SSRP 2001 Final Report: Making HCC Probabilistic

Jonathan Moody
jwmoody+@cs.cmu.edu

August 22, 2001

1 Overview

1.1 Goals

At the beginning of the summer, I was given the goal of "adding probability to HCC". My approach to the problem was guided by my understanding of the probabilistic programming language literature and consideration for the uses to which a probabilistic HCC would most likely be put at NASA Ames. In my readings, I had encountered several flavors or ways of describing a probabilistic semantics. One style spoke of the enumeration of all possible outcomes or states labeled with their probabilities. This is definitely a desirable formal model for a language, because all outcomes, even the most unlikely failures are generated and assigned exact probability values. Another was similarly precise in nature, but was based on a Hoare-logic type analysis of the program text, clearly unsuitable for an operational model. The third took a more operational point of view – At certain points in program execution, a non-deterministic choice is made, and program execution simply continues as usual along one branch of that choice. This third approach is much easier to implement, and I could forsee some serious practical difficulties with a strategy based on the first approach. Furthermore, I learned that HCC is currently used to generate single simulated trajectories of test data. The user needs were support for noisy modeling, and the generation of random failures, not exhaustive analysis of model behavior. Therefore, I decided to base my work on the simple, operational approach. This would be easy to complete within ten weeks, and would allow some time to explore other improvements to HCC.

1.2 Problem Description

HCC has a capability for calling foreign functions written in C. This can be leveraged quickly to get random values into HCC, but it is not always safe. These randomized function calls may appear in a constraint like "a * b = random(0,1)" or "if (a < gaussian()) P" which, may be checked/evaluated *m*ultiple times during constraint propagation. In certain situations, this creates unexpected inconsistencies in the store of active constraints. The problem never arises in C, because the C language has a sequential semantics: statements are evaluated only once, in a well-defined order. To fix the problem, we must find a way to assign a single, well-defined value to these types of function calls, or when modeling random processes, a sequence of values over simulated time. The semantics of such expressions should be independent of the implementation details of the constraint solver. For example, the value of "random(0,b)" should not depend on whether a constraint on "b" occurs "before" it in the program. Care must be taken to choose a well-defined time (during constraint propagation) at which to evaluate these expressions.

1.3 Probabilistic Choice

A new type of constraint, the "choose" constraint was added to HCC. It was inspired by several papers on probabilistic concurrent constraint programming, including one by Vineet Gupta. However, it does not directly implement the semantic model presented in these papers. Its meaning is given operationally, as hinted at above.

```
choose <variable> from <expression>;
```

The <expression> is intended to be a function call to an external random number generator. Thus any distribution can be supported; new functions to generate samples from them can be added to a C library and called via HCC's foreign function interface. At certain step in HCC's constraint processing, <expression> is evaluated, and <variable> is constrained to that particular value. After evaluation of <expression>, the "choose" constraint behaves like "<variable> = <constant-value>", a constraint with no safety issues.

When, then, to evaluate? The difficulty is caused by variables which may appear in the expression (possibly as arguments to the random number generator routine). If no variables were allowed in such expressions, then evaluation could be done at any time. The presence of variables, whose values may change over the course of constraint propagation, requires more care. Several alternatives were considered, but in the end, it was decided to defer evaluation until the variables in the expression are constrained to "fully-determined" values. A numerical value is considered "fully-determined" when it is constrained to a single point (as opposed to an interval), and other data types are "fully-determined" when they are constrained to any value (besides the unknown value). I later discovered this strategy was analogous to the method of "residuation" in logic programming (see the long report).

1.4 Modeling Random Processes

Random processes are distinct from random choice, because they involve a value that changes randomly over time. They are the mathematical objects that play a key role in the formulation of stochastic differential equations (SDE's). Although there are methods for numerical integration of SDE's, I did not feel comfortable digging too deeply into HCC's numerical integrator. Rather than further modifying the semantics of HCC, I simply used the new probabilistic choice construct to model explicitly a noise process "inside" of HCC itself. This noise process has a characteristic time-scale (Δt) which determines how often a new random value is chosen. A true, mathematical noise process does not depend on such a parameter. But having it available allows the user to choose what level of fidelity is desirable, keeping in mind that more frequent changes in value require more processing overhead.

