Pipit on the Post: Proving Pre- and Post-conditions of Reactive Systems

Amos Robinson
Australian National University, Canberra, Australia

Alex Potanin
Australian National University, Canberra, Australia

Abstract

Reactive languages such as Lustre and Scade are used to implement safety-critical control systems; proving such programs correct and having the proved properties apply to the compiled code is therefore equally critical. We introduce Pipit, a small reactive language embedded in F*, designed for verifying control systems and executing them in real-time. Pipit includes a verified translation to transition systems; by reusing F*'s existing proof automation, certain safety properties can be automatically proved by k-induction on the transition system. Pipit can also generate executable code in a subset of F* which is suitable for compilation and real-time execution on embedded devices. The executable code is deterministic and total and preserves the semantics of the original program.

2012 ACM Subject Classification Computer systems organization → Real-time languages; Theory of computation → Program verification; Software and its engineering → Specialized application languages

Keywords and phrases Lustre, streaming, reactive, verification

1 Introduction

Safety-critical control systems, such as the anti-lock braking systems that are present in most cars today, need to be correct and execute in real-time. One approach, favoured by parts of the aerospace industry, is to implement the controllers in a high-level language such as Lustre [10] or Scade [13], and verify that the implementations satisfy the high-level specification using a model-checker, such as Kind2 [11]. These model-checkers can prove many interesting safety properties automatically, but do not provide many options for manual proofs when the automated proof techniques fail. Additionally, the semantics used by the model-checker may not match the semantics of the compiled code, in which case properties proved do not necessarily hold on the real system. This mismatch may occur even when the compiler has been verified to be correct, as in the case of Vélus [5]. For example, in Vélus, integer division rounds towards zero, matching the semantics of C; however, integer division in Kind2 rounds to negative infinity, matching SMT-lib [2, 25].

To be confident that our proofs hold on the real system, we need a single semantics that is shared between the compiler and the model-checker or prover. In this paper we introduce Pipit[1], an embedded domain-specific language for implementing and verifying controllers in F*. Pipit aims to provide a high-level language based on Lustre, while reusing F*'s proof automation and manual proofs for verifying controllers [31], and using Low*'s C-code generation for real-time execution [34]. To verify programs, Pipit translates its expression language to a transition system for k-inductive proofs, which is verified to be an abstraction of the original semantics. To execute programs, Pipit can generate executable code, which is total and semantics-preserving.

In this paper, we make the following contributions:

[1] Implementation available at `https://github.com/songlarknet/pipit`

- we motivate the need to combine manual and automated proofs of reactive systems with a strong specification language (Section 2);
- we introduce Pipit, a minimal reactive language that supports rely-guarantee contracts and properties; crucially, proof obligations are annotated with a status — *valid* or *deferred* — allowing proofs to be delayed until more is known of the program context (Section 3);
- we describe a *checked semantics* for Pipit, which is parameterised by the property status; after checking deferred properties, programs can be *blessed*, and their properties lifted to valid status (Subsection 3.2);
- we describe an encoding of transition systems that can express under-specified rely-guarantee contracts as functions rather than relations; composing functions results in simpler transition systems (Section 4);
- we identify the invariants and lemmas required to prove that the abstract transition system is an abstraction of the original semantics (Subsection 3.3, Subsection 4.1);
- similarly, we offer a mechanised proof that the executable transition system preserves the original semantics (Section 5);
- finally, we evaluate Pipit by implementing the high-level logic of a time-triggered Controller Area Network (CAN) bus driver, which we have partially verified (Section 6).

2 Pipit for time-triggered networks

To introduce Pipit, we consider a driver with a static schedule of *triggers*, or actions to be performed at a particular time; this driver is a simplification of the time-triggered Controller Area Network (CAN) bus specification [15] which we will discuss further in Section 6.

2.1 Deferring and proving properties

The schedule of our time-triggered driver is determined by a constant array of triggers, sorted by their associated time-mark. The driver maintains an index that refers to the current trigger. At each instant in time, the driver checks if the current trigger has expired or is inactive, and if so, it increments the index. We first implement a streaming function *count_when* to maintain the index; the function takes a constant natural number *max* and a stream of booleans *inc*. At each time step, *count_when* checks whether the current increment flag is true; if so, it increments the previous counter, saturating at the maximum; otherwise, it leaves the previous counter as-is.

```
let count_when (max: ℕ) (inc: stream 𝔹): stream ℕ =
  rec count.
    check_[?] (0 ≤ count ≤ max);
    let count' = (0 fby count) + (if inc then 1 else 0) in
    if  count' ≥ max then max else count'
```

The implementation of *count_when* first defines a recursive stream, *count*, which states an invariant about the count before defining the incremented stream *count'*. Inside *count'*, the syntax 0 `fby` *count* is read as "the initial value of zero *followed by* the previous count".

The syntax $\texttt{check}_{\boxed{?}}$ $(0 \leq \mathit{count} \leq \mathit{max})$ asserts that the count is within the range $[0, \mathit{max}]$. The subscript $\boxed{?}$ on the check is the *property status*, which in this case denotes that the assertion has been stated, but it is not yet known whether it holds. A property status of $\boxed{\checkmark}$, on the other hand, denotes that a property has been proved to hold. These property statuses are used to defer checking properties until enough is known about the environment, and to avoid rechecking properties that have already been proven. In practice, the user does not explicitly specify property statuses in the source language. The stated property ($0 \leq count \leq max$) is a stream of booleans which must always be true. Non-streaming operations such as $\leq$ are implicitly lifted to streaming operations, and non-streaming values such as 0 and *max* are implicitly lifted to constant streams.

We defer the proof of the property here because, at the point of stating the property inside the `rec` combinator, we don't yet have a concrete definition for the count variable. In this case, we could have instead deferred the *statement* of the property by introducing a let-binding for the recursive count and putting the `check` outside of the `rec` combinator. However, it is not always possible to defer property statements: for example, when calling other streaming functions that have their own preconditions, it may not be possible to move the function call outside of its enclosing `rec`.

Pipit is an embedded domain-specific language. The program above is really syntactic sugar for an F⋆ program that takes a natural number and constructs a Pipit core expression with a free boolean variable. We will discuss the details of the core language in Section 3, but for now we focus on the source program with some minor embedding details omitted.

To actually prove the property above, we use the meta-language F⋆'s tactics to translate the program into a transition system and prove the property inductively on the system. Finally, we *bless* the expression, which marks the properties as valid ([$\boxed{?}$:= ☑]). Blessing is an intensional operation: it traverses the expression and updates the internal metadata, but it does not affect the runtime semantics.

```
let count_when☑ (max: ℕ): stream 𝔹 → stream ℕ =
  let system = System.translate₁(count_when max) in
  assert (System.inductive_check system) by (pipit_simplify ());
  bless₁ (count_when max)
```

The subscript 1 in the translation to transition system and blessing operations refers to the fact that the stream function has one stream parameter. The *pipit_simplify* tactic in the assertion performs normalisation-by-evaluation to simplify away the translation to a first-order transition system; F⋆'s proof-by-SMT can then solve the inductive check directly.

Callers of *count_when* can now use the validated variant without needing to re-prove the count-range property. In a dedicated model-checker such as Kind2 [11] or Lesar [35], this kind of bookkeeping would all be performed under-the-hood. By embedding Pipit in a general-purpose theorem prover, we move some of the bookkeeping burden onto the user; however, we have increased confidence that the compiled code matches the verified code and, as we shall see, we also have access to a rich specification language.

2.2 The time-triggered system matrix

The schedule of the time-triggered network is abstractly described by a *system matrix*, consisting of rows of *basic cycles*, columns of *transmission columns*, and cells of optional messages. Each basic cycle is identified by its cycle index and each transmission column has an associated time-mark.

