Synthesizing Concurrent Code using a Solver-aided DSL

Harlan Kringen and Zachary Sisco

1 Introduction

Systems which require multiple processes and threads tend rely on a classic suite of synchronization tools, including locks, mutexes, condition variables, monitors, and semaphores. Synchronization refers to using these primitives in a way that prevents data races, race conditions, staleness, and any other problems that might stem from distinct pieces of code trying to modify the same region of data in an unstructured way. Unfortunately, writing synchronization code is difficult and error-prone. Programmers have susbequently researched ways of automatically generating synchronization code based on a higher-level understanding of the intended behavior. This field of research is known as *program synthesis* and often done through the use of SMT sovlers or other logical specification tools. In this paper we build a concurrent, imperative programming language to study program synthesis with the aid of the Rosette program synthesis framework (Torlak and Bodik 2014). We demonstrate how to design and implement such a language in the Racket programming language and use the Rosette framework to perform a set of basic synthesis tasks.

1.1 Motivation

Ferles et al. (2018) explored synthesizing explicit signal monitors for multithreaded code in Java. Such monitors use condition variables to update locks to communicate with other threads working on shared memory. Using automated program synthesis ensures that the generated synchronization code is correct and, ideally, optimized. Based on our experiences in the CS170 Operating Systems class, which uses semaphores extensively, we were interested specifically in synthesizing semaphores for basic concurrent code. Given a simple threaded implementation of a multiple-reader -multiple-writer queue, we wanted to state a simple, high-level description of the behavior—e.g., that two writers may not both push to the queue at the same time, but multiple readers could pop, and generate code with the corresponding counting semaphores. This requires a specification language, as well as some type of semantics the Rosette framework can use to build in semaphores where it needs.

1.2 Contributions

Our concurrent, imperative programming language is able to simulate parallel executions of multithreaded code, as well as give the user a synchronization primitive similar to a mutex to lock code so as not to be interrupted by other threads. We are able to pass this to the Rosette library with bits of user code replaced by Rosette's symbolic values, and can perform basic solver-based queries on the code. These queries include making assertions about the end state of the program and having Rosette provide assignments for the symbolic values so that the assertion is satisfied, or alternatively find a counterexample to the assertion (assuming it was invalid). We were not able, however, to replace portions of our code with Rosette's "holes" and replace those holes with syntactic expressions from our input language. Nevertheless, we have a good foundation moving forward for solving this problem.

2 Methodology

In this project we followed the paradigm presented in this course's first homework assignment which demonstrated Rosette as a solver-aided programming language. To that end, we designed a DSL (domain-specific language) to simulate features of a concurrent programming language, using as a starting point the toy language Imp (Winskel 1993), which we have termed `conimp`. We present the grammar for `conimp` in Table 1. Our main contributions to the language were adding the `par` and `atomic` operators which give us the semantic notion of concurrency, and an interpreter which offers a way to simulate a non-deterministic environment. With these additions, we simulated a form of concurrency by being able to interleave commands at the will of a scheduler built into our interpreter. We will elaborate on the design of our language by using the following code snippet as an example. It is based on a midterm question used in the CS170 class.

2.1 Imp as a DSL

Rosette requires the user to provide a specification language that it can use to lift and convert into logical SMT formulas. We chose Imp because it afforded the basic features necessary to test concurrent semantics, namely integer arithmetic and logical sequencing. There are a number of examples and tutorials using custom DSLs in the lectures of James Bornholt and Emina Torlak. These however used fairly simple languages and could not ultimately provide the level of detail we needed to employ Rosette to any degree of utility with our own concurrent language. We present in Listing 1 a basic example of the features of `conimp`. In the example we create two threads that each increment two global data values, with some of those increments being locked under a so-called "mutex". The listing serves as a basis for our experimentations with the solver features afforded by Rosette as well.

```
par A0 A1 with G = { A -> 5, B -> 50 }
(define A0 (list
       (: (load 'A 'a)
       (: (:= 'a (add 'a 1))
       (: (store 'a 'A)
       (: (atomic
          (list (: (load 'A 'a)
                (: (:= 'a (add 'a 1))
                (: (store 'a 'A)
                (: (load 'B 'b)
                (: (:= 'b (add 'b 1))
                   (store 'b 'B))))))))
        (halt)))))))

(define A1 (list
       (: (atomic
          (list (: (load 'A 'a)
                (: (:= 'a (add 'a 1))
                (: (store 'a 'A)
                (: (load 'B 'b)
                (: (:= 'b (add 'b 1))
                   (store 'b 'B))))))))
       (halt))))
```

Listing 1: Two threads concurrently updating a global store (`conimp`).

