Shellac: a compiler synthesizer for concurrent programs[⋆]

Christopher K. Chen[1] [0000−0002−5608−7550] cchen@nougat.org, Margo I. Seltzer[1] [0000−0002−2165−4658] mseltzer@cs.ubc.ca, and Mark R. Greenstreet[1] [0000−0002−1864−9495] mrg@cs.ubc.ca

The University of British Columbia, Vancouver, BC V6T 1Z4, Canada

Abstract. Formal specification languages such as TLA+ and UNITY are used to design and verify concurrent programs. These languages are intended for analysis rather than for execution. A compiler or a human must implement the specified program in a lower-level executable language. We present Shellac, a synthesized compiler from UNITY to Arduino C++ and Verilog. The approach is essentially syntax-directed translation, where the translation rules are automatically generated via program synthesis. This approach produces a correct-by-construction compiler without burdening the compiler writer with manual specification and verification. We evaluate Shellac by compiling Paxos consensus in UNITY to implementations in Arduino C++ for microcontrollers and Verilog for reconfigurable hardware.

Keywords: Automatic programming · Compilation · Concurrency · Formal models · Program synthesis

1 Introduction

Concurrent programs are notoriously difficult to write, debug, and verify. This is especially the case with imperative programs that mix state mutation and control flow, where the resulting state explosion makes formal analysis intractable.

Formal specification languages and program logics, e.g., TLA+ and UNITY, enable the design and verification of concurrent programs at an abstract level [7, 10]. A specification written in such a language describes a state machine by formally defining valid initial states and permitted state transitions. Such specifications are *behavioural* as opposed to *logical*, e.g., a temporal logic formula.

A behavioural specification by itself is useful for analysis and as a design tool, but in general, it is too abstract for direct execution. Often, a programmer manually translates a specification to a lower-level implementation language. This process is error-prone and verifying correctness, e.g., showing refinement, requires substantial proof effort, e.g., seL4 [5]. Alternatively, a programmer could

[⋆] We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Work by Christopher K. Chen was supported by NSERC and The University of British Columbia.

write a compiler from a specification to an implementation language, e.g., CompCert, but still, the engineering and verification effort remains incredibly high [8]. We are interested in *automatic* and *verified* techniques for generating programs that satisfy their specification. In particular, we exploit *program synthesis* to reduce the proof burden of verifying an implementation against its behavioural specification.

Program synthesis is any procedure that generates a program that satisfies some constraint. If the constraint is a correctness condition with regard to a specification, then the synthesized implementation is correct by construction. This means that, assuming the correctness of the synthesis procedure, no further proof is required. We focus on counterexample-guided inductive program synthesis (CEGIS), where a search-verify-refine loop uses verification counterexamples to prune the search space [15]. Any decidable search-based synthesis procedure places bounds, e.g., expression depth, over the infinite space of programs. This search space explosion, exponential in the case of a bound on expression depth, makes it difficult for CEGIS to find programs that satisfy nontrivial specifications.

We present a method to construct a compiler by generating verified translation rules via program synthesis. Our synthesized compiler accepts UNITY specifications and generates implementations in Arduino C++ and Verilog [1,4]. Instead of synthesizing concrete C++ or Verilog *programs* from a complete specification, we synthesize implementations of *elements* of UNITY's expression syntax. Each of these synthesized implementations takes the form of a rewrite rule from source to target. These synthesized rewrite rules are assembled into a recursive syntax-guided compiler pass. We show that for channel-based UNITY programs, syntax-guided translation preserves the specification's safety and liveness properties. Shellac, our compiler synthesizer, is written in Rosette [16]. The programs that Shellac synthesizes have all the benefits of traditional compilation: they are deterministic and handle arbitrary source programs that satisfy a channel-based schema. We provide:

1. A rewrite rule synthesizer that includes language embeddings in Rosette and a procedure for generating rewrite rules between languages.
2. A proof that the compiler preserves safety and liveness properties for channel-based UNITY programs.
3. An evaluation of rewrite rule synthesis performance and the compilation of Paxos consensus from UNITY to Arduino C++ and Verilog.

2 Preliminaries

Shellac starts with a partial compiler, or a sketch, whose organization is illustrated in Figure 1. We begin with an discussion of our state and process model. Next, we describe an abstract channel model for asynchronous communication and a dataflow merge specification used as a running example. Given that context, we introduce the UNITY specification, boolean-bitvector parallel, boolean-bitvector scalar, boolean-bitvector sequential, Arduino C++, and Verilog languages.

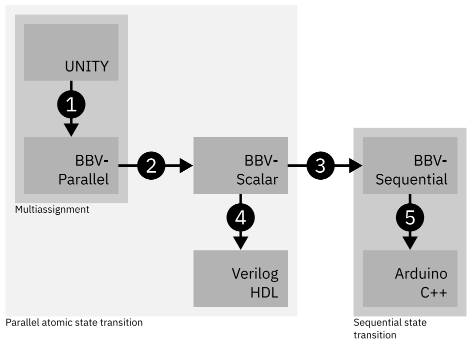

Fig. 1. Source, intermediate languages, and target languages with compiler passes. Languages with important semantic similarities are contained in boxes.