Figure 1 (left) shows an example system matrix with cycles `C0` and `C1` and transmission columns at time-marks 0, 1 and 2. For this example, we assume that one message can be sent per clock cycle. To execute this system matrix, we synchronise the local time to zero at the start of basic cycle `C0`. After a basic cycle completes, the nodes on the network synchronise before execution continues to the next basic cycle.

	TM0	TM1	TM2
C0	MSG A	MSG B	-
C1	MSG A	-	MSG C

```
0: { time_mark = 0; enabled = {C0,C1}; msg = A; }
1: { time_mark = 1; enabled = {C0};    msg = B; }
2: { time_mark = 2; enabled = {C1};    msg = C; }
```

Figure 1 Left: system matrix; right: corresponding triggers array configuration

Figure 1 (right) shows the corresponding configuration for the triggers array. The enabled set denotes the basic cycles for which a trigger is active.

The system has strict timing requirements which restrict how triggers can be defined. In this example, each trigger has a unique time; in general, trigger times can overlap, but they need to be enabled on distinct cycles. Additionally, the schedule must allow sufficient time for the driver to skip over the disabled triggers. Concretely, we could postpone trigger 1 to send message `B` at time-mark 2, as triggers 1 and 2 have distinct cycles. However, we could not bring forward trigger 2 to send message `C` at time-mark 1: the driver can only process one trigger per tick, and it takes two steps to reach trigger 2 from the start of the array.

We impose three main restrictions on the triggers array: the time-marks must be sorted; there must be an adequate time-gap between any two triggers that are enabled on the same cycle index; and each trigger's time-mark must be greater-than-or-equal to its index.

With these restrictions in place, we prove a lemma *lemma_can_reach_next*, which states that for all valid cycle indices and trigger indices, if the current trigger is enabled in the current cycle and there is another enabled trigger scheduled to occur somewhere in the array after the current one, then there is an adequate time-gap to allow the driver to skip over any disabled triggers in-between. These properties are straightforward in a theorem prover, but would be difficult to state in a model-checker with a limited specification language.

2.3 Instantiating lemmas and defining contracts

We can now implement the trigger-fetch logic, which keeps track of the current trigger. The trigger-fetch logic uses the *count_when* streaming function to define the index of the current trigger; we tell *count_when* to increment the index whenever the previous index has expired or is inactive in the current basic cycle. We simplify our presentation here and only consider a single cycle in isolation: the real system presented in Section 6 has some extra complexity such as resetting the index, incrementing the cycle index at the start of a new cycle, and using machine integers.

```
let trigger_fetch (cycle: ℕ) (time: stream ℕ): stream ℕ =
  rec index.
    let inc = false fby ((time_mark index) ≤ time ∨ ¬(enabled index cycle)) in
    let index = count_when_☑ trigger_count inc in
    pose_1 (lemma_can_reach_next cycle) index;
    check_? (can_reach_next_active cycle time index);
    index
```

The *trigger_fetch* function takes a static cycle index and a stream denoting the current time. The increment flag and the index are mutually dependent — the increment flag depends on the previous value of the index, while the index depends on the current value of the increment flag — so we introduce a recursive stream for the index. We allow the index to go one past the end of the array to denote that there are no more triggers.

We use the *pose*$_1$ helper function to lift the *lemma_can_reach_next* lemma to a streaming context and instantiate it; the subscript 1 indicates that the lemma is being applied to one streaming argument (the index). We then state an invariant as a deferred property. Informally, the invariant states that, either the current active trigger is not late, or the next active trigger after the current index is in the future and we can reach it in time.

With the explicitly instantiated lemma, we can prove the streaming invariant by straightforward induction on the transition system. To help compose this function with the rest of the system, we also abstract over the details of the trigger-fetch mechanism by introducing a rely-guarantee contract for *trigger_fetch*. The contract we state is that if the environment ensures that the time doesn't skip — that is, we are called once per microsecond — then we guarantee that we never encounter a late trigger.

```
let trigger_fetch☑ (cycle: ℕ): stream ℕ → stream ℕ =
  let contract = Contract.contract_of_stream₁ {
    rely = (λtime. time_no_skips time)
    guar = (λtime index. (index_valid index ∧ enabled index cycle)
                ⟹ (time_mark index) ≥ time)
    body = (λtime. trigger_fetch cycle time)
  } in
  assert (Contract.inductive_check contract) by (pipit_simplify ());
  Contract.stream_of_contract₁ contract
```

In the implementation of the validated variant of *trigger_fetch*, we first construct the contract from streaming functions. The Contract.contract_of_stream$_1$ combinator describes a contract with one input (the time stream), and takes stream transformers for each of the rely, guarantee and body. The combinator transforms the surface syntax into core expressions. The assertion (Contract.inductive_check *contract*) then translates the expressions into a transition system, and checks that if the rely always holds then the guarantee always holds, and that the as-yet-unchecked subproperties hold. Finally, Contract.stream_of_contract$_1$ blesses the core expression and converts it back to a stream transformer, so it can be easily used by other parts of the program.

When this function is used in other parts of the program, the caller must ensure that the environment satisfies the rely clause. In the core language, this is tracked by another deferred property status attached to the contract; we will discuss this further in Section 3.

3 Core language

We now introduce the core Pipit language. Note that this form differs slightly from the surface syntax presented earlier in Section 2, which used the syntax of the metalanguage F*, as well as including proofs in F* itself.

Figure 2 defines the grammar of Pipit. The expression form e includes standard syntax for values (v), variables (x) and primitive applications ($p(\overline{e})$). Most of the expression forms were introduced informally in Section 2 and correspond to the clock-free expressions of Lustre [10].

The expression syntax for delayed streams (v `fby` e) denotes the previous value of the stream e, with an initial value of v when there is no previous value.

Recursive streams, which can refer to previous values of the stream itself, are defined using the fixpoint operator (`rec` $x.\ e[x]$); the syntax $e[x]$ means that the variable x can occur in e. As in Lustre, recursive streams can only refer to their previous values and must be *guarded* by a delay: the stream (`rec` $x.$ 0 `fby` $(x + 1)$) is well-defined, but stream (`rec` $x.\ x + 1$) is invalid and has no computational interpretation. This form of recursion differs slightly from standard Lustre, which uses a set of mutually-recursive bindings. Although we cannot express

$$
\begin{array}{lcll}
e, e' & := & v \mid x \mid p(\overline{e}) & \text{(values, variables and operations)} \\
 & \mid & v\ \texttt{fby}\ e \mid \texttt{rec}\ x.\ e[x] & \text{(delayed and recursive streams)} \\
 & \mid & \texttt{let}\ x = e\ \texttt{in}\ e'[x] & \text{(let-expressions)} \\
 & \mid & \texttt{check}_\pi\ e_{\text{prop}} & \text{(checked properties)} \\
 & \mid & \texttt{contract}_\pi\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\} & \text{(rely-guarantee contracts)} \\
 & & & \\
v & := & n \in \mathbb{N} \mid b \in \mathbb{B} \mid r \in \mathbb{R} \mid \ \ldots & \text{(values)} \\
p & := & (+) \mid (-) \mid (\times) \mid \texttt{if-then-else} \mid \ \ldots & \text{(primitives)} \\
 & & & \\
\pi & := & \boxed{\checkmark} \mid \boxed{?} & \text{(property statuses: valid or unknown)} \\
 & & & \\
V & := & \cdot \mid V; v & \text{(streams of values)} \\
\sigma & := & \{\overline{x \mapsto v}\} & \text{(heaps)} \\
\Sigma & := & \cdot \mid \Sigma; \sigma & \text{(streaming history environments)} \\
\tau, \tau' & := & \mathbb{N} \mid \mathbb{B} \mid \tau \times \tau \mid \ \ldots & \text{(value types)} \\
\Gamma & := & \cdot \mid x : \tau, \Gamma & \text{(type environments)}
\end{array}
$$

Figure 2 Pipit core language grammar, which contains expressions e, values v, primitive operations p, and property statuses π.

mutually-recursive bindings in the core syntax here, we can express them as a notation on the surface syntax by combining the bindings together into a single tuple.

