2$ or $x = y = 0$). Then the right-hand side is evaluated using *compile*() to push the value to be assigned onto the stack. Then *ph2*() stores the value.
- *ph2*(*self*, *scope*, *code*, *skip*, *first*, *last*, *stmt*): Stores the values in an assignment statement.
- *isConstant*(*self*, *scope*): returns whether the rule is a constant expression.
- *getLabels*(*self*): returns the set of labels defined by this rule. A label is a pair of a token and a value of type **class** *LabelValue*. Each method name is a label, and statements of the form *label*: *statement* have labels. These LabelValues have to be turned into PcValues (program counters values) during a phase of compilation called *linking*.

2.1 Harmony Values Representation in Python

The file "harmony_model_checker/harmony/value.py" deals with the representation of Harmony values in Python. **class** *Value* is the base class, although simple Harmony values like booleans, integers, and strings are represented as is. Note that the Harmony compiler sometimes refers to strings as *atoms*. Other values (sets, lists, dictionaries, addresses, contexts, and program counters) are subclasses of **class** *Value*. There is also a value type **class** *LabelValue* that represents labels in Harmony programs during compilation. During linking, those values have to be turned into program counter values. A program counter value is simply an index into the generated HVM code.

The module contains useful methods to turn values into other representations:

strValue(v): returns a string containing the Harmony programming language representation of value v.

jsonValue(v): returns a string containing the JSON representation of value v described in Section 1.1.

tlaValue(v): returns a string containing the TLA+ representation of value v.

2.2 HVM Output File

The output of the parser is a ".hvm" file, which is a JSON formatted file representing, among others, HVM code that the model checker can run.

The JSON schema is as follows. The file contains a JSON dictionary including the following fields:

"modules": a dictionary describing each of the input modules used.

"code": a JSON list containing the HVM code.

"locs": a JSON list that describes, for each HVM instruction, what source code it corresponds to.

"pretty": human-readable version of the HVM code.

2.2.1 "modules"

This is a JSON dictionary that maps module names to information about that module. The main source file is in a module called "__main__". For each module, a JSON dictionary has the following information:

"file": the source file name of the module.

"lines": a JSON array containing the lines of the file.

"identifiers": a JSON dictionary describing, for each identifier used in the module, what type it is.

The identifier types are as follows:

"local-const": an immutable method variable such as an argument to a method or a bound variable of a **for** loop.

"local-var": a mutable method variable.

"constant": a module constant, including method names.

"global": a mutable shared variable.

"module": the name of a module.

2.2.2 "code"

The "code" entry is a list of HVM instructions. Each HVM instruction is encoded as a dictionary. The "op" entry in the dictionary describes the type of instruction (which are listed in the Harmony book). For example, "Frame" is an HVM instruction that is the first instruction of a method, while "Return" is the last instruction of a method. Depending on the type of the instruction, the dictionary may contain additional information.

[Add per instruction description here.]

2.2.3 "locs"

The list "locs" has the same length as "code" and contains one entry per HVM instruction. Each entry is a dictionary that tells more about "where the instruction came from" in the Harmony source code. The Harmony source code can be thought of as a sequence of Harmony statements. Each statement starts with a "start" token and ends with an "end" token. The dictionary has the following fields:

- "module": the module of the statement.
- "line": the line number of the start token.
- "column": the index of where the start token begins on the line.
- "endline": the line number of the end token.
- "endcolumn": the index of where the end token ends on the line.
- "stmt": the part of the statement that is being evaluated.

The "stmt" field contains four fields [line1, column1, line2, column2] that represent what part of the statement is being evaluated. For example, if the statement is $x = y + z$ in line 3 of the source file and the instruction is `Load y`, then "stmt" would be [3, 5, 3, 5], as the token y is being evaluated and it is in line 3 starting and ending at column 5.

2.2.4 "pretty"

Like "locs", "pretty" is a list with one entry per HVM instruction. Each entry is a tuple with two string subentries. The first subentry presents a more human-readable presentation of the HVM instruction. The second subentry describes in English what the instruction does.

Chapter 3

Charm: Execution

Charm is a C program that reads a ".hvm" file containing a Harmony Virtual Machine program and produces a ".hco" (Harmony Charm Object) file. Charm can also optionally take in a ".hfa" file containing a Harmony Finite Automaton describing the behavior of some other Harmony program. The source code for Charm is in the `charm` folder.