2 Sampling the Results of HCC Computation

Although HCC with "choose" has a probabilistic semantics, one cannot gather much useful information from a single simulation run. The probability of a particular outcome is implicit, hidden in the semantics of "choose". To obtain estimates for the probabilities of various outcomes, it is necessary to run multiple sample simulations with the HCC interpreter.

Toward this end, I designed and implemented two mechanisms for collecting output (in an abstract sense) from HCC. The first is simply a script which invokes HCC some number of times. Any printed messages that the model produces while running are redirected to a file. By embedding the special "sample" statement in the model, trajectories for all the sample runs can be collected and plotted. The second approach is an improved C$\rightarrow$HCC interface, which allows a C program to (among other things) run multiple instances of the HCC interpreter simultaneously, and extract values of variables, compute a mean across samples, compute probability of an event across samples, etc. This improved interface is working, but not yet complete.

2.1 Conclusions

Overall, I feel my project was successful in meeting the goals I set. Ten weeks is not a lot of time however, and I had to rule out some approaches for the sake of expediency. I am most excited about some new things I learned this summer about logic programming, specifically how it can be integrated with other programming language paradigms. I will stay in touch with the Model-Based Autonomy group at NASA Ames, and I plan to continue working on HCC for a few weeks after returning to CMU. I feel there are still some interesting issues to be explored, and some cleanup of the HCC source code to be done.

Long Report (chronologically organized)

3 Goal (from proposal)

The purpose of my work is to enable HCC modeling of noise (both in initial conditions and time-varying, dynamic noise) and random failures. It should be possible to accomplish this through a combination of modeling tricks in HCC and addition of a new statement ”choose”. The choose statement will allow calls to impure native functions (such as a random number generator) to be evaluated in a controlled way.

4 What is the problem?

To summarize, HCC has a capability for calling ”native” functions written in C. This can be leveraged quickly to get random values into HCC, but it is not safe because these function calls may appear in a constraint like ”a * b = random(0,1)” or ”if (a ¡ gaussian()) P” which, in turn, may be checked/evaluated multiple times. In certain situations, this creates inconsistencies in the store of active constraints. We must find a way to assign a single, well-defined value to these types of function calls, or when modeling random processes, a sequence of values over simulated time. The semantics of such expressions should be independent of the implementation details of the constraint solver. For example, the value of ”random(0,b)” should not depend on whether a constraint on ”b” occurs ”before” it in the program. Care must be taken to choose a well-defined time (during constraint propagation) at which to evaluate these expressions.

4.1 Attempt 1

I looked into using existing HCC constructs like ”eval” and function application, but neither can be used to force evaluation at the right time. Function application is call-by-name in HCC (no evaluation occurs). The eval construct has no effect on evaluation during constraint propagation; replacement of an eval expression with its value is deferred until the end of the phase.

For example, in ”x = ... eval(random(0,1)) ...”, random may be called many times, and the store may become inconsistent. Only at the end of the phase will this constraint become x = <constant> (a constant which will likely be different from the value x had during the phase proper!)

Deferring replacement of eval with its value has advantages: the end of the phase is when the maximum amount of information is known about variables in the expression. Forcing evaluation earlier could prematurely freeze the value of variables when they were (partially) unknown.

4.2 Attempt 2

I implemented a "choose" statement for introducing randomness. The syntax is inspired by the paper by Vineet Gupta (Gupta 1999). But instead of a finite, explicit representation of the distribution from which a value is chosen, HCC's "choose" accepts an arbitrary expression. It is intended that the expression contain a call to a random number generator routine.

```
choose <var> from <expr>
```

The first implementation of "choose" simply evaluated <expr> to a constant at the moment the constraint was added to the store, continuing as if "<var> = <constant>" had been told. However, forcing evaluation at this arbitrary instant does not treat the variables in such an expression in a consistent way. For example:

```
mean = [-1,1];
choose x from gaussian(mean);
mean = 0.5;
```

Assume the simple semantics (described above) in which the <expr> in choose is evaluated to constant c, and the constraint x = c is told to the store in the usual way. Which value of mean is used to generate the gaussian sample? The choice is dependent on the order in which the constraints are processed, an unacceptable violation of abstraction around the constraint solver. The difficulty is in finding a single correct time during constraint processing to evaluate such expressions.