Checked properties and contracts are annotated with their property status π, which can either be valid ($\boxed{\checkmark}$) or unknown ($\boxed{?}$). For checked properies $\texttt{check}_\pi\ e$, the property status denotes whether the property has been proved to be valid.

Contracts $\texttt{contract}_\pi\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\}$ involve two verification conditions. Firstly, when a contract is *defined*, the definer must prove that the body e_{body} satisfies the contract: roughly, if e_{rely} is always true, then $e_{\text{guar}}[x := e_{\text{body}}]$ is always true. Secondly, when a contract is *instantiated*, the caller must prove that the environment satisfies the precondition: that is, e_{rely} is always true. Conceptually, then, a contract could have two property statuses: one for the definer, and one for the instantiation. However, in practice, it is not useful to defer the proof of a contract definition — one could achieve a similar effect by replacing the contract with its implementation. For this reason, we only annotate contracts with one property status, which denotes whether the instantiation has been proved to satisfy the precondition.

Streams V are represented as a sequence of values; streaming history environments Σ are streams of heaps. Types τ and type environments Γ are standard.

We define the typing judgments for Pipit in Figure 3. Most of the typing rules are standard for an unclocked Lustre. The typing judgment $\Gamma \vdash e : \tau$ denotes that, in an environment of streams Γ, expression e denotes a stream of type τ. This core typing judgment differs from the surface syntax used in Section 2, which used an explicit stream type; for the core language, we instead assume that everything is a stream.

For values, we use an auxiliary judgment form prim-value-type$(v) = \tau$ to denote that value v has type τ. Likewise, for primitives we use the auxiliary judgment form prim-type$(p) = (\tau_1 \times \cdots \ldots \times \tau_n) \to \tau'$ to denote that primitive p takes arguments of type τ_i and returns a result of type τ'. Primitives are pure, non-streaming functions.

Rules TValue, TVar, TPrim and TLet are standard.

Rule TFby states that expression $v\ \texttt{fby}\ e$ requires both v and e to have equal types; the

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\text{prim-value-type}(v) = \tau}{\Gamma \vdash v : \tau}\ (\text{TV}\textsc{alue}) \qquad \frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}\ (\text{TV}\textsc{ar})$$

$$\frac{\text{prim-type}(p) = (\tau_1 \times \cdots \times \tau_n) \to \tau' \qquad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash p(\overline{e}) : \tau'}\ (\text{TP}\textsc{rim})$$

$$\frac{\text{prim-value-type}(v) = \tau \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash v\ \texttt{fby}\ e' : \tau}\ (\text{TF}\textsc{by}) \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{rec}\ \ x.\ e[x] : \tau}\ (\text{TR}\textsc{ec})$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \texttt{let}\ \ x = e\ \ \texttt{in}\ \ e'[x] : \tau'}\ (\text{TL}\textsc{et}) \qquad \frac{\Gamma \vdash e : \mathbb{B}}{\Gamma \vdash \texttt{check}_\pi\ e : \texttt{unit}}\ (\text{TC}\textsc{heck})$$

$$\frac{\Gamma \vdash e_{\text{rely}} : \mathbb{B} \qquad \Gamma \vdash e_{\text{body}} : \tau \qquad \Gamma, x : \tau \vdash e_{\text{guar}} : \mathbb{B}}{\Gamma \vdash \texttt{contract}_\pi\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\} : \tau}\ (\text{TC}\textsc{ontract})$$

Figure 3 Typing rules for Pipit; the judgment $\Gamma \vdash e : \tau$ denotes that expression e describes a *stream* of values of type τ. Two auxiliary judgment forms are used for values and primitive operations; their rules are standard and are omitted.

result is the same type.

Rule TREC states that a recursive stream $\texttt{rec}\ \ x.\ e$ has the recursive stream bound inside e. The recursion must also be guarded, in that any recursive references to x are delayed, but this requirement is performed as a separate syntactic check described in Subsection 3.3.

Rule TCHECK states that statically checking a property $\texttt{check}_\pi\ e$ requires a boolean property e and returns unit.

Finally, rule TCONTRACT applies for a contract $\texttt{contract}_\pi\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\}$ with a body expression of some type τ. The overall expression has result type τ. Both rely and guarantee clauses must be boolean expressions. Additionally, the guarantee clause can refer to the result value by x.

3.1 Dynamic semantics

The dynamic semantics of Pipit are defined in Figure 4. We present our semantics in a big-step form. This differs somewhat from traditional *reactive* semantics of Lustre [10]. Our big-step semantics emphasises the equational nature of Pipit, as it is substitution-based, while the reactive semantics emphasises the finite-state streaming execution of the system. We use transition systems for reasoning about the finite-state execution (Section 4), which is fairly standard [9, 11, 35]. Previous work on the W-CALCULUS [17] for linear digital signal processing filters makes a similar distinction and provides a non-streaming semantics for reasoning about programs and a streaming semantics for executing programs.

The judgment form $\Sigma \vdash e \Downarrow v$ denotes that expression e evaluates to value v under streaming history Σ. The streaming history is a stream of heaps; in practice, we only evaluate expressions with a non-empty streaming history.

Rule VALUE states that evaluating a value results in the value itself.

$$\boxed{\Sigma \vdash e \Downarrow v}$$

$$\frac{}{\Sigma \vdash v \Downarrow v}\ (\textsc{Value}) \qquad \frac{}{\Sigma; \sigma \vdash x \Downarrow \sigma(x)}\ (\textsc{Var})$$

$$\frac{\Sigma \vdash e_1 \Downarrow v_1 \quad \ldots \quad \Sigma \vdash e_n \Downarrow v_n}{\Sigma \vdash p(\overline{e}) \Downarrow \text{prim-sem}(p, \overline{v})}\ (\textsc{Prim})$$

$$\frac{}{\sigma \vdash v \ \texttt{fby}\ e' \Downarrow v}\ (\textsc{Fby}_1) \qquad \frac{\text{length}(\Sigma) > 0 \qquad \Sigma \vdash e' \Downarrow v'}{\Sigma; \sigma \vdash v \ \texttt{fby}\ e' \Downarrow v'}\ (\textsc{Fby}_S)$$

$$\frac{\Sigma \vdash e[x := \texttt{rec}\ \ x.\ e] \Downarrow v}{\Sigma \vdash \texttt{rec}\ \ x.\ e[x] \Downarrow v}\ (\textsc{Rec}) \qquad \frac{\Sigma \vdash e'[x := e] \Downarrow v}{\Sigma \vdash \texttt{let}\ \ x = e\ \ \texttt{in}\ \ e'[x] \Downarrow v}\ (\textsc{Let})$$

$$\frac{}{\Sigma \vdash \texttt{check}_\pi\ e \Downarrow ()}\ (\textsc{Check})$$

$$\frac{\Sigma \vdash e_{\text{body}} \Downarrow v}{\Sigma \vdash \texttt{contract}_\pi\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\} \Downarrow v}\ (\textsc{Contract})$$

$$\boxed{\Sigma \vdash e \Downarrow^* V} \qquad \boxed{\Sigma \vdash e \Downarrow^{\Box} \top}$$

$$\frac{}{\cdot \vdash e \Downarrow^* \cdot}\ (\textsc{Steps}_0) \qquad \frac{\Sigma \vdash e \Downarrow V \qquad \Sigma; \sigma \vdash e \Downarrow v}{\Sigma; \sigma \vdash e \Downarrow V; v}\ (\textsc{Steps}_S)$$

$$\frac{\Sigma \vdash e \Downarrow^* \top; \ldots}{\Sigma \vdash e \Downarrow^{\Box} \top}\ (\textsc{Always})$$

Figure 4 Dynamic semantics for Pipit; the judgment form $\Sigma \vdash e \Downarrow v$ denotes that evaluating expression e under streaming history Σ results in value v.