This chapter describes the first phase of what Charm does: producing a "Kripke structure," that is, a directed graph containing all reachable states of the HVM program. A "state" describes the values of the variables as well as the (multi-)set of "contexts." A context is the state of a thread, and in particular contains the program counter of the thread, its local variables, and its stack. An edge between two states corresponds to a "transition": an execution of part of the HVM program by one of its threads.

As Charm tries to find the shortest possible path to a problematic issue, Charm generates the graph in a breadth-first fashion. It starts with an initial state in which there is only a single thread (called *__init__*). This thread first explores the states that are reachable from this initial state in one transition. Because of the non-deterministic **choose** operation, there can be more than one such state even if there is only one thread. Next it explores states that are reachable in two steps, in three steps, and so on.

The initial thread runs in "atomic mode," meaning that it cannot interleave with other threads. But it can spawn other threads that can start running once the initial thread has completed. So, there are two kind of branching that can happen in the Kripke graph: those caused by **choose** expressions and those caused by threads interleaving. In fact, there is a third kind of branching, which is caused by interrupts (as a result of the Harmony **trap** statement).

There are two kinds of states:

1. "choosing state": a state in which one of the threads has reached a **choose** instruction. There is an edge from this type of state for every possible choice the thread can make. The set of choices is on the top of the stack.

2. "normal state": a non-choosing state in which two or more threads can make a transition. There is an edge from this state for each such thread. There is also an interrupt transition for each thread that has set a trap.

While in theory Charm could create a transition for every HVM instruction, Charm applies a form of an optimization technique called "Partial Order Reduction" to reduce the number of states

in the Kripke structure, making it significantly more scalable. The optimization is based on the observation that operations that only modify the state of a thread are not directly visible to other threads, and so there is no value in trying all possible interleavings. For example, suppose thread X pushes the values 1 and 2 onto its stack, while thread Y pushes the values 3 and 4 onto its stack. There are 6 possible interleavings of these four instructions, but they all have the same effect.

In the absence of threads executing atomically, Harmony only tries interleaving when threads try to load, store, or delete shared variables, when a thread enters atomic mode, or when a thread executes a **print** statement. After a thread has entered atomic mode, that thread continues executing until it leaves atomic mode. When not in atomic mode, a thread continues executing until it hits a choose, load, store, delete, or print operation, or until it goes into atomic mode. If interrupts are possible, then a thread also executes its last instruction (a return from its entry method) as a separate transition to allow for an interrupt to happen at the last possible moment before the thread terminates.

3.1 The Graph Representation

Key to making model checking efficient is how the graph is stored. The main definitions for how the graph is represented is in a file called `graph.h`. Here you find definitions for `struct node` and `struct edge`. The most important fields in `struct node` are:

- `id`: a unique integer identifier for this node;
- `state`: the HVM state corresponding to this node, which includes the values of the shared variables and the contexts. Importantly, there is a one-to-one correspondence between nodes and states;
- `fwd`: a linked list of outgoing edges;
- `bwd`: a linked list of incoming edges;
- `to_parent`: the incoming edge on the shortest path from the initial state;
- `len`: the number of edges on the shortest path from the initial state;
- `steps`: the number of HVM instructions executed on the path from the initial state;
- `next`: a pointer to another node, mostly used for maintaining a list of new nodes that still need to be "explored" (determining what states are reachable from this node).

The most important fields in a `struct edge` (which is directed) are:

- `src`: the source node of the edge;
- `dst`: the destination node of the edge;
- `fwdnext`: next outgoing edge from `src`;
- `bwdnext`: next incoming edge to `dst`;
- `ctx`: the starting context of the thread that made the transition;

- `after`: the ending context of the thread that made the transition;
- `choice`: the choice that the thread made, assuming the first instruction is a **choose** operation;
- `interrupt`: boolean that is set if this is an interrupt transition;
- `ai`: information about load and store operations that the thread made (to detect race conditions);
- `log`: the values that were printed in this transition;
- `nlog`: the number of values that were printed in this transition.

Note that the edge does not keep information about the individual HVM instructions that were executed. That information can be reconstructed if needed by running the thread again. But while model checking, it's important to keep memory usage low.

The nodes in the graph are also organized as a simple array of nodes in `struct graph` (also in `graph.h`).

3.2 Harmony Values

The "state" of a Harmony program consists of the values of its shared values and the contexts of its threads (including terminated ones). Values are hierarchically structured, as several Harmony value types are containers consisting of Harmony values.