2.1 State, assignment, and processes

The source and target languages modelled here are all imperative: we effect computation by mutating state. Each program is defined over a set of variables, and a state is a mapping from those variables to values. Programs operate by inspecting the current state of their variables and, if permitted, assigning new values to a subset of them, effecting a state transition. Variables are either *internal* or *shared*. Shared variables are reserved for channel communications, described below. The languages presented differ in their assignment/state transition semantics. In particular, UNITY, boolean-bitvector parallel/scalar, and Verilog state transitions are parallel and atomic, while boolean-bitvector sequential and Arduino C++ state transitions occur sequentially with each assignment statement.

Or model of concurrency is based upon communicating processes, with varying degrees of parallelism. Each such process, P, must be coherent: if P writes to some variable, v, no other process writes to v; if process P reads v and some other process, Ch, writes to v, then Ch must be a channel as described below. Processes execute one assignment at a time. Concurrency is a property of a system of processes, not of a process itself.

2.2 Channels

Interprocess communication is via point-to-point channels. At the specification level, a channel, c is an abstract data type with the following operations:

– *empty?(c)* $\rightarrow$ *boolean*
– *fill!(c, v)* $\rightarrow$ *channel*, precondition *empty?(c)*
– *full?(c)* $\rightarrow$ *boolean*
– *read(c)* $\rightarrow$ *value*, precondition *full?(c)*
– *drain!(c)* $\rightarrow$ *channel*, precondition *full?(c)*

The channel ensures that *ch := fill!(c,v)* leads to a state where $\neg empty?(c)$ holds and from which $full?(c) \wedge (read?(c) = v)$ will eventually hold. Likewise, *ch :=*

drain!(c,v) leads to a state where $\neg$*full*?(c) hold and from which *empty*?(c) will eventually hold. The *empty?* query and *fill!* operation are only valid for a process designated as the sender. Likewise, *full?*, *read*, and *drain!* are only valid for the receiver.

Figure 2 illustrates the channel protocol.

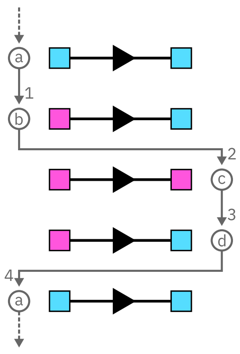

Fig. 2. One round of the channel protocol. Sender to the left, receiver to the right. Cyan boxes represent the empty state, magenta boxes the full state. Circles are states, arrows are transitions.

2.3 Dataflow merge element

We use a merge element from dataflow programming as a running example. The dataflow merge element is a forwarding mechanism on the receiving end of two channels: *inA* and *inB* and the sending end of one channel: *out*. Two assignments are defined:

1. If *inA* is full and *out* is empty, fill *out* with the value of *inA* and drain *inA*
2. If *inB* is full and *out* is empty, fill *out* with the value of *inB* and drain *inB*

In either assignment, the state of the other input channel is immaterial, not part of the condition nor the object of the assignment. In the case where both input channels are full and *out* is empty, the specified behaviour is nondeterministic, and either assignment can occur.

2.4 UNITY

UNITY is a language for specifying parallel and distributed programs [10]. We provide an informal overview of assignment syntax and semantics here, and provide a formalization of expressions and state transitions later, but we do not cover Chandy and Misra's UNITY program logic. A specification defines variables, initial state, and next-state assignments. A UNITY program may reach a fixed point, but it does not halt in the traditional sense. We discuss datatypes, followed by parallel assignment, then nondeterministic choice. Our examples include dataflow merge and channel processes.

Datatypes Our model of UNITY includes primitive datatypes, send/receive buffers, and channels. Send/receive buffers are fixed length lists with a cursor for serializing data to be sent over a channel. Presently, Shellac supports booleans and natural numbers for primitive datatypes, send/receive buffers that are lists of booleans, and channels for boolean data.

Simultaneous assignment A simultaneous assignment statement has a list of variables on the left side and an expression list on the right side. The expression list is either *simple* or *conditional.* A simple list contains expressions corresponding to the assignment's variables: e.g., $a, b := 42, a + 3$ assigns the value 42 to a and the sum of 3 and the *original* value of a to b. Parallel assignments are *independent* and *simultaneous*: the expressions are evaluated to values *before* any assignments are made, and all assignments occur in one atomic action. A conditional list pairs simple expression lists with boolean guard expressions: e.g., for the dataflow merge assignment from *inA* to *out*:

$$\mathit{inA.drain}, \mathit{out.fill} := \begin{cases} \mathit{drain!}(\mathit{inA}), \mathit{fill!}(\mathit{out}, \mathit{read}(\mathit{inA})) \\ \quad \text{if } \mathit{full?}(\mathit{inA}) \wedge \mathit{empty?}(\mathit{out}) \end{cases}$$

This example introduces channel predicates $\mathit{full?}(c)$ and $\mathit{empty?}(c)$ and functions $\mathit{fill!}(c, \mathit{data} : \mathit{boolean})$, $\mathit{drain!}(c)$, and $\mathit{read}(c)$.