Rule Var states that to evalute a variable x under some non-empty stream history $\Sigma; \sigma$, where σ is the most recent heap, we look up the variable in σ.

Rule Prim states that to evaluate a primitive p applied to many arguments e_1 to e_n, we evaluate each argument separately; we then use the prim-sem metafunction to apply the primitive.

Rule Fby_1 evaluates a followed-by expression when the streaming history contains only a single element. Here, v `fby` e evaluates to v, as there is no previous value of e to use.

Rule Fby_S evaluates a followed-by expression when the streaming history contains multiple entries. In this case, v `fby` e proceeds to evaluate the previous value of e by discarding the most recent entry from the streaming history.

Rule Rec evaluates a recursive stream `rec` $x.\ e$ by unfolding the recursion one step. For causal expressions (Subsection 3.3), where each recursive occurrence of x is guarded by a followed-by, this unfolding will eventually terminate, as the follow-by shortens the streaming history.

Rule Let is standard.

Rule Check states that check expressions always evaluate to unit. We do not perform a dynamic check that the property is true here; checking the truth of properties is dealt with in the checked semantics (Subsection 3.2).

Rule Contract states that contracts evaluate by just evaluating their body. Like with checks, we do not perform a dynamic check that the precondition and postcondition hold.

We also use two auxiliary judgment forms: $\Sigma \vdash e \Downarrow^* V$ and $\Sigma \vdash e \Downarrow^\square \top$.

Judgment form $\Sigma \vdash e \Downarrow^* V$ denotes that, under streaming history Σ, expression e evaluates to the *stream* V. This judgment is an iterated application of the single-value big-step form.

Judgment form $\Sigma \vdash e \Downarrow^\square \top$ denotes that expression e, which must be a boolean, evaluates to the stream of trues under history Σ. Informally, it can be read as "in streaming history Σ, e is always true".

3.2 Checked semantics

In addition to the big-step semantics above, we also define a judgment form for checking that the properties and contracts of a program hold for a particular streaming history. We call these the *checked* semantics. Unlike an axiomatic semantics, the checked semantics operate on a concrete set of input streams.

The checked semantics have the judgment form $\Sigma \vdash_\pi e$ valid, which denotes that under streaming history Σ, the properties of e with status π hold. The property status dictates which properties should be checked and which should be ignored.

To show that an expression e's unknown properties hold, we prove that for all streaming histories Σ, assuming the valid properties hold ($\Sigma \vdash_{\boxed{\checkmark}} e$ valid), then the unknown properties ($\Sigma \vdash_{\boxed{?}} e$ valid) hold. The assumption here means that we do not have to re-check properties after proving them once.

Contracts involve two proofs: one for the definition and one for the instantiation. To prove that a contract definition $\texttt{contract}_\pi\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\}$ is valid, we show that for all streaming histories Σ, assuming the rely is always true under the history ($\Sigma \vdash e_{\text{rely}} \Downarrow^\square \top$), then the body always satisfies the guarantee ($\Sigma \vdash e_{\text{guar}}[x := e_{\text{body}}] \Downarrow^\square \top$). Additionally, we can also assume that the valid properties in all three components hold, and we must also show that the unknown properties are valid. The fact that the checked semantics refers to a particular Σ is significant here: it allows the proof of contract validity to only consider

$$\boxed{\Sigma \vdash_\pi e \text{ valid}}$$

$$\frac{}{\Sigma \vdash_\pi v \text{ valid}}\ (\textsc{ChkValue}) \qquad \frac{}{\Sigma \vdash_\pi x \text{ valid}}\ (\textsc{ChkVar})$$

$$\frac{\Sigma \vdash_\pi e_1 \text{ valid} \quad \ldots \quad \Sigma \vdash_\pi e_n \text{ valid}}{\Sigma \vdash_\pi p(\overline{e}) \text{ valid}}\ (\textsc{ChkPrim})$$

$$\frac{}{\sigma \vdash_\pi v\ \texttt{fby}\ e' \text{ valid}}\ (\textsc{ChkFby}_1) \qquad \frac{\text{length}(\Sigma) > 0 \qquad \Sigma \vdash_\pi e' \text{ valid}}{\Sigma; \sigma \vdash_\pi v\ \texttt{fby}\ e' \text{ valid}}\ (\textsc{ChkFby}_S)$$

$$\frac{\Sigma \vdash_\pi e[x := \texttt{rec}\ \ x.\ e] \text{ valid}}{\Sigma \vdash_\pi \texttt{rec}\ \ x.\ e[x] \text{ valid}}\ (\textsc{ChkRec}) \qquad \frac{\Sigma \vdash_\pi e'[x := e] \text{ valid}}{\Sigma \vdash_\pi \texttt{let}\ \ x = e\ \ \texttt{in}\ \ e'[x] \text{ valid}}\ (\textsc{ChkLet})$$

$$\frac{(\pi = \pi' \implies \Sigma \vdash e \Downarrow^{\square} \top) \qquad \Sigma \vdash_\pi e \text{ valid}}{\Sigma \vdash_\pi \texttt{check}_{\pi'}\ e \text{ valid}}\ (\textsc{ChkCheck})$$

$$\frac{\begin{array}{c}(\pi = \pi' \implies \Sigma \vdash e_{\text{rely}} \Downarrow^{\square} \top) \\ (\pi = \boxed{\checkmark} \implies \Sigma \vdash e_{\text{rely}} \Downarrow^{\square} \top \implies \Sigma \vdash e_{\text{guar}}[x := e_{\text{body}}] \Downarrow^{\square} \top) \\ \Sigma \vdash_\pi e_{\text{rely}} \text{ valid} \\ (\Sigma \vdash e_{\text{rely}} \Downarrow^{\square} \top \implies \Sigma \vdash_\pi e_{\text{body}} \text{ valid} \ \wedge\ \Sigma \vdash_\pi e_{\text{guar}}[x := e_{\text{body}}] \text{ valid})\end{array}}{\Sigma \vdash_\pi \texttt{contract}_{\pi'}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\} \text{ valid}}\ (\textsc{ChkContract})$$

Figure 5 Checked semantics for Pipit; the judgment form $\Sigma \vdash_\pi e$ valid denotes that evaluating expression e under streaming history Σ satisfies the checks and rely-guarantee contract requirements that are labelled with property status π.

streaming histories where the rely actually holds. ***TODO:*** *WAFFLE? This is expanded on in Subsubsection 3.2.1; kill here?*

To prove that a contract *instantiation* (a call-site) is valid, we show that, under the calling environment, the rely clause is always true. Crucially, the proof can also use the fact that, if the rely is always true, then the guarantee is always true. This sort of feedback is necessary for proving properties of mutually-dependent calls. Although this feedback appears circular, we enforce causality by requiring that occurrences of recursive streams are guarded by delays (Subsection 3.3).

We define the checked semantics of Pipit in Figure 5. The checked semantics mostly follows the structure of the dynamic semantics, checking any properties and contracts as they are encountered.