There is no similar problem with "eval" because replacement of eval expressions with their values occurs at the end of a phase, when the constraint solver has reached quiescence. However, if choose defers evaluation until the end of a phase, then new information can be added to the store and the phase is no longer at an end.

```
b = [-1,1];
choose x from (b + random(0,2));
if (x > 0) b = -0.5;
```

4.3 Attempt 3

After more discussions with Benoit Hudson and Adam Sweet, it became apparent that a more general mechanism for controlling evaluation would be desirable. In addition to the case of randomized functions, Adam had encountered situations in which he needed to control evaluation of a call to a computationally expensive external procedure. I had several more discussions with Benoit and searched for related work on the issues we were facing. I found some literature on VHDL (another modeling language) and from the field of Functional Logic Programming.

Eventually we settled on an expression-level operator "once" which would protect expressions from unnecessary re-evaluation, returning a memoized value consistent with previous evaluations. Occurrences of "once(<expr>)" are evaluated under the following conditions:

- 1 all variables in <expr> are constrained to fully determined values AND
- 2
  - 2a <expr> has not been previously evaluated in the same generation OR
  - 2b one of the variables has been "unset" because a constraint was retracted

Thereafter, a stored value (the previous result of evaluation) is returned without further computation. A variable is considered "fully determined" when (for intervals) it is constrained to a point, or, for other data types, it has a value (as opposed to being unknown). A "generation" is an instance of one of HCC's three phases (point, interval, and trace). Each phase has a unique integer ID that is incremented before each new phase of constraint processing. Note that "once" does not really mean evaluate only once; condition 2b is necessary to deal with the following case.

```
storeval(c,5);
when(time=0.5) {
storeval(c,6);
d=once(c);
}
```

Is d equal to 5 or 6 at time t = 0.5? Clearly, it should be 6 because storeval(c,6) exists as a constraint on c at time t = 0.5, even if it has not been processed before d=once(c). Without condition 2b, the answer depends on the order in which constraints are processed. Without default logic, it would be impossible to tell c = 6 to the store while c = 5 is true. But storeval includes a "hencewatching" statement (a default ask) which can retract c = 5 when c = 6 is told.

I kept the "choose" constraint, but it became sytactic sugar for "once":

```
choose x from <expr>; ==>  x = once(<expr>);
```

My reasoning is that in the future, it might be possible to implement a different strategy for evaluation of probabilistic programs. The "choose" constraint type provides a syntactic hook to differentiate probabilistic choice from a normal native function call.

I also added the capability to label functions as "impure" – meaning that syntactically identical call to the function could evaluate to different values depending on subtle aspects of the context or the state of the outside world. Compile-time warnings are generated if a call to an impure function occurs outside of a "once" or "choose". This will not catch all such errors – it is possible to call an impure function indirectly by wrapping it in an HCC function.

5 Search for Related Work

5.1 VHDL

VHDL is a modeling language used to describe electronic components and integrated circuits. Like HCC, it attempts to describe a concurrent system declaratively, but it also has a sequential syntax for implementing primitives and constructing "black-box" components.

It is interesting to see how VHDL's designers dealt with some of the same issues faced by HCC.

VHDL has something analogous to HCC's point/interval phase distinction as well. In VHDL, constraint processing occurs in a sequence of delta cycles at a single instant in time. After this iterative solving process is complete, the VHDL interpreter proceeds to the next event in its list.

VHDL also has the concept of a "postponed" assignment, the semantics of which are similar to HCC's "eval". In HCC, "eval" expressions are replaced with their value at the end of a phase, in the same manner, postpone is used in VHDL to delay assignment until the "definitive" end of the phase, when all signals have reached their stable values. It is a runtime error to alter signals in a way that necessitates more delta cycles (Thirunarayan 1996).