In Charm, all values are represented by a 64-bit `hvalue_t`. The four least significant bits of an `hvalue_t` indicate the type of the value: boolean, int, string (aka atom), set, list, dictionary, address, program counter, context (defined in `value.h`). In the case of a boolean, int, or program counter, the remaining bits contain the rest of the value. In the other cases, the remaining bits point to a structure that describes the contents of the value:

- a `string` (or atom) is represented as an array of bytes in UTF8 format;
- a `list` is represented as an array of its `hvalue_t` elements;
- a `set` is represented as a sorted array of its `hvalue_t` elements;
- a `dictionary` is represented as an array of its (key, value) pairs, sorted by key. Thus, if a dictionary has n keys, the representation has $2n$ `hvalue_t`'s;
- an `address` is represented as an array of its path elements (much like a list). The first element is always a string representing the name of a shared variable;
- a `context` is represented by a `struct context` structure (defined in `value.h`) and further described below.

A context (the state of a thread) includes the following fields:

- `pc`: the program counter;
- `thestack`: an array of `hvalue_t`'s;

- `sp`: the stack pointer;
- `vars`: the local variables of the method that is currently executing;
- `atomic`: the "atomic counter," incremented when the thread enters an atomic section and decremented when it leaves one. If the atomic counter is non-zero, the thread executes atomically;
- `readonly`: the "readonly counter," which works similarly as the atomic counter. When in readonly mode, the thread cannot modify shared variables;
- `terminated`: the thread has terminated;
- `failed`: the thread has failed;
- `stopped`: the thread has stopped (the result of a **stop** or **save** operation);
- `eternal`: the thread is eternal;
- `extended`: the context is "extended."

In order to keep contexts small (they are a major contributor to memory usage during model checking), some rarely values are only there when the thread is "extended." They are:

- `ctx_this`: thread-local state;
- `ctx_failure`: a string value containing a description of what went wrong;
- `ctx_trap_pc`: a program counter value of a Harmony **trap** that was set;
- `ctx_trap_arg`: the argument to the Harmony trap.

These values occupy the first four elements of the stack if the `extended` flag is set. The macro `ctx_stack(ctx)` can be used to find the bottom of the stack in a context independent of whether it has been extended or not.

The module `value.[ch]` has various convenient routines to manipulate Harmony values. For each container type T, there is a `value_put_`T() function that turns it into an `hvalue_t`. Vice versa, the function `value_get()` turns an `hvalue_t` for a container type into its value (an array of bytes) and size. For each non-container type, macros `VALUE_TO_`T() and `VALUE_FROM_`T() fulfill similar roles. `VALUE_TYPE(v)` returns the type of an `hvalue_t`.

3.3 Hashdict

Importantly, two Harmony values are the same if and only if they have the same `hvalue_t`. To ensure this invariant for container types, it is necessary to look up a newly generated Harmony value in order to make sure no duplicates are constructed. For example, if we have the list [1, 2] and add the value 3 to the end, we end up with a new list [1, 2, 3]. We have to check if this list was already assigned an `hvalue_t` and if so what it is.

To this end, Harmony maintains hash tables (of type `struct dict`) that maps container values to `hvalue_t`. Currently Harmony has one for each such value type (`struct values`). This was

done to try and reduce lock collisions due to concurrency, although in retrospect it does not actually seem to help much—at a later time these different hash tables may be merged into a single hash table.

The implementation of the hash table is in the files `hashdict.h` and `hashdict.c`. The concurrent performance of the hash table is critical to the performance of the model checker. The hash table uses "open hashing," which means that each bucket maintains a (linked) list of key/value pairs. In fact, each bucket maintains two such lists. There is a *stable* list of key/value pairs that is read-only and whose access does not require a lock. It is always checked first during a lookup operation. The *unstable list* requires a lock for access.

The hash table uses locks. A lock may be large (64 bytes—the size of a cache line), so instead of a lock per bucket, there is a fixed size list of locks per hash table. That is, multiple buckets share the same lock. When the hash table's array of buckets is resized, the locks don't have to be moved or recreated.

The hash table implementation leverages the way the model checker works. Essentially each round of model checking, which adds new states at a particular distance from the initial state, consists of three phases (separated by barrier synchronization):

1. Evaluating all transitions from the last collection of new states. During this phase, the number of buckets in each of the hash tables remains static.

2. At this time it is known how many unstable entries have been added to the hash table. A new size is computed and if needed a new array of bucket is allocated but not yet filled. This does not take much time.

3. If the array was resized, the existing entries are rehashed and moved into their new location. Then the unstable entries are moved straight into their final stable location. This can be done in a trivial parallel fashion: each thread deals with a part of the array.