Rules ChkValue and ChkVar state that values and variables are always valid.

Rule ChkPrim checks a primitive application by descending into the subexpressions.

Rules ChkFby_1 and ChkFby_S are derived from the structure of the big-step rules Fby_1 and Fby_S. At an input stream of length one, ChkFby_1 asserts that all subproperties hold for the (non-existent) previous values in the stream. At subsequent parts of the stream, ChkFby_S discards the most recent element of the stream history and checks the subexpression with the previous inputs.

Rules ChkRec and ChkLet both perform the same unfolding as the corresponding big-step rules and check the resulting expression.

Finally, the heavy lifting is performed by rules ChkCheck and ChkContract.

Rule ChkCheck applies when checking property status π of an expression $\texttt{check}_{\pi'}\ e$. If the check-expression has the same status as what we are checking ($\pi = \pi'$), then we perform the actual check by evaluating the expression e and requiring it to evaluate to a stream of trues. Otherwise, we do not need to evaluate the check-expression. In both cases, we descend into the expression and check its subexpressions, as they may have nested properties. Such nested properties are unlikely to be written directly by the user, but might occur after program transformations such as inlining.

Rule ChkContract applies when checking property status π of a contract with expression $\texttt{contract}_{\pi'}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\}$. Although we only include one property status on the contract, conceptually there are two distinct properties: one for the caller (π') and one for the definition itself (assumed to be ☑). To check the caller property when $\pi = \pi'$, we evaluate the rely e_{rely} and require it to be true. To check the definition property when $\pi =$ ☑, we assume that the rely holds, and check that the body satisfies the guarantee. We also descend into the subexpressions to check them; when checking the body and guarantee, we can assume that the rely holds. Unfortunately, this rule must deal with the two different roles of a contract at once; in the next section, we will separate the two roles.

3.2.1 Blessing expressions and contracts

Blessing is a meta-operation that replaces the property statuses in an expression so that all checks and contracts are marked as valid (☑). Blessing an expression requires a proof that the checked semantics hold for all input streams:

$$\frac{\forall \Sigma.\ \Sigma \vdash_{\boxed{\checkmark}} e \text{ valid} \implies \Sigma \vdash_{\boxed{?}} e \text{ valid}}{\text{bless } e}\ (\textsc{BlessExpression})$$

Blessing is slightly different for contract definitions, as we need to separate the definition of the contract from the instantiation. To check that a contract definition is valid, we show that if the rely clause is always true for a particular input, then the body satisfies the guarantee for the same inputs. We also assume that the valid properties in the rely, body and guarantee hold, and show the corresponding unknown properties:

$$
\begin{array}{l}
\texttt{let}\ \text{contract\_valid}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{e_{\text{guar}}\} : \text{prop} = \\
\quad \forall \Sigma.\ \ (\Sigma \vdash_{\boxed{\checkmark}} (e_{\text{rely}}, e_{\text{body}}, e_{\text{guar}}[x := e_{\text{body}}])\ \text{valid}\ \wedge\ \Sigma \vdash e_{\text{rely}} \Downarrow^{\square} \top) \\
\quad \Longrightarrow\ (\Sigma \vdash_{\boxed{?}} (e_{\text{rely}}, e_{\text{body}}, e_{\text{guar}}[x := e_{\text{body}}])\ \text{valid}\ \wedge\ \Sigma \vdash e_{\text{guar}}[x := e_{\text{body}}] \Downarrow^{\square} \top)
\end{array}
$$

After proving that the contract is valid for all inputs, we can bless the contract definition. Blessing the contract definition blesses the subexpressions for the rely, body and guarantee, but leaves the contract's *instantiation* property status as unknown:

$$
\frac{\text{contract\_valid}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{e_{\text{guar}}\}}{\text{bless\_contract}\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{e_{\text{guar}}\}}\,(\textsc{BlessContract})
$$

3.3 Causality and metatheory

To ensure that recursive streams have a computational interpretation, we require that all recursive streams are guarded by a followed-by delay. We implement this as a simple syntactic check: each $\texttt{rec}\ x.\ e$ can only mention x inside a followed-by. This check is stricter than necessary: for example, the expression $\texttt{rec}\ x.\ (\texttt{let}\ x' = x + 1\ \texttt{in}\ 0\ \texttt{fby}\ x')$ does mention the recursive stream x outside of the delay, but after inlining the let, it would be causal. We hope to relax this restriction somewhat in future work.

The causality restriction gives us some important properties about the metatheory. The most important property is that the dynamic semantics form a total function: given a streaming history and a causal expression, we can evaluate the expression to a value. These properties are mechanised in F⋆.

▶ **Theorem 1** (bigstep-is-total). *For any non-empty streaming history Σ and causal expression e, there exists some value v such that e evaluates to v ($\Sigma \vdash e \Downarrow v$).*

The relationship between substitution and the streaming history is also important. In general, we have a substitution property that states that evaluating a substituted expression $e[x := e']$ under some context Σ is equivalent to evaluating e' and adding it to the context Σ:

▶ **Theorem 2** (bigstep-substitute). *For a streaming history Σ and a causal expression e, if $e[x := e']$ evaluates to a value v ($\Sigma \vdash e \Downarrow v$), then we can evaluate e' to some stream V and extend the streaming history to evaluate e to the original value ($\Sigma[x \mapsto V] \vdash e \Downarrow v$). The converse is also true.*

The semantics in Figure 4 for a recursive expression $\texttt{rec}\ x.\ e$ performs one step of recursion by substituting x for the recursive expression. An alternative semantics would be to have the environment outside the semantics invent a stream V such that if we extend the streaming history with $x \mapsto V$, then e evaluates to V itself. The above substitution theorem can be used to show that these two semantics are equivalent. Thanks to causality, we can additionally show that, when evaluating e with $x \mapsto V$, the most recent value in V does not affect the result. This fact can be used to "seed" evaluation by starting with an arbitrary value:

▶ **Theorem 3** (bigstep-rec-causal). *For a streaming history $\Sigma; \sigma$ and a causal recursive expression $\texttt{rec}\ x.\ e$, if ($\Sigma; \sigma \vdash e \Downarrow v$), then updating $\sigma[x]$ with any value v' results in the same value: ($\Sigma; \sigma[x \mapsto v'] \vdash e \Downarrow v$).*

```
type system (input: Γ) (result: τ) = {
    state:   Γ;
    free:    Γ;
    init:    heap state;
    step:    heap input → heap free → heap state → step_result state result;
}

type step_result (state: Γ) (result: τ) = {
    update:  heap state;
    value:   result;
    rely:    prop;
    guar:    prop;
}
```

Figure 6 Abstract transition system type definitions

4 Abstract transition systems

To prove properties about Pipit programs, we translate to an *abstract* transition system, so-called because it abstracts away the implementation details of contract instantiations. For extraction we also translate to *executable* transition systems, which we discuss in Section 5.

Figure 6 shows the types of transition systems. A transition system is parameterised by its input context and the result type. It also contains two internal contexts: firstly, the state context describes the private state required to execute the machine; secondly, the free context contains any extra input values that the transition system would like to quantify over. The free context is used to allow the system to ask for arbitrary values from the environment, when it would not otherwise be able to return a concrete value.

For contract instantiations, which abstract over the implementation, the natural translation to a transition system would involve an existential quantifier: there exists some value that satisfies the specification. Unfortunately, such an existential quantifier requires a step *relation* rather than a step *function*. Using a step relation complicates the resulting transition system, as other operations such as primitive application must also introduce existential quantifiers; such quantifiers block normalisation and result in a more complex transition system. Instead, the free context provides the step function with a fresh unconstrained value of the desired type, which the step function can then constrain.