In the above case, there is only one guarded assignment, but in general, a conditional list may contain an arbitrary number of expression list and guard pairs. Guards are not required to be exhaustive. If no guard is true the state is left unchanged. If two guards are true, their corresponding expression lists must evaluate to the same values. Parallel assignment is deterministic.

Nondeterministic choice Note that our description of dataflow merge does not specify what to do if both input channels were full, when both assignments are permitted. We express this nondeterminism in UNITY by composing simultaneous assignments with the *box* operator $\Box$:

$$\begin{array}{lcl} \mathit{inA.drain}, \mathit{out.fill} & := & \begin{cases} \mathit{drain!}(\mathit{inA}), \mathit{fill!}(\mathit{out}, \mathit{read}(\mathit{inA})) \\ \quad \text{if } \mathit{full?}(\mathit{inA}) \wedge \mathit{empty?}(\mathit{out}) \end{cases} \\ \Box & & \\ \mathit{inB.drain}, \mathit{out.fill} & := & \begin{cases} \mathit{drain!}(\mathit{inB}), \mathit{fill!}(\mathit{out}, \mathit{read}(\mathit{inB})) \\ \quad \text{if } \mathit{full?}(\mathit{inB}) \wedge \mathit{empty?}(\mathit{out}) \end{cases} \end{array}$$

Nondeterministic choice is subject to a *fair selection* or *absolute fairness* constraint: every assignment is executed infinitely often. Note that this property implies weak but not strong fairness.

Channel process As described previously, channels are asynchronous processes interfaced via channel shared variables. Channel processes propagate fill and drain actions from sender and receiver, as transitions 2 and 4 in Figure 2. Because fill and drain actions propagate asynchronously, parallel channel actions by one process may appear to an observer in any sequence or simultaneously.

2.5 BBV parallel

The boolean-bitvector parallel intermediate language is a lowering of UNITY to the booleans and fixed-length bitvectors. UNITY values are encoded as tuples of booleans or bitvectors. Booleans are encoded as $\langle boolean \rangle$ and bounded naturals are encoded as $\langle bitvector \rangle$. Send/receive buffers are encoded as the pair $\langle cursor : bitvector, data : bitvector \rangle$. Channel values are already encoded in UNITY as triples $\langle req : boolean, ack : boolean, data : boolean \rangle$, resembling signals on a serial communication line. A BBV channel is empty if $req = ack$, full if $req \neq ack$. Senders cannot modify ack, and receivers cannot modify req or $data$.

Assignment semantics are identical to UNITY, where individual assignments are deterministic, parallel, and atomic. Nondeterministic choice over assignments is also subject to a fair selection constraint. We describe the behaviour of dataflow merge in BBV parallel:

$$inA.ack, out.req, out.data \quad := \begin{cases} \langle inA.req, \neg out.req, inA.data \rangle \\ \quad \text{if } inA.req \neq inA.ack \wedge \\ \quad\quad out.req = out.ack \end{cases}$$

$$\Box$$

$$inB.ack, out.req, out.data \quad := \begin{cases} \langle inB.req, \neg out.req, inB.data \rangle \\ \quad \text{if } inB.req \neq inB.ack \wedge \\ \quad\quad out.req = out.ack \end{cases}$$

BBV scalar The boolean-bitvector scalar intermediate language is a scalar version of BBV parallel, with tuple values split into scalars. Assignment semantics are unchanged.

2.6 BBV sequential

The boolean-bitvector sequential intermediate language is a sequential version of BBV scalar. Parallel atomic assignments are replaced by sequences of scalar assignments, with fair selection over sequences. We describe the behaviour of dataflow merge in BBV sequential:

$$\text{if } inA.req \neq inA.ack \wedge out.req = out.ack$$
$$\begin{cases} out.data & := inA.data; \\ out.req & := \neg out.req; \\ inA.ack & := inA.req; \end{cases}$$
$$\Box$$
$$\text{if } inB.req \neq inB.ack \wedge out.req = out.ack$$
$$\begin{cases} out.data & := inB.data; \\ out.req & := \neg out.req; \\ inB.ack & := inB.req; \end{cases}$$

A BBV sequential program executes variable assignments one at a time and in order. This exposes intermediate states not accounted for in the specification, opening the possibility to a violation of a specification property. We describe in Section 3.4 ordering constraints that rewrite rules must encode such that sequential implementations preserve the behaviours of the original specification.