2.1.1 Big-Step Semantics

We first implemented Imp using a big-step style semantics. The big-step style is exemplified by having a simple recursive function evaluate all terms in a given expression until termination (Huttel 2010). Most of the examples we found that defined a DSL to feed into Rosette were given similarly uncomplicated, big-step style semantics. This can be straightforwardly accomplished by giving the terms of the language as ground functions of integer arity and then pattern matching over them. The advantages of this approach are its simplicity and ease of setup. There are disadvantages though in how the evaluation procedure is a monolithic function call. That is, there was no way to inspect a single step of the evaluation, which is necessary to do more complicated control-flow procedures. In practice, any meaningful extension of a toy programming language should evolve beyond a big-step style, or natural, semantics.

2.1.2 Small-Step Semantics

To remedy the above shortcomings, we realized we needed to adopt a more granular approach to evaluating expressions (Fernandez 2004). The programming languages community defines an analogue to the big-step style, appropriately termed "small-step semantics." In this protocol, the recursion is modded

Arithmetic Expression	e	$::=$	$e + e \mid e - e \mid e * e \mid v \mid n$
Boolean Expression	b	$::=$	$\neg\, b \mid b \wedge b \mid e = e \mid e \leq e \mid \text{true} \mid \text{false}$
Command	c	$::=$	$v := e \mid \texttt{load}\ g\ v \mid \texttt{store}\ v\ g$
		$\mid$	$\text{if}\ b\ \text{then}\ c\ \text{else}\ c \mid \text{while}\ b\ \text{do}\ c$
		$\mid$	$c\ ;\ c \mid \texttt{atomic}\ c \mid \texttt{par}\ c \mid \texttt{skip} \mid \texttt{halt}$
Local Variables	v	$\in$	$\mathcal{V}$
Global Variables	g	$\in$	$\mathcal{G}$
Integer	n	$\in$	$\mathbb{Z}$

Table 1: Concurrent Imp DSL grammar.

out and each pattern match, instead of *destructing* a given term, *constructs* a data structure representing the state of the overall evaluation. A common notion of state used in the literature comes from the notion of an *abstract machine* (Huttel 2010).

Abstract machines are an approach to operational semantics that centralize the notion of registers, so that temporary evaluations of code, ad future directions of evaluation, are stored in registers as if they were stacks. The tradition began with Peter Landin's SECD machine and was redesigned by Matthias Fellesen under the CESK moniker, an acronym that stands for (C)ontrol, (E)nvironment, (S)tore, (K)ontinutation. We adopted the CESK approach to rebuilding our interpreter in the small-step style.

Intuitively, the C stack keeps the expressions currently being evaluated, the E and S stacks keep track of variable assignments, while the K stack maintains partially evaluated code that will be referenced later. As an example, in the implementation of the while loop function, we can think of the entire while loop expression residing on the C stack. It can be pattern matched over to split the command into two pieces, a boolean test and the set of possible commands to execute based on the test returning true or false. We may put the boolean test on the C stack and the rest on the K stack, then proceed to evaluate the boolean test alone. Based on the result of that test we may pop the K stack to the appropriate command, regarding the alternative option. It is this interplay between the C and K stack, being able to store references to code with a placeholder value waiting on a future computation, that affords more complicated flow of control.

2.1.3 Par and Atomic

After reimplementing our language as an abstract CESK machine, we had only really mimicked the functionality that we already had with the big-step semantics. However, we could implement the `par` and `atomic` functions in the above style.

Figure 1 shows the small-step semantics for the `par` operator. In our implementation, we actually combined the interpreter and the semantics for the `par` function. We designed it to randomly choose between a set of commands (implicit threads) and evaluate the command list a single step recursively. After each step, the interpreter would be back to randomly choosing a command branch. In this way we could interleave commands at a more granular level and simulate multiple branches of code running in parallel.