As usual, the step-result contains the updated state for the transition system, as well as the result value. The step-result additionally contains two propositions for the 'rely', or assumptions about the execution environment, and 'guarantee', or obligations that the transition system must show. For the transition system corresponding to an expression e, these propositions are analogous to the known checked semantics $\Sigma \vdash_{\boxed{\checkmark}} e$ valid and unknown checks $\Sigma \vdash_{\boxed{?}} e$ valid respectively.

Our implementation includes a mechanised proof that, for causal expressions, the transition system is an abstraction of the original expression's dynamic semantics. The proof that the rely and guarantee propositions correspond to the checked semantics is future work.

Figure 7 defines the internal state and free contexts required for an expression. For most expression forms, the state and free contexts are defined by taking the union of the contexts of subexpressions. Followed-by delays introduce a local state variable $x_{\texttt{fby}}$ in which to store

$$
\begin{array}{rcll}
[\![v]\!]_{\text{state}} & = & \cdot & \\
[\![x]\!]_{\text{state}} & = & \cdot & \\
[\![p(\overline{e})]\!]_{\text{state}} & = & \bigcup_i [\![e_i]\!]_{\text{state}} & \\
[\![v\ \texttt{fby}\ e]\!]_{\text{state}} & = & x_{\texttt{fby}} : \tau, [\![e]\!]_{\text{state}} & \qquad (\text{fresh } x_{\texttt{fby}}) \\
[\![\texttt{rec}\ x.\ e]\!]_{\text{state}} & = & [\![e]\!]_{\text{state}} & \\
[\![\texttt{let}\ x = e\ \texttt{in}\ e']\!]_{\text{state}} & = & [\![e]\!]_{\text{state}} \cup [\![e']\!]_{\text{state}} & \\
[\![\texttt{check}_\pi\ e]\!]_{\text{state}} & = & [\![e]\!]_{\text{state}} & \\
[\![\texttt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\text{state}} & = & [\![e_r]\!]_{\text{state}} \cup [\![e_b]\!]_{\text{state}} &
\end{array}
$$

$$
\begin{array}{rcl}
[\![v]\!]_{\text{free}} & = & \cdot \\
[\![x]\!]_{\text{free}} & = & \cdot \\
[\![p(\overline{e})]\!]_{\text{free}} & = & \bigcup_i [\![e_i]\!]_{\text{free}} \\
[\![v\ \texttt{fby}\ e]\!]_{\text{free}} & = & [\![e]\!]_{\text{free}} \\
[\![\texttt{rec}\ x.\ e]\!]_{\text{free}} & = & x : \tau, [\![e]\!]_{\text{free}} \\
[\![\texttt{let}\ x = e\ \texttt{in}\ e']\!]_{\text{free}} & = & [\![e]\!]_{\text{free}} \cup [\![e']\!]_{\text{state}} \\
[\![\texttt{check}_\pi\ e]\!]_{\text{free}} & = & [\![e]\!]_{\text{free}} \\
[\![\texttt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\text{free}} & = & x : \tau, [\![e_r]\!]_{\text{free}} \cup [\![e_b]\!]_{\text{state}}
\end{array}
$$

Figure 7 Transition system typing contexts of expressions; for an expression e, $[\![e]\!]_{\text{state}} : \Gamma$ and $[\![e]\!]_{\text{free}} : \Gamma$ describe the heaps used to store the expression's internal state and extra inputs.

the most recent stream value. We generate a fresh variable here, though the implementation uses de Bruijn indices. Recursive streams and contracts both introduce new bindings into the free context, assuming that their binders x are unique.

Figure 8 defines the translation for expressions. Values and expressions have no internal state. For variables, we look for the variable binding in either of the input or free heaps; bindings are unique and cannot occur in both. We omit the rely and guarantee definitions here; both are trivially true.

To translate primitives, we union together the initial states of the subexpressions; updating the state is similar. For the rely and guarantee definitions, we take the conjunction: we can assume that all subexpressions rely clauses hold, and must show that all guarantees hold.

To translate a followed-by v `fby` e, we initialise the follow-by's unique binder $x_{\texttt{fby}}$ to v. At each step, we return the value in the local state, before updating the local state to the subexpression's new value. The rely and guarantee differ from the checked semantics here: in the checked semantics, we check the subexpression on the previous inputs, but here we check the current subexpression. This means that a single step of the rely and guarantee do not exactly correspond to the checked semantics; however, we posit that they are equivalent for a rely and guarantee that has been proven to hold for any sequence of inputs.

To translate a recursive expression `rec` $x.\ e$ of type τ, we require an arbitrary value $x : \tau$ in the free heap. The rely proposition constrains the free variable x to be the result of evaluating e with the binding for x passed along, thus closing the recursive loop.

To translate let-expressions `let` $x = e$ `in` e', we extend the input heap with the value of e before evaluating e'. The presentation here duplicates the computation of the value of e, but this is not an issue in practice.

To translate a check property, we inspect the property status. If the property is known to be valid, then we can assume the property is true in the rely clause. Otherwise, we include the property as an obligation in the guarantee clause. In either case, we also include the subexpression's rely and guarantee clauses.

Finally, to translate contract instantiations, we use the contract's rely and guarantee and