To enable this, the hash table maintains information for each thread (worker). In `struct dict_worker` (one per worker), the hash table keeps track of how many unstable entries it has created. Moreover, it also generates a list of unstable entries for each other worker (remember, the buckets are subdivided among the workers in the last phase). During the third and final phase, each worker can iterate over the unstable entries generated for its buckets and add them to the stable list.

While the hash table supports mapping to a particular value, we only use it in this case as a set, essentially. Each entry in the hash table is of type `struct dict_assoc`. This is just the header, which contains the length of the key and a pointer to the next entry in the same (stable or unstable) list. The header is immediately followed by the key. It is guaranteed that the header is aligned to 16 bytes so that the lower 4 bits of the pointer can be used to hold the Harmony value type.

3.4 State

State itself is a container of Harmony values. It is defined in `value.h` as `struct state`. It has the following fields (all `hvalue_t`'s):

- `vars`: a dictionary containing the values of the global variables. The keys are string values containing the names of the global variables;

- `choosing`: if not 0, the context of the thread that is making a choice;
- `stopbag`: a bag of contexts that have stopped;
- `dfa_state`: the state of the Harmony finite automaton that is being evaluated in concert with the Harmony program that is executing;
- `contexts`: a multiset or "bag" of executing contexts.

The context bag is represented as a sorted array of the `hvalue_t`s of the contexts in the bag, followed by a `uint8_t` array of multiplicities with the same number of entries. The reason contexts are kept in a bag is because multiple threads may be in exactly the same state. Threads in Harmony are not uniquely identified—this allows a dimension of scaling.

The `stopbag` is also a bag, but maintained instead as a Harmony dictionary that maps contexts to multiplicities. It's a little simpler to it that way, but at higher overhead. The `stopbag` is needed to be able to detect non-terminating states.

A hash table is used to map states to nodes in the graph, while each node directly points to the corresponding state.

3.5 Global Data

It's important to keep separate the state of the Harmony program (kept in `struct state`), and the global state of the Harmony Model Checker (kept in various places, including `struct global`, `struct worker` (one for each C thread), as well as some other global variables and state that is kept on the stacks of the C threads. Some examples of the global state have already been discussed: `struct node`, `struct edge`, and `struct dict`.

The variable `global->todo` points in the `struct graph` array of nodes. It points to the first node that is not scheduled to be "explored."

3.6 Main Loop

If there are n cores, Charm uses $n + 1$ C threads: 1 thread is a coordinator, while the other n threads do the hard work. Charm first explores all states that are reachable in one "step" from the initial state, then two steps, and so on. In other words, the graph is explored breadth-first. Let's call each such step an *epoch*. In each epoch, the threads compute new states and grow and stabilize the hash tables as needed. Since an epoch could consist of many states, this may not give enough opportunity to stabilize and/or grow the hash tables. Therefore epochs are limited to a certain number of states, currently 1024 per worker. Thus it may take multiple epochs to complete the computation of all new states in a step. An epoch explores all states starting from `global->todo` to `global->goal`.

Each epoch is further subdivided into three phases, separated by synchronization barriers:

1. the worker threads are computing the states that are reachable from the new states determined in the previous epoch. The coordinator simply waits for all workers to be done.
2. The coordinator grows the number of buckets in the hash tables as appropriate and computes the goal for the next epoch. The workers fix the forward edges they computed.

3. The worker threads stabilize the hash data, also moving data from the old buckets to the new buckets as needed. The coordinator simply waits.

There are three correspondings synchronization barriers; `start_barrier`, `middle_barrier`, `end_barrier`.

In the first phase, each worker executes method `do_work` and grabs a subset of the nodes that need to be explored. This requires a lock to manipulate `global->todo`, which points at the first unexplored node. While in theory a worker could just obtain one unexplored node at a time, doing so would be inefficient due to lock contention. So it uses some heuristics based on how many nodes are left to explore. For each state, the worker determines if it is a choosing state or a normal state. If it is a choosing state, it looks at the choices that the choosing context has and invokes `make_step` for each choice. If it is a normal state, then it calls `make_step` for each active Harmony thread.

Method `make_step` in turn calls `onestep`, a function that actually computes the state reachable from a given one given which context is making the transition, the choice it has (if any), and whether or not it is handling an interrupt. Method `onestep` executes Harmony HVM instructions, usually until it hits a load, store, or choose instruction (not counting the first).

Chapter 4

Charm: Analysis

Chapter 5

Visualization

Acknowledgments