There must be a duality with the `par` function however, which is the user's ability to constrain parallel evaluation, an ability to tell the interpreter that a certain region of code may not be parallelized, or must be run sequentially, in an uninterruptible fashion. We adopted the operator `atomic` for this purpose. The user writes the `atomic` keyword followed by a Racket list of expressions. If the interpreter sees the `atomic` keyword it circumvents the parallel choice and continues until the list of expressions have been evaluated.

There is a subtle distinction between providing the semantics for a concurrent programming language and simulating its behavior. We thus note the difference between the pattern matching rules we provide for our syntax and the interpreter which selects expressions of code to evaluate. The interpreter implements a simple "scheduler" which in our case is a random choice among expressions. By providing an interpreter ourselves, we gain the ability to run example code sequences. Without such a function, we would have to manually construct a given "run" of the code.

$$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_1' \texttt{ par } S_2, s'\rangle}\textsc{Par}^1$$

$$\frac{\langle S_1, s\rangle \Rightarrow s'\rangle}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}\textsc{Par}^2$$

$$\frac{\langle S_2, s\rangle \Rightarrow \langle S_2', s'\rangle}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_1 \texttt{ par } S_2', s'\rangle}\textsc{Par}^3$$

$$\frac{\langle S_2, s\rangle \Rightarrow s'\rangle}{\langle S_1 \texttt{ par } S_2, s\rangle \Rightarrow \langle S_1, s'\rangle}\textsc{Par}^4$$

Figure 1: Semantics for the `par` operator in `conimp` (adapted from Huttel (2010)). For clarity, the rules evaluate over two threads—$S1$ and $S2$—but our implementation generalizes for any number of threads.

3 Evaluation

We evaluate the implementation of `conimp` on Listing 1, which is emblematic of a typical race condition. The evaluation proceeds by employing constructs in the Rosette language to facilitate solver-based queries. That is, we treat the detection of race conditions as a solver-based query, which Rosette excels at solving. This illustrates the dual ability of our interpreter to handle concurrency through the `par` operator and our language's ability to be instrumented by the Rosette framework. We conclude by reexamining the race condition example with an eye to real world implementation details.

3.1 Rosette Integration

Rosette takes expressions in the DSL annotated with Rosette constructs for symbolic values. The annotated code is then lifted into Rosette's internal language that affords functions for common tasks utilizing SMT solvers. The three most common are solving, verification, and synthesis. In the "solve" case, the user code has expressions replaced with symbolic values, and is additionally given an assertion for what the code should do or compute to. Rosette uses the backend SMT solver to find values for the symbolic variables to evaluate the expressions in a way that satisfies the assertion. It can be thought of as a simplistic form of synthesis. For the "verify" case, Rosette looks for a way to fill in the symbolic variables to find a counterexample to the user code's assertion. Finally, in the "synthesis" case, the user code is additionally given "holes" replacing whole expressions along with a specification for behavior. Rosette employs its solver to find candidate expressions in the DSL for the holes that maintain the specification. We found Rosette readily capable of solving and verifying expressions. Unfortunately, we were not able to synthesize program holes with Rosette.

3.2 Solving and Verifying

The goal in both solving and verification queries is to have the SMT solver search for values to fill in symbolic variables that satisfy a set of logical formulas. In our case, the logical formulas are our assertions about the end state of running our program, which in the running example, is just a set of two integer equations. We assert the state of two integer counters in a global store at the outset of the program, and then at the end of the program, noting that they should have been incremented a certain number of times. The example puts some increments inside of a locked `atomic` region, so there is a sort of race each thread performs in trying to access the global store. This race can result in the increments outside the atomic region being lost, and thus there are multiple possible end states for the program. We replace the sensitive components in our program with symbolic values, which come from the Rosette language, so that we can define our logical formulas in the assertion about the program's end state. The program along with its assertion is then passed to a Rosette function that employs its solver backend to answer the query. An example of the solve query is presented in Listing 2.