$$
\begin{array}{lcl}
[\![v]\!]_{\mathrm{init}} & = & () \\
[\![v]\!]_{\mathrm{value}}(i, f, s) & = & v \\[1em]
[\![x]\!]_{\mathrm{init}} & = & () \\
[\![x]\!]_{\mathrm{value}}(i, f, s) & = & (i \cup f).x \\[1em]
[\![p(\overline{e})]\!]_{\mathrm{init}} & = & \bigcup_i [\![e_i]\!]_{\mathrm{init}} \\
[\![p(\overline{e})]\!]_{\mathrm{value}}(i, f, s) & = & \text{prim-sem}(p, \overline{[\![e]\!]_{\mathrm{value}}(i, f, s)}) \\
[\![p(\overline{e})]\!]_{\mathrm{update}}(i, f, s) & = & \bigcup_i [\![e_i]\!]_{\mathrm{update}}(i, f, s) \\
[\![p(\overline{e})]\!]_{\mathrm{rely}}(i, f, s) & = & \bigwedge_i [\![e_i]\!]_{\mathrm{rely}}(i, f, s) \\
[\![p(\overline{e})]\!]_{\mathrm{guar}}(i, f, s) & = & \bigwedge_i [\![e_i]\!]_{\mathrm{guar}}(i, f, s) \\[1em]
[\![v\ \mathtt{fby}\ e]\!]_{\mathrm{init}} & = & [\![e]\!]_{\mathrm{init}} \cup \{x_{\mathtt{fby}} \mapsto v\} \\
[\![v\ \mathtt{fby}\ e]\!]_{\mathrm{value}}(i, f, s) & = & s.x_{\mathtt{fby}} \\
[\![v\ \mathtt{fby}\ e]\!]_{\mathrm{update}}(i, f, s) & = & [\![e]\!]_{\mathrm{update}}(i, f, s) \cup \{x_{\mathtt{fby}} \mapsto [\![e]\!]_{\mathrm{value}}(i, f, s)\} \\
[\![v\ \mathtt{fby}\ e]\!]_{\mathrm{rely}}(i, f, s) & = & [\![e]\!]_{\mathrm{rely}}(i, f, s) \\
[\![v\ \mathtt{fby}\ e]\!]_{\mathrm{guar}}(i, f, s) & = & [\![e]\!]_{\mathrm{guar}}(i, f, s) \\[1em]
[\![\mathtt{rec}\ x.\ e]\!]_{\mathrm{init}} & = & [\![e]\!]_{\mathrm{init}} \\
[\![\mathtt{rec}\ x.\ e]\!]_{\mathrm{value}}(i, f, s) & = & f.x \\
[\![\mathtt{rec}\ x.\ e]\!]_{\mathrm{update}}(i, f, s) & = & [\![e]\!]_{\mathrm{update}}(i, f, s) \\
[\![\mathtt{rec}\ x.\ e]\!]_{\mathrm{rely}}(i, f, s) & = & [\![e]\!]_{\mathrm{rely}}(i, f, s) \\
 & \wedge & f.x = [\![e]\!]_{\mathrm{value}}(i, f, s) \\
[\![\mathtt{rec}\ x.\ e]\!]_{\mathrm{guar}}(i, f, s) & = & [\![e]\!]_{\mathrm{guar}}(i, f, s) \\[1em]
[\![\mathtt{let}\ x = e\ \mathtt{in}\ e']\!]_{\mathrm{init}} & = & [\![e]\!]_{\mathrm{init}} \cup [\![e']\!]_{\mathrm{init}} \\
[\![\mathtt{let}\ x = e\ \mathtt{in}\ e']\!]_{\mathrm{value}}(i, f, s) & = & [\![e']\!]_{\mathrm{value}}(i \cup \{x \mapsto [\![e]\!]_{\mathrm{value}}(i, f, s)\}, f, s) \\
[\![\mathtt{let}\ x = e\ \mathtt{in}\ e']\!]_{\mathrm{update}}(i, f, s) & = & [\![e']\!]_{\mathrm{update}}(i \cup \{x \mapsto [\![e]\!]_{\mathrm{value}}(i, f, s)\}, f, s) \\
 & \cup & [\![e]\!]_{\mathrm{update}}(i, f, s) \\
[\![\mathtt{let}\ x = e\ \mathtt{in}\ e']\!]_{\mathrm{rely}}(i, f, s) & = & [\![e']\!]_{\mathrm{rely}}(i \cup \{x \mapsto [\![e]\!]_{\mathrm{value}}(i, f, s)\}, f, s) \\
 & \wedge & [\![e]\!]_{\mathrm{rely}}(i, f, s) \\
[\![\mathtt{let}\ x = e\ \mathtt{in}\ e']\!]_{\mathrm{guar}}(i, f, s) & = & [\![e']\!]_{\mathrm{guar}}(i \cup \{x \mapsto [\![e]\!]_{\mathrm{value}}(i, f, s)\}, f, s) \\
 & \wedge & [\![e]\!]_{\mathrm{guar}}(i, f, s) \\[1em]
[\![\mathtt{check}_\pi\ e]\!]_{\mathrm{init}} & = & [\![e]\!]_{\mathrm{init}} \\
[\![\mathtt{check}_\pi\ e]\!]_{\mathrm{value}}(i, f, s) & = & () \\
[\![\mathtt{check}_\pi\ e]\!]_{\mathrm{update}}(i, f, s) & = & [\![e]\!]_{\mathrm{update}}(i, f, s) \\
[\![\mathtt{check}_\pi\ e]\!]_{\mathrm{rely}}(i, f, s) & = & (\pi = \boxed{\checkmark} \implies [\![e]\!]_{\mathrm{value}}(i, f, s)) \wedge [\![e]\!]_{\mathrm{rely}}(i, f, s) \\
[\![\mathtt{check}_\pi\ e]\!]_{\mathrm{guar}}(i, f, s) & = & (\pi = \boxed{?} \implies [\![e]\!]_{\mathrm{value}}(i, f, s)) \wedge [\![e]\!]_{\mathrm{guar}}(i, f, s) \\[1em]
[\![\mathtt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\mathrm{init}} & = & [\![e_r]\!]_{\mathrm{init}} \cup [\![e_g]\!]_{\mathrm{init}} \\
[\![\mathtt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\mathrm{value}}(i, f, s) & = & f.x \\
[\![\mathtt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\mathrm{update}}(i, f, s) & = & [\![e_r]\!]_{\mathrm{update}}(i, f, s) \cup [\![e_g]\!]_{\mathrm{update}}(i, f, s) \\
[\![\mathtt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\mathrm{rely}}(i, f, s) & = & ([\![e_r]\!]_{\mathrm{value}}(i, f, s) \implies [\![e_g]\!]_{\mathrm{value}}(i, f, s)) \\
 & \wedge & (\pi = \boxed{\checkmark} \implies [\![e_r]\!]_{\mathrm{value}}(i, f, s)) \\
 & \wedge & [\![e_r]\!]_{\mathrm{rely}}(i, f, s) \wedge [\![e_g]\!]_{\mathrm{rely}}(i, f, s) \\
[\![\mathtt{contract}_\pi\ \{e_r\}\ e_b\ \{x.\ e_g\}]\!]_{\mathrm{guar}}(i, f, s) & = & (\pi = \boxed{?} \implies [\![e_r]\!]_{\mathrm{value}}(i, f, s)) \\
 & \wedge & [\![e_r]\!]_{\mathrm{guar}}(i, f, s) \wedge [\![e_g]\!]_{\mathrm{guar}}(i, f, s)
\end{array}
$$

Figure 8 Transition system semantics; for an expression $\Gamma \vdash e : \tau$, $[\![e]\!]_{\mathrm{init}} : \text{heap } [\![e]\!]_{\mathrm{state}}$ is the initial state. For each field of the step-result type, we define a translation function that takes the input, free and state heaps: for example, we define the value-result of a step with type $[\![e]\!]_{\mathrm{value}} : \text{heap } \Gamma \to \text{heap } [\![e]\!]_{\mathrm{free}} \to \text{heap } [\![e]\!]_{\mathrm{state}} \to \tau$.

$$\boxed{\Sigma \vdash e \sim s}$$

$$\frac{}{\Sigma \vdash v \sim s}\ (\textsc{IValue}) \qquad \frac{}{\Sigma \vdash x \sim s}\ (\textsc{IVar})$$

$$\frac{\Sigma \vdash e_1 \sim s \quad \ldots \quad \Sigma \vdash e_n \sim s}{\Sigma \vdash p(\overline{e}) \sim s}\ (\textsc{IPrim}) \qquad \frac{s.x_{\texttt{fby}} = v \qquad \cdot \vdash e' \sim s}{\cdot \vdash v \ \texttt{fby}\ e' \sim s}\ (\textsc{IFby}_0)$$

$$\frac{\Sigma; \sigma \vdash e' \Downarrow s.x_{\texttt{fby}} \qquad \Sigma; \sigma \vdash e' \sim s}{\Sigma; \sigma \vdash v \ \texttt{fby}\ e' \sim s}\ (\textsc{IFby}_S)$$

$$\frac{\Sigma \vdash \texttt{rec}\ \ x.\ e \Downarrow^* V \qquad \Sigma[x \mapsto V] \vdash e \sim s}{\Sigma \vdash \texttt{rec}\ \ x.\ e[x] \sim s}\ (\textsc{IRec})$$

$$\frac{\Sigma \vdash e \Downarrow^* V \qquad \Sigma \vdash e \sim s \qquad \Sigma[x \mapsto V] \vdash e' \sim s}{\Sigma \vdash \texttt{let}\ \ x = e\ \ \texttt{in}\ \ e'[x] \sim s}\ (\textsc{ILet})$$

$$\frac{\Sigma \vdash e \sim s}{\Sigma \vdash \texttt{check}_\pi\ e \sim s}\ (\textsc{ICheck})$$

$$\frac{\Sigma \vdash e_{\text{body}} \Downarrow^* V \qquad \Sigma \vdash e_{\text{rely}} \sim s \qquad \Sigma[x \mapsto V] \vdash e_{\text{guar}} \sim s}{\Sigma \vdash \texttt{contract}_\pi\ \{e_{\text{rely}}\}\ e_{\text{body}}\ \{x.\ e_{\text{guar}}[x]\} \sim s}\ (\textsc{IContract})$$

Figure 9 Transition system state invariant

ignore the body. As with recursive expressions, we require an arbitrary value $x : \tau$ in the free heap. The translation's rely allows us to assume that the contract definition holds: that is, the contract's rely implies the contract's guarantee. If the contract instantiation is known to be valid, we can also assume that the contract's rely holds. Otherwise, we include the contract's rely as an obligation by putting it in the translation's guarantee.