2.7 Arduino C++

To demonstrate Shellac, we use Arduino ARM microcontroller prototyping boards with FPGAs [1]. This allows us to demonstrate compilation to both hardware and software. With several Arduino boards and some wire, we demonstrate a distributed system on a desk. The Arduino language is an API in C++ derived from the Wiring [3] language. The API provides functions for reading and writing from hardware input/output pins. A toolchain by the Arduino project compiles C++ into ARM machine code and uploads the program into prototyping board flash memory.

Arduino programs are centred around an infinite *event loop* with conditional statements. Shellac generates a compiler with a backend from BBV sequential to Arduino C++.

2.8 Verilog

The Arduino prototyping boards also include Intel Cyclone FPGAs. Shellac generates a compiler with a backend from BBV scalar to synthesizable Verilog [4]. An Intel toolchain compiles Verilog into a *bitstream* that configures the FPGA into a custom digital device.

Verilog programs emitted by our compiler are centred around an *always* block containing *nonblocking* assignments. The always block is triggered by external events, e.g., clock transitions. All nonblocking assignments inside an always block defer variable updates until the end of the block, when the values update with the next clock transition.

3 Formalization and mechanization

The UNITY specification language is structured around an explicit separation between functional expression evaluation and atomic state transition. This separation allows us to formulate correctness by focusing on each concern independently. Two compilation passes engage in a semantic transformation that requires a formalization of correctness:

1. UNITY assignment to BBV simultaneous assignment, where we wish to show that specification expression semantics are preserved
2. BBV simultaneous to sequential assignment, we wish to show that target state transitions are a refinement of the specification

We describe these formal relations, their mechanization in Rosette, and how Shellac generates verification conditions for synthesizing rewrite rules.

3.1 The implements relation between expressions

Our language embeddings give a functional interpretation of expression semantics. The expressions of a language L are defined over:

- A set of values $\mathit{vals}(L)$
- A set of variables $\mathit{vars}(L)$
- A set of operators $\mathit{ops}(L)$
- An inductively defined set of expressions $\mathit{exprs}(L)$
  - If $t \in \mathit{vals}(L)$, then $t \in \mathit{exprs}(L)$
  - If $t \in \mathit{vars}(L)$, then $t \in \mathit{exprs}(L)$
  - If n-ary $\mathit{op} \in \mathit{ops}(L)$ and $a_0, \dots a_n \in \mathit{exprs}(L)$, then $\mathit{op}(a_0, \dots a_n) \in \mathit{exprs}(L)$
- A set of states, $\mathit{states}(L)$, each a mapping function $\mathit{vars}(L) \to \mathit{vals}(L)$
- A partial function $\mathit{eval}_L : \mathit{exprs}(L) \times \mathit{states}(L) \to \mathit{vals}(L)$

We describe relations between a source language S and a target language T. Because S and T may be at different levels of abstraction, we describe a scheme where values in $\mathit{vals}(S)$ can be encoded using n-tuples of values in $\mathit{vals}(T)^n$. We define a value mapping function valmap from T value tuples to S values:

$$\mathit{valtuples}(T) = \bigcup_{n=1}^{\infty} \mathit{vals}(T)^n$$

$$\mathit{valmap} \subset \mathit{valtuples}(T) \times \mathit{vals}(S)$$

Similarly, we define a variable mapping function varmap from S variables to T variable tuples:

$$\mathit{varmap} \subset \mathit{vars}(S) \times \mathit{vartuples}(T)$$

In the following, we will treat valmap as a partial function from $\mathit{valtuples}(T)$ to $\mathit{vals}(S)$ and varmap as a function from $\mathit{vars}(S)$ to $\mathit{vartuples}(T)$ when convenient.

With inter-language value and variable mappings, we can describe what it means to relate states across languages. We say that st_t in $states(T)$ encodes st_s in $states(S)$, or $st_t \precsim st_s$ iff there exist value and variable mappings such that information in st_s is recoverable from st_t after variable and value mapping:

$$\begin{aligned}
&\forall st_t \in states(T), st_s \in states(S).\\
&st_t \precsim st_s \iff\\
&\quad \exists valmap, varmap.\\
&\quad \forall \langle var_s, val_s \rangle \in st_s.\\
&\quad val_s = valmap(st_t(varmap(var_s)_0), \ldots)
\end{aligned}$$

Now we are ready to describe an implementation relation between source expressions and target expression tuples. We define a set of expression tuples over the expressions of T:

$$exprtuples(T) = \bigcup_{n=1}^{\infty} exprs(T)^n$$

We say that $exprt_t$ in $exprtuples(T)$ implements $expr_s$ in $exprs(S)$, or $exprt_t \lhd_{expr}$ $expr_s$ iff there exists a value mapping such that for all target states st_t and source states st_s, if $st_t \precsim st_s$, value mapping of the evaluation of $exprt_t$ in st_t is equal to evaluation of $expr_s$ in st_s:

$$\begin{aligned}
&\forall exprt_t \in exprtuples(T), expr_s \in exprs(S).\\
&exprt_t \lhd_{expr} expr_s \iff\\
&\quad \exists valmap.\\
&\quad \forall st_t \in states(T), st_s \in states(S).\\
&\quad st_t \precsim st_s \implies\\
&\qquad valmap(eval_T(exprt_{t0}, st_t), \ldots) = eval_S(expr_s, st_s)
\end{aligned}$$