Our program state consists of integer variables in a global hash table, and we generate two symbolic integers. The "evaluate," "solve," and "assert" keywords additionally come from Rosette. The logical assertion about our program behavior follows the "assert" keyword. Running Listing 2 over the Listing 1 example, with the increments of 1 replaced by the symbolic x and y variables, results in a list, sure enough, of "(1 1)" indicating Rosette found a way to supply values for the variables in a way that made the assertion true.

The code in Listing 2 can be altered for the verification case. In that case we keep the same symbolic variable replacements, but change the word "solve" for "verify" and instead make the assertion something untrue. Again, running the example has Rosette produce a list with assignments for the symbolic variables that prove the assertion false. We were able in both cases to replace the integer values of the increments, as well as the initial global store values to get the desired behavior. While the examples demonstrated are somewhat trivial, they form a solid foundation for future studies in verification of concurrent programs written in `conimp`.

3.3 Synthesis

We used Rosette language features, namely symbolic values, in our own language to make use of the solving, verification, and synthesis capabilities Rosette affords. Specifically, we were able to replace basic expressions in our language with symbolic values, along with an assertion about the behavior of the program at its end state, and have Rosette compute assignments for those symbolic values to satisfy the assertion. We were also provided false assertions about the behavior of the program and were able to have Rosette determine appropriate counterexamples. We did not unfortunately use Rosette to synthesize missing pieces of our test code with the more expressive "hole" construct, as opposed to mere symbolic values. This, we speculate, is due to the complicated nature of `conimp`. It is not obvious what the nearest logic for the language should be, and we did not give Rosette any sort of logical specification about how the program was to behave. This comes for free in a lot of simpler examples where the DSL is essentially a reflection of the logic itself, be it integer linear arithmetic or quantifier-free boolean formulas.

3.4 The Example Program Revisited

The example in Listing 1 above is based on a similar example written in C for the CS170 class (shown in Listing 3). The class example is designed to show that there is a data race for the integer assignment in thread 1 that is not locked by the mutex. Technically, it is possible for thread 1 to begin the integer assignment expression, then because it accesses data outside of a mutex, it could be pre-empted, or interrupted, by thread 2, which begins modifying the data inside its own locked region. Thread 1, when it regains control of the CPU, will begin a logical line down from the initial line and so the integer assignment will be lost. This suggests at least two possible pathways for the code. In fact there are 3, with the third following the above logic but for the opposite interruption scheme.

There is a sense then in which Listing 1 is a subtly inexact rendering of Listing 3, stemming from the fact that there is no way for our interpreter to race the threads. It is only capable of interleaving states. The code in Listing 3 trades on a more complicated notion of execution where threads have a notion of ownership. A given thread owns the CPU while it is executing, and by proxy, any global data. Ownership can be preempted and in this case a thread will abandon a region of code that it was running. That is to say, there is a wholly separate semantics for thread ownership, or data ownership that we have not programmed into our language. The concept of data races such as this have been studied, for instance in the Rust programming language (Matsakis and Turon 2018) where ownership is built into the type system itself. It is not immediately clear how we could add this to our language without adopting

```
(define test-k (hash-set (hash-set (hash) 0 (list)) 1 (list)))
(define global-senv
(hash-set (hash-set (hash) 'A 5) 'B 50))

(define-symbolic x y integer?)

(define test-se
  (hash-set (hash-set (hash-set (hash) 0 (hash)) 1 (hash)) 'global global-senv))

(evaluate (list x y)
          (solve
            (assert (let ([g (hash-ref (parrun (hash-set (hash-set (hash) 0 A0) 1
                A1) test-se test-k) 'global)])
                      (and (or (= (hash-ref g 'A) 7) (= (hash-ref g 'A) 8))
                           (= (hash-ref g 'B) 52))))))
```

Listing 2: Setup and Test Environment.

types, nor how we would have our interpreter simulate the race without some sort of a clock signal. This reinforces the distinction made earlier about implementing specific program behaviors in the object language or in the interpreter. Nevertheless, both are interesting possible extensions to `conimp`.