However, when it comes to randomness, VHDL's design does not offer any guidance. If a system of "constraints" cannot be solved in a finite number of delta cycles (because of, for example, cyclic dependencies or careless use of random values), the interpreter simply does not terminate. So the definition of VHDL ignores the problem, leaving this up to the user to detect and resolve.

5.2 Functional Logic Programming

Constraint programming languages are based on a notion of entailment between constraints. HCC's entailment relation is specified operationally as an constraint solver. One can imagine an alternate specification as a logic program:

```
eq(X,X).
eq(X,Y) <- eq(Y,X).
eq(X, Z / Y) <- eq(X * Y, Z).
false <- eq(X,A), eq(X,B), !eq(A,B).
....etc....
<(X,Z) <- <(X,Y), <(Y,Z).
<(X,Y) <- eq(sign(X - Y),-1).
false <- <(X,Y), <(Y,X).
```

So a system of constraints could be translated as follows:

```
a * b = f(c) <=> eq(a * b, f(c));
b = 2 <=> eq(b,2)
c = 0 <=> eq(c,0)
```

One can imagine that instead of the iterative local propagation solver that HCC uses, asks become active when it is possible to prove the query in their guard.

```
if (a < 2) P

Rule or Fact Constraints/Bindings
----------------------------------------------------------
eq(b,2) {b = 2}
eq(c,0) {c = 2}
<(a,2) <- eq(sign(a - 2),-1) {sign(a - 2) = -1}
eq(a,f(c)/b) <- eq(a * b, f(c). {a = f(c)/b}
```

Quickly, the logic program accumulates a set of constraints that look similar to the original HCC program. Some of the constraints accumulated in proof search cannot be verified by the usual syntactic unification. Now the prover must find some way to rewrite the terms using definitions (in some form) of functions f and sign.

The Functional Logic programming community has been wrestling with these issues for some time. In the most general setting, the problem can be characterized as: "How does one reason about equality of functional terms?" A logic programming language typically defines equality (=) in terms of simple syntactic unification. In Functional Logic programming, (=) is extended to cover functional terms. There are two families of operational strategies for solving such equations: narrowing and residuation. (Hanus 1994).

Narrowing, while being a complete method, requires that programs be "constructor-based". In other words, functions must be defined in a case-based style:

```
last(X:[]) = X.
last(X:R) = last(R).
```

Thus last(a) can be rewritten as either "a" (with additional constraint a = X:[]) or "last(R)" (with the additional constraint a = X:R). In this way even function calls on unknown variables can be "evaluated". But this is clearly not applicable to HCC's problem with randomized native functions because we do not have access to the function definitions; they are black-boxes of compiled C code.

The second strategy is called residuation, and can be summarized as deferring evaluation of function terms until all arguments are known values (as opposed to variables). It is incomplete, but it is compatible with the black-box view of functions. An analogous strategy was chosen for "once". Once expressions are only evaluated if all of the variables inside have known values, and these values are points (when dealing with intervals). Partial information such as (x = [a,b]) does not help – there is no way to be sure x = [a,b] incorporates all all available information about x at the moment the expression is evaluated.

6 Modeling random noise processes in HCC

When one says that a term in a DE is "random", there is a choice of how to interpret this statement. The term could represent some uncertain initial value or parameter, or it could represent a time-variable stochastic process.

One can use "choose" to fix (random) initial conditions or parameters of a DE, but this alone does not allow modeling of stochastic differential equations (those with stochastic process terms). Furthermore, simply calling an external random number generator function from a DE constraint has unpredictable results. The trace-phase solver needs to "back up" in simulated time to find the instant at which a condition becomes true. But new random values will replace the old when the expression is re-evaluated, leading to a slightly different solution. The adaptive step size algorithm interprets this as a numerical error, and slows to a crawl as it reduces its step size to the minimum allowed.

6.1 Attempt 1

To fix this problem with modeling random processes, one must make the random value dependent on simulation time, as opposed to an unpredictable dependency on the global RNG state. Using the choose construct, I defined a noise process along the lines of the following code. It chooses a new random value for b every 0.1 time units. By using objects, it is possible to have multiple noise process instances of various types.

```
process(interval b, interval t) {
     interval x;
     choose x from gaussian();
     storeval(b,x);
     when (time = eval(time) + t) process(b,t);
}

interval b;
process(b,0.1);
```

The variable b can be used in differential equations, and every 0.1 time units, a new value for b is chosen. Safety is the primary advantage of modeling the noise explicitly inside of HCC. This approach requires no modifications to the numerical integration algorithm because b is constant over every trace phase; the interpreter must revert to point phase to change the value of b.