3.2 Generating the verification condition for rule synthesis

Shellac generates syntax-guided recursive functions that translate elements of the source syntax, i.e., variables, literal values, and operators. Shellac provides a deep embedding of UNITY and uses Rosette's boolean and bitvector functions for a shallow embedding of BBV. The user provides:

1. *typemap* a mapping from UNITY types to tuples of BBV types
2. *valmap*, a mapping from tuples of BBV values to UNITY values
3. $eval_U$, an evaluation function for UNITY expressions
4. $precond_U$, a mapping from UNITY operators to precondition predicates

Variable translation is via a generated lookup table *varmap*, from declared variables of the UNITY specification and the corresponding fresh BBV variables according to *typemap*. Literal translation is via runtime synthesis, using Rosette to generate an SMT query to invert *valmap* for the given UNITY literal.

We show how a rewrite rule is synthesized for the channel fill operator: $channel \times boolean \rightarrow channel$. The corresponding BBV expression types are determined by *typemap*:

$$\begin{aligned} typemap(channel_{out}) &= \langle boolean, boolean, boolean \rangle \\ typemap(boolean) &= \langle boolean \rangle \end{aligned}$$

Therefore the corresponding BBV expression tuple will be of type:

$$\begin{aligned} &\langle boolean, boolean, boolean \rangle \times \langle boolean \rangle \rightarrow \\ &\quad \langle boolean, boolean, boolean \rangle \end{aligned}$$

From here we can see the shape of the rewrite rule: from four BBV expressions to three BBV expressions. Shellac uses Rosette's symbolic execution features to evaluate the fill operator for any proper input. It begins by allocating symbolic booleans to elements of the domain: $c_{req}, c_{ack}, c_{data}$ for the channel and v for the value written to the channel. Given these symbolic constants, it constructs a three-tuple of symbolic booleans representing as-yet unknown expressions over the domain constants, represented by $\square_i$:

$$\begin{aligned} \langle \square_{req} &: \langle c_{req}, c_{ack}, c_{data} \rangle \times \langle v \rangle \rightarrow boolean \\ \square_{ack} &: \langle c_{req}, c_{ack}, c_{data} \rangle \times \langle v \rangle \rightarrow boolean \\ \square_{data} &: \langle c_{req}, c_{ack}, c_{data} \rangle \times \langle v \rangle \rightarrow boolean \rangle \end{aligned}$$

Elements of the UNITY domain for fill are derived by applying *valmap* to the symbolic constants:

$$fill!(valmap(c_{req}, c_{ack}, c_{data}), valmap(v))$$

The fill operator is only defined over empty channels. Applying the channel empty predicate to the mapped symbolic domain elements yields the symbolic precondition P:

$$P = empty?(valmap(c_{req}, c_{ack}, c_{data})) : boolean$$

Applying channel fill to the mapped symbolic domain elements yields the symbolic postcondition Q:

$$Q = fill!(valmap(c_{req}, c_{ack}, c_{data}), valmap(v))$$

Additionally, a BBV invariant I restricts synthesis from choosing a different acknowledge value:

$$I = \square_{ack} = c_{ack}$$

The verification condition that synthesis must satisfy is the following, predicated on the precondition P, where the value mapping of the as-yet unknown expressions equals the postcondition Q and satisfies the invariant I:

$$P \implies I \wedge valmap(\square_{req}, \square_{ack}, \square_{data}) = Q$$

3.3 Inversion over target expressions for rule synthesis

Shellac engages in a bounded search over target expressions to satisfy a verification condition. The search space is generated using Rosette choose expressions, e.g., to choose between the numbers 1, 2, or 3:

```
> (choose* 1 2 3)
(ite x?$1 1 (ite x?$2 2 3))
```

The result of the choose expression is a *symbolic union* whose value, 1, 2, or 3 is predicated on the true values of fresh boolean constants `x?$1` and `x?$2`. We use choose expressions to build symbolic syntax trees that represent all expressions allowed by a language up to some depth. To find satisfying expressions for channel fill, Shellac starts by generating symbolic syntax trees at zero-depth for $\square_{req}, \square_{ack}, \square_{data}$, pushes the verification condition onto Rosette's verification stack, then asks Rosette to start CEGIS. If CEGIS succeeds, Rosette has found a model that satisfies the verification condition, i.e., a valid rewrite rule. If CEGIS returns unsat, the search depth is increased until a model is found or a user-defined depth limit is exceeded.