4 Conclusion

At the conclusion of the CS292C class, we extended the toy language Imp by providing language constructs for executing commands in parallel as well as dually preventing parallel execution for specific regions. The parallelism is based on the ability to interleave single steps of execution of expressions based on non-deterministic choice. Reflecting a central tension language designers face, between incorporating functionality into the host, or meta-, language and the object language, we chose to write a minimalistic interpreter for the language which implemented the non-deterministic choice between expressions. The interpreter gave us the ability to run our code as well as instrument it with Rosette. We could not get synthesis to work with our toy language, due to our sparse knowledge of Rosette internals and lack of documentation, as well as due to a possible gap in semantic information about our language that we could even provide to Rosette. We have however identified ways forward to ultimately make synthesis for `conimp` go through.

4.1 Future Work

Going forward with this work, the main goal is to get synthesis working. To that end, there are two obstacles that must be addressed. The first is our lack of familiarity with the internals of Rosette. The error messages Rosette gives often amount to "unsat", without any instrumentation context or traceback as to what was happening internally that caused this outcome. To some extent this is a feature of Racket as well, namely that as a fairly minimal language based off the lambda calculus, a lot of the errors amount to the unhelpful observation that "application: is not a procedure." The second obstacle is the current absence of a logical specification language for our program, or a more detailed semantics about how things *should* behave. There is work by E. Allen Emerson in this field. In Emerson and Samanta (2011), E. Allen Emerson and Roopsha Samanta adopt the approach of specifying regions of code as state in a finite-state automaton where the regions are one of "non-critical", "trying", and "critical". These regions are considered states in the graph while the edges between them reflect code sequencing and are annotated by logical

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

int A;
int B;

pthread_mutex_t Lock;

void *
Thread_1(void *arg)
{
 A++;
 pthread_mutex_lock(&Lock);
 A++;
 B++;
 pthread_mutex_unlock(&Lock);

 pthread_exit(NULL);
 return(NULL);
}

void *
Thread_2(void *arg)
{
 pthread_mutex_lock(&Lock);
 A++;
 B++;
 pthread_mutex_unlock(&Lock);

 pthread_exit(NULL);
 return(NULL);
}

int main(int argc, char *argv[])
{
 pthread_t t1;
 pthread_t t2;
 int err;

 pthread_mutex_init(&Lock,NULL);
 A = 5;
 B = 50;

 err = pthread_create(&t1, NULL,
     Thread_1, NULL);
 err = pthread_create(&t2, NULL,
     Thread_2, NULL);

 pthread_join(t1, NULL);
 pthread_join(t2, NULL);

 printf("A: %d, B: %d\n",A,B);

 pthread_exit(NULL);

 return(0);
}
```

Listing 3: Two threads concurrently updating a global store (C).

formulas. The formulas specify properties such as "deadlock free". We would like to adopt this strategy as our specification language; however, it represents a substantial amount of developer time.

If we can get synthesis working, it would finally be possible to write a backend component of our project which mechanically translates synthesized code into the C language. This would further enable evaluation of generated code against hand-written code from the CS170 class as well as classic problems in concurrency. Most likely, this would be based on translating synthesized structures into `pthreads` implementations for locks and mutexes, and the class's `kthreads` library for semaphores. There is further work to be addressed in synthesizing non-preemptive versus preemptive threading code. We think there is utility in pursuing the project to this point, both as an educational tool for the operating systems class, as well as for operationalizing the Emerson and Samanta semantics, to determine if this is a viable method for synthesizing concurrent code for non-trivial software tasks.

References

E. Emerson and Roopsha Samanta. An algorithmic framework for synthesis of concurrent programs. pages 522–530, 10 2011. doi: 10.1007/978-3-642-24372-1_41.

Kostas Ferles, Jacob Van Geffen, Isil Dillig, and Yannis Smaragdakis. Symbolic reasoning for automatic signal placement. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 120–134, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192395. URL `http://doi.acm.org/10.1145/3192366.3192395`.

Maribel Fernandez. *Programming Languages and Operational Semantics*. King's College Publications, London, England, 2004.

Hans Huttel. *Transitions and Trees*. Cambridge University Press, New York, New York, 2010.

Nicholas Matsakis and Aaron Turon. The rust programming language, 2018. Accessed: 2019-12-10.

Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594340. URL `http://doi.acm.org/10.1145/2594291.2594340`.

Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1993.