6.1.1 Experiment

For an experiment, I changed the RWGS model to use "choose" to simulate failures and instantiated a noise process inside each FlowController object. I anticipated a small (maybe 2x) slowdown due to the overhead of switching between point and trace phases. Unfortunately, the simulation runs at least 10 times slower now. To understand why, one must look to the algorithm for processing default logic statements. When a default logic statement is encountered, an assumption is made about its truth value and processing continues. If a contradiction is encountered later, then the algorithm backtracks and tries another truth value. Periodically choosing a new random value for a random variable requires such a default ask:

```
void storeval(interval var, /*const*/ interval val) {
     interval temp;
     temp = eval(val);
     do {
          always var=eval(temp);
     } hencewatching(var!=eval(temp));
}
```

Every instance of a random noise process adds default logic statements to the model, and the worst case complexity of default processing is exponential in the number of default statements. Adding 5 random noise processes swamps the interpreter in default processing.

6.2 Attempt 2

To avoid the switch back to point phase, and the associated default processing, the integrator must be made "aware" of stochastic terms. This is more difficult (to do right) than the first approach.

As a first attempt, I altered the definition of "once" to force re-evaluation at a fixed time intervals. Because the "once" values are sufficiently stable, there are no obvious problems with root-finding. However, I don't know enough about the integrator to say for certain that this has a well-defined semantics.

Some issues to be resolved are:

- 1 When is the special variable "time" updated relative to the others? Is there some hidden dependency on the order in which the variables are processed?
- 2 The timescale hard-coded into once is not respected by the integrator actual changes to the noise parameter occur at times determined by the step size (both full h steps and fractional h steps?).

Because of these unresolved issues and a need to begin work on the C→HCC interface I had to abandon this approach.

7 Sampling from multiple runs of HCC

Although HCC with "choose" has a probabilistic semantics, one cannot gather much useful information from a single simulation run. The probability of a particular outcome is implicit, hidden in the semantics of "choose". To obtain estimates for the probabilities of various outcomes, it is necessary to run multiple sample simulations with the HCC interpreter. To this end, I wrote a small wrapper script around the HCC interpreter. It is called "hcc_sample", and its usage is as follows:

```
% hcc_sample <number-of-samples> <output-directory> <hcc-options...>
```

Arguments after the first two are passed unmodified to "hcc". The script runs the hcc interpreter with the given options some number of times, generating <number-of-samples> data files in <output-directory>. Note: the HCC model being simulated must contain "sample(<var>,...)" statements for any meaningful output to be produced. These output files are suitable for display with a command like:

```
% hcc_plot_nfiles [options] <x-var> <y-var> <output-directory>/*
```

The "hcc_sample" script does not do any quantitative analysis of the output files. It simply spawns a number of "hcc" child processes which run in parallel. Actually, the output files are not suited for any statistical analysis because the sample points in different files are not comparable. Because of the way the function "sample" behaves, corresponding lines in each output file may not have been produced at the same time during the simulation. The script's main use is to produce nice plots so one can get a qualitative feel for the probabilistic behavior of a system. The new C→HCC API (described below) provides a better way to get probabilistic information from simulation runs.

8 External control of HCC simulations

8.1 Related Work

This can be seen as an instance of the general problem of how to integrate a sequential language with another "pure" language which does not support sequencing or side-effects. The Haskell designers solved this problem by providing a special set of sequencing operators for expressions of abstract type "IO a". Only these sequencing operators can manipulate values of type "IO a", so the programmer is forced to specify an order of evaluation for I/O or other side-effect dependent expressions. The type system prevents nesting of I/O operations inside of pure functions; thus sequential code occurs at the outermost levels of nesting, and is in a position to "control" evaluation of the rest of the program. If one were to allow the opposite, then the pure functional semantics of the language would be destroyed. In a similar way, C is in a position to control evaluation of HCC code. It has a sequential semantics (statements are executed in a certain order) whereas HCC does not. What then, are the pieces of an HCC model that the C program will manipulate? Phases are a natural abstraction that allows fine-grained control without disturbing the semantics of HCC's inference. Any C→HCC API should only allow execution of whole phases, stopping HCC in a consistent state in which variables/expressions can be evaluated, constraints added, etc.