3.4 Ordering to satisfy refinement

In the case of channel fill, the order of assignments on a sequential execution matters a great deal. The final phase of rewrite rule synthesis finds an ordering of assignments such that the externally visible states induced by the assignments satisfies refinement, i.e., intermediate states map to P or Q, and the transition is monotonic. The notion of refinement used here is of Lynch and Vaandrager from their work on simulation relations between automata [9].

The final rewrite rule for the fill operation is encoded in a translation-rule form:

```
(translation-rule
     ;; Precondition (channel empty)
     (<=> req ack)
     ;; Domain
     (list (list req ack data) (list value))
     ;; Codomain (synthesized expressions)
     (list (! req) ack value) ;; Codomain
     ;; Ordering constraints (indices into codomain)
     (list (ordering 2 0) (ordering 1 0)))
```

A translation-rule form containts symbolic constants, e.g., `req`, `ack`, `data`, and `value`. The ordering constraints specify "come before or synchronously with" relationships between indices in the tuple of synthesized expressions for sequential assignment. For example, `(ordering 2 0)` specifies that the `value` assignment must come before or synchronously with the `(! req)` assignment.

3.5 Correctness of the synthesized programs

The asynchronously composed simultaneous assignments are partitioned by the user into *processes*. Processes can be channels or compute-processes. Channels provide point-to-point communication between two (not-necessarily distinct) processes. A property is stable in a process if no actions by *other* processes can falsify it. The channel protocol ensures that a channel being full is a stable property of the process that reads the channel; furthermore, the value of such a channel is stable when the channel is full. Likewise, a channel being empty is a stable property of the process that writes the channel.

The rules for translating channels to BBV (sequential or parallel) are provided by the Shellac developer – they are effectively an API. These rewrite rules ensure that a receive channel is only read or drained from states in which a channel is full, and likewise for send channels. When synthesizing sequential code, all read operations on a channel in a simultaneous assignment must be performed before any drain operation; likewise, the data value of a send channel must be updated before the status is set to full. Finally, guards on receive channels must be monotonic in the channel being full, and guards on send channels must be monotonic in the channel being empty. We can now sketch the correctness and liveness properties for programs synthesized by Shellac.

Correctness of synthesized sequential implementations For each simultaneous assignment of the compute process, Shellac synthesizes code that evaluates the guard(s), then evaluates the right-hand side(s), and finally updates the left-hand side(s) of the assignment. When execution reaches a point where the guard(s) have been shown to be satisfied, the abstraction function can map the implementation state, and all subsequent states until the code block is finished, to the state corresponding to a completed simultaneous assignment. The careful reader might note that a single simultaneous assignment could fill and/or drain several channels, that the sequential implementation will perform these operations in some order, and this could enable external processes to fill or drain channels written or read by this process before the sequential implementation of the simultaneous assignment is complete. This is indeed the case. Such operations are non-interfering due to the stable properties noted above, and thus they do not affect the outcome of the simultaneous assignment. The "explanation" in the abstraction of implementation state to specification state is that the simultaneous assignment completed (as soon as it was started), and these operations by other processes happened later.

Correctness of parallel implementations Each simultaneous assignment is performed on a single clock "tick" and the state update matches the specification.

Liveness UNITY requires fair selection but does not provide stronger fairness guarantees. Both the sequential and parallel implementations produced by Shellac perform round-robin execution of the simultaneous assignments in each process. This ensures fair selection.

4 Evaluation

We evaluate Shellac by synthesizing rewrite rules for UNITY operators. We then demonstrate the efficacy of the generated compiler with a single-proposer version of the Paxos consensus algorithm [6]. We target the Arduino MKR Vidor 4000 development board, which contains an ARM Cortex-M0 microprocessor and an Intel FPGA [2]. The processor provides a platform for compiled Arduino C++, and the FPGA provides a platform for compiled Verilog. Each development board has 22 I/O pins, which limits the physical size of the system we can implement, because each one-way channel uses three pins.

4.1 Experimental setup

We synthesized code on an Intel Core i7-6660U 2.4 GHz CPU with 16 GiB of memory. Shellac ran on Racket 8.3 with Rosette 4.1. We used the version of the Z3 SMT solver bundled with the Rosette distribution.

4.2 Rewrite rule synthesis

We study the synthesis time for various UNITY operations, presented in Table 1. Operators are categorized by their general types: boolean, natural numbers, channels, and arbitrary-length boolean list buffers. Rewrite rules for boolean, natural number, and channel operators synthesize quickly; this is due to the similarities in abstraction level between source and target. Buffer operations deserve further study. Because arbitrary-list-of-booleans buffers are encoded as bitvectors in BBV, buffer operations turn into bitwise operations. Setting and reading arbitrary bits in a bitvector requires a composition of bitwise functions. In the case of `recv-buf-put`, the synthesized BBV expression was 4 deep. In addition to the exponential growth in the search space as expression depth increases, it is known that CEGIS is less efficient at finding useful counterexamples for bitvector program synthesis [13].

4.3 Paxos consensus

We implemented Lamport's *single-decree synod* consensus algorithm [6] in UNITY. Paxos solves the problem of achieving *distributed consensus*: getting a collection of distributed processes to agree on a value. The processes execute in a *shared nothing* environment, which means that they interact with each other only through message passing.

The basic Paxos protocol defines three classes of participants: proposers, acceptors, and learners. Proposers and acceptors are active participants and learners are passive. Proposers initiate a protocol round by sending *prepare* messages to a majority of the acceptors. The acceptors reply with *promise* messages, promising to accept a proposed value. Once a proposer receives promise replies from the majority of the acceptors, it sends *accept* messages to acceptors to commit a value. Acceptors reply to the accept message with an *accepted* message,

Category	Operator	Rosette	Rosette + SMT
Boolean	`and`	47	125
	`or`	49	123
	`not`	27	74
	`<=>`	39	117
Natural	`=?`	196	724
	`<?`	61	207
	`+`	28	131
Channel	`empty?`	31	91
	`full?`	43	118
	`drain`	21	80
	`fill`	98	199
	`read`	10	37
Buffer	`empty-recv-buf`	23	51
	`empty-send-buf`	24	52
	`nat->send-buf`	55	104
	`recv-buf->nat`	25	145
	`recv-buf-full?`	29	92
	`recv-buf-put`	1244	527877
	`send-buf-empty?`	23	87
	`send-buf-get`	690	92228
	`send-buf-next`	80	213

Table 1. UNITY to BBV rewrite rule synthesis time in milliseconds

indicating that the value is committed and the round is complete. After a value is accepted by an acceptor, additional *accepted* messages are sent from acceptors to learners: this propagates the consensus value.

The safety guarantee of the Paxos algorithm ensures that once a value has been chosen, that value will remain stable. The algorithm guarantees this by associating each protocol round with a *ballot number*. Proposers's *prepare* messages are required to have a ballot number greater than that of any existing prepare request. Acceptors are required to inform proposers in promise messages if they have already accepted a value and the associated ballot number. When an acceptor sends a *promise* reply, it promises to ignore any requests with lesser ballot numbers. Proposers are required to propose the previously accepted value with the greatest ballot number. This ensures that once a value is accepted, it remains so.

4.4 Specification of Paxos

Specifications for the proposer and acceptor in UNITY use a pair of channels between each proposer and acceptor. Each pair of channels require six I/O pins. With a 22 pin budget, this limits specifications to three channel pairs using 18

Role	BBV parallel	BBV sequential
Proposer	2940	22
Acceptor	1039	6

Table 2. Paxos compilation time for BBV parallel and sequential passes in milliseconds

pins. The specification defines a topology with one proposer and three acceptors. The acceptor and proposer specifications contain 14 and 34 clauses respectively.

Topologies containing up to three proposers and three acceptors are possible. A 3×3 topology requires a modified acceptor specification to include the additional proposers. No changes to the proposer specification are required, because proposers communicate only with acceptors.

Compilation of proposer and acceptor Compilation times for proposer and acceptor specifications are shown in Table 2. Translating from UNITY to BBV parallel takes a few seconds mostly due to SMT verification that guards discharge any operator preconditions generated during the pass. In comparison, translating from BBV parallel to sequential only requires solving for ordering constraints and completes very quickly.

5 Related work

We are not aware of other work in the synthesis of rewrite rules for compiling concurrent specifications. However, we consider related work in inductive program synthesis, compiler synthesis, and asynchronous circuit design.

The space of expressions is encoded as a symbolic syntax tree structure, where the choice of possible children for a node is taken from the grammar of the language. Many of the expressions we are interested in synthesizing involve bitwise manipulations and comparisons. This is the case when encoding buffers at bitvectors. Solar-Lezama et al. provide the first example of *sketch-based* program synthesis, referring to their technique as compilation by constraint-solving [14]. This work also provided the first example of the insight behind the CEGIS technique. Sketch-based programming requires the user to provide a *partial program* with *holes* that the program synthesizer fills to satisfy a constraint.

There have been previous efforts in exploiting program synthesis to guarantee compiler correctness. Van Geffen et al. use sketch-based program synthesis to build a just-in-time compiler from the eBPF virtual instruction set to RISC-V [18]. The search space of assembly routines for an instruction set like RISC-V is huge, so they partition the search space an ordered set of *compiler metasketches*. Our work differs in the relative abstraction difference between source and target languages. Both eBPF and RISC-V are load-store register machines, while our focus on UNITY is to enable the compilation of concurrent or distributed programs.

Our focus on channel-based UNITY specifications was inspired by self-timed digital circuit design. Udding described three classes of circuit specifications invariant to signal delay: for synchronization, data communication, and arbitration [17]. Our channel model is defined to satisfy the properties of Udding's *arbitration* class of specifications. Our notion of channel state and a specification of a channel as a participant in data propagation is descended from Roncken's link-and-joint model, where channels are equivalent to links [11, 12]. Roncken gives us a model to bifurcate our specifications between processes with parallel atomic assignment and concurrent communications.