8.2 C→HCC API description

Any interface designed to allow control of HCC from inside a C program should provide at least these 4 categories of operations:

- 1) add code or individual constraints to the store
- 2) query the store (whether a constraint is entailed)
- 3) control simulation time
- 4) evaluate variables/expressions given the current store

The current "c2hcc" API adequately covers items 1 and 3 but is somewhat lacking in the others. It would be nice to be able to stop the simulation when a condition (other than time = maxTime) occurs and evaluate arbitrary expressions. To this end, I designed a higher-level interface to HCC, layered on top of the current set of functions. The new functions are the only way to access capabilities relating to tracking multiple HCC interpreter states simultaneously. The old functions should continue to work, but they apply only to a single global HCC state.

Initialization:	
HCC_init_samples(n)	'n' instances of the HCC interpreter data structure are created and initialized. subsequent calls to HCC_tell, HCC_ask, etc... are applied to all of these states.
Adding Constraints:	
HCC_tell(code)	parses and adds code to HCC's queue, then runs one point phase of inference to propagate effects of the addition.
Querying for Constraints:	
HCC_ask(condition)	adds "when (condition) stop", then runs interpreter until condition becomes true, maxTime is reached, or an error condition occurs.
HCC_ask_now(condition)	similar to HCC_ask, but if the condition is not true at this instant, HCC does not continue running in an attempt to find the time at which it becomes true. not yet implemented.
Extracting Values:	
HCC_eval(expr)	expr is evaluated in an environment consisting of the current variable store. currently only works for expressions containing a single variable ("ht" or "time", for example).
HCC_mean(expr)	calls HCC_eval for all the HCC states being tracked, and computes the mean of the resulting intervals.
HCC_pr(condition)	calls HCC_ask_now for all the HCC states being tracked, and returns an estimate of the probability of the condition occurring. not yet implemented.

Example

```
HCC_tell("#include \"/home/jmoody/hcc/examples/randomball.hcc\";");

while(HCC_ask("ht = 0") == HccStop) {
       v = HCC_eval("time");
       printf("bounced at time %s\n", HCC_sprintfValue(v,HNTInterval));
       /* make sure it gets off the floor before we ask (ht = 0) again */
       HCC_ask("ht > 0");
}
```

To implement HCC_ask, it was necessary to add a "stop" pseudo-statement into the language. Actually stop is implemented as a new built-in variable like "time". Because it is a boolean, the fragment "stop" is read as "stop = true". At the end of each phase, the interpreter checks whether stop has become true. It is not necessary to use HCC_ask to access the functionality of stop. If desired, stop may be used to halt simulation from "inside" of a regular HCC program.

Example:

```
when (altitude > 9500) stop;
```

Note: Because of the way that checking for stop is implemented, the program fragment "if (altitude $>$ 9500) stop" will NOT have the desired effect. Statements inside an "if" expire at the end of phase, so stop is no longer true when the check is done.

9 Bibliography

References

[1] Gupta, Vineet, Radha Jagadeesan, and Prakash Panangaden. "Stochastic Processes as Concurrent Constraint Programs (an extended abstract)." presented at POPL. 1999.

[2] Hanus, Michael. "The Integration of Functions into Logic Programming: From Theory to Practice." Journal of Logic Programming. v19,20 1994, pp583-628.

[3] Ma/luszy'nski, Jan, Staffan Bonnier, et. al. "Logic Programs with External Procedures." In Apt, de Bakker, Rutten (eds.) Logic programming languages: constraints, functions and objects. The MIT Press, 1993.

[4] Thirunarayan, K., and Ewing, R. "Characterizing a portable subset of behavioral VHDL-93." In: Computer Hardware Description Languages and their Applications. Chapman and Hall, April 1997, pp97-113.