6 Future work

Shellac has shown that given a specific domain-specific language, a compiler with a relatively high-level of assurance can be built using program synthesis. Extending the language or developing other DSLs for other systems-level programming problems would extend this level of assurance to those areas.

The intermediate languages are a shallow embedding of the boolean and bitvector libraries of the Rosette language. Industrial intermediate representations such as LLVM or MLIR, or alternate virtual machines like Webassembly are possible targets for compiler synthesis.

The specifications that the current syntax-guided compilers generated by Shellac preserve liveness by hewing closely to the assignments specified in the UNITY program and by following a round-robin scheduling. This is admittedly conservative, and static or dynamic analysis should allow us to find more efficient schedulings.

7 Conclusion

Concurrent, imperative programs that mix state mutation and control flow admit a state explosion that makes debugging notoriously difficult and formal analysis intractable. The advent of formal specification languages allow for a concurrent program or system of concurrent programs to be described as a state machine. Such a specification enables automated reasoning. Unfortunately, formal specifications are usually too abstract for direct execution. Instead of manually translating a specification to a low-level implementation, which can introduce errors, we describe a method for exploiting program synthesis to generate compiler rewrite rules. We show that such a compiler can process UNITY specifications with channel-based communication and output both hardware and software implementations that preserve safety and liveness properties. Source code is available at https://github.com/chchen/shellac-can.

References

1. Arduino: Arduino language reference (2020), https://www.arduino.cc/reference/en/, accessed: 2020-09-02

2. Arduino: Arduino MKR vidor 4000 (2020), https://store.arduino.cc/usa/mkr-vidor-4000, accessed: 2021-01-06
3. Barragán, H.: Wiring: Prototyping Physical Interaction Design. Master's thesis, Interaction Design Institute Ivrea (2004), http://people.interactionivrea.org/h.barragan/thesis/thesis_low_res.pdf
4. IEEE Standards Association, et al.: IEEE standard for verilog hardware description language. Design Automation Standards Committee, IEEE Std 1364TM-2005 **2** (2005)
5. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: SeL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 207–220. SOSP '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1629575.1629596, https://doi.org/10.1145/1629575.1629596
6. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (May 1998). https://doi.org/10.1145/279227.279229, https://doi.org/10.1145/279227.279229
7. Lamport, L.: Specifying concurrent systems with TLA+. Calculational System Design pp. 183–247 (April 1999), https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/
8. Leroy, X.: Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 42–54. POPL '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1111037.1111042, https://doi.org/10.1145/1111037.1111042
9. Lynch, N., Vaandrager, F.: Forward and backward simulations. Information and Computation **121**(2), 214–233 (1995). https://doi.org/https://doi.org/10.1006/inco.1995.1134, https://www.sciencedirect.com/science/article/pii/S0890540185711340
10. Mani Chandy, K., Misra, J.: Parallel program design: a foundation. Addison-Wesley, Reading (1988)
11. Roncken, M., Gilla, S.M., Park, H., Jamadagni, N., Cowan, C., Sutherland, I.: Naturalized communication and testing. In: 2015 21st IEEE International Symposium on Asynchronous Circuits and Systems. pp. 77–84 (2015). https://doi.org/10.1109/ASYNC.2015.20
12. Roncken, M., Sutherland, I., Chen, C., Hei, Y., Hunt, W., Chau, C., Gilla, S.M., Park, H., Song, X., He, A., Chen, H.: How to think about self-timed systems. In: 2017 51st Asilomar Conference on Signals, Systems, and Computers. pp. 1597–1604 (2017). https://doi.org/10.1109/ACSSC.2017.8335628
13. Solar-Lezama, A., Rabbah, R., Bodík, R., Ebcioundefinedlu, K.: Programming by sketching for bit-streaming programs. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 281–294. PLDI '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1065010.1065045, https://doi.org/10.1145/1065010.1065045
14. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. SIGARCH Comput. Archit. News **34**(5), 404–415 (oct 2006). https://doi.org/10.1145/1168919.1168907, https://doi.org/10.1145/1168919.1168907

15. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006. pp. 404–415 (2006). https://doi.org/10.1145/1168857.1168907, http://doi.acm.org/10.1145/1168857.1168907
16. Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. SIGPLAN Not. **49**(6), 530–541 (Jun 2014). https://doi.org/10.1145/2666356.2594340, https://doi.org/10.1145/2666356.2594340
17. Udding, J.T.: A formal model for defining and classifying delay-insensitive circuits and systems. Distributed Computing **1**(4), 197–204 (1986). https://doi.org/10.1007/BF01660032, https://doi.org/10.1007/BF01660032
18. Van Geffen, J., Nelson, L., Dillig, I., Wang, X., Torlak, E.: Synthesizing jit compilers for in-kernel dsls. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 564–586. Springer International Publishing, Cham (2020)