In the contract instantiation, we assume that if the contract rely is true *at the current step*, then the contract guarantee also holds at the current step. The true semantics of the contract, however, only holds if the contract rely is true *at every step so far*. This simplification is justified as our definition of validity for a transition system also requires the translation rely to be true at every step. This simplification is essentially an application of the $\Box\Box p \implies \Box p$ axiom of modal logic. The mechanised proof of this simplification is future work.

4.1 Translation correctness proofs

We prove that the transition system is an abstraction of the dynamic semantics: that is, if the expression evaluates to v under some context, then there exists some execution of the transition system that also results in v. The transition system itself is deterministic, but the free context provides the non-determinism; our theorem statement existentially quantifies the free heap.

The results presented here rely heavily on the totality and substitution metaproperties described in Subsection 3.3. Figure 9 defines the invariant for the abstraction proof; the judgment form $\Sigma \vdash e \sim s$ checks that s is a valid state heap. We use the invariant to state that, if executing the transition system for e on the entire streaming history Σ results in state heap s, then s is a valid state.

As most expressions do not modify the state heap, the invariant for most expressions simply descends into the subexpressions. Where new bindings are added, we use the dynamic semantics to extend the context with the new values. The invariant for follow-by expressions asserts that the initial state of the follow-by is the default value; on subsequent steps, the state corresponds to the dynamic semantics.

▶ **Theorem 4** (translation-abstraction). *For a well-typed causal expression e and streaming history Σ, if e evaluates to v ($\Sigma \vdash e \Downarrow v$), then there exists a sequence of free heaps Σ_F such that repeated application of the transition system's step results in v.*

5 Extraction

Pipit can generate executable code which is suitable for real-time execution on embedded devices. The code extraction uses a variation of the abstract transition system described in Section 4, with two main differences to ensure that the result is executable without relying on the environment to provide values for the free context. Contracts are straightforward to execute by using the body of the contract rather than abstracting over the implementation.

To execute recursive expressions $\texttt{rec}\ x.\ e : \tau$, we require an arbitrary value of type τ to seed the fixpoint, as described in Subsection 3.3. We first call the step function to evaluate e with x bound to $\bot_\tau$. This step call returns the correct value, but the updated state is invalid, as it may refer to the bottom value. To get the correct state, we call the step function again, this time with e bound to v.

This translation to transition systems is verified to preserve the original semantics. To extract the program, we use a *hybrid embedding* as described in [23], which is similar to staged-compilation. The hybrid embedding involves a deep embedding of the Pipit core language, while the translation to executable transition systems produces a shallow embedding. We use the F* host language's normalisation-by-evaluation and tactic support [31] to specialise the application of the translation to a particular input program. This specialisation results in a concrete transition system that fits in the Low* [34] subset of F*, which can then be extracted to statically-allocated C code.

The translation for recursive streams described above calls the step function of the sub-stream twice, which can duplicate work. The normalisation strategy used to partially-evaluate the translation inlines the two occurrences of the step function, and is often able to remove the duplicate work, but this removal is not guaranteed. Our current approach is also unsuitable for generating imperative array code, as our shallowly-embedded pure transition system requires pure arrays. In the future, we intend to address array computations and the above work duplication by introducing an intermediate imperative language such as Obc [3], a static object-based language suitable for synchronous systems. Even with an added intermediate language, we believe that a variant of our current translation and proof-of-correctness will remain useful as an intermediate semantics.

6 Evaluation

As a preliminary evaluation of Pipit, we have implemented the high-level logic of a time-triggered Controller Area Network (CAN) bus driver [1]. The CAN bus is commonly found in safety-critical automotive and industrial settings. The time-triggered network architecture defines a static schedule of network traffic. All nodes on the network must adhere to the same schedule, which significantly increases the reliability of periodic messages [15].

At a high level, the schedule is described by a system matrix which consists of rows of basic cycles. Each basic cycle consists of a sequence of actions to be performed at particular time-marks. Actions in the schedule may not be relevant to all nodes, so each node has its own local array containing the relevant triggers; trigger actions include sending and receiving application-specific messages, sending reference messages, and triggering 'watch' alerts. The trigger action for receiving an application-specific message checks that a particular message has been received since the trigger was last executed; depending on this, the driver increments or decrements a message-status-counter, which will in turn signal an error once the upper limit is reached. Reference messages start a new basic cycle and are used to synchronise the nodes. Watch alerts are generally placed after the expected end of the cycle and are used to signal an error if no reference message is received.

The TTCAN protocol can be implemented in two levels of increasing complexity. In the first level, reference messages contain the index of the newly-started cycle. In the second level, the reference messages also contain the value of a global fractional clock and whether any gaps have occurred in the global clock, which allows other nodes to calibrate their own clocks. We implement the first level as it is more amenable to software implementation [22].

The implementation defines a streaming function that takes a stream describing the current time, the state of the hardware, and any received messages. It returns a stream of commands to be performed, such as sending a particular reference message. The implementation defines a pure streaming function. To actually interact with the hardware we assume a small hardware-interop layer that reads from the hardware registers and translates the commands to hardware-register writes, but we have not yet implemented this. We package the driver's inputs into a record for convenience:

```
type driver_input = {
    local_time:  network_time_unit;
    mode_cmd:    option mode;
    tx_status:   tx_status;
    bus_status:  bus_status;
    rx_ref:      option ref_message;
    rx_app:      option app_message_index;
}
```

Here, the local-time field denotes the time-since-boot in *network time units*, which are based on the bitrate of the underlying network bus. The mode-command is an optional field which indicates requests from the application to enter configuration or execution mode. The transmission-status describes the status of the last transmission request and may be none, success, or various error conditions. The bus-status describes whether the bus is currently idle, busy, or in an error state. The two receive fields denote messages received from the bus; for application-specific messages the time-triggered logic only needs the message identifier.

The driver-logic returns a stream of commands for the hardware-interop layer to perform:

```
type commands = {
    enable_acks:  bool;
    tx_ref:       option ref_message;
    tx_app:       option app_message_index;
    tx_delay:     network_time_unit;
}
```

The enable-acks field denotes whether the hardware should respond to messages from other nodes with an acknowledgement bit; in the case of a severe error acknowledgements are disabled, as the node must not write to the bus at all. The transmit fields denote whether to send a reference message or an application-specific message. For application-specific messages, the hardware-interop layer maintains the transmission buffers containing the actual message payload. To meet the schedule as closely as possible, the driver anticipates the next transmission and includes a transmission delay to tell the hardware exactly when to send the next message.

6.1 Runtime

The implementation includes an extension of the trigger-fetch logic described in Section 2, as well as state machines for tracking node synchronisation, master status and fault handling. We generate real-time C code as described in Section 5. We evaluated the generated C code by executing with randomised inputs and measuring the worst-case-execution-time on a Raspberry Pi Pico (RP2040) microcontroller. The runtime of the driver logic is fairly stable: over 5,000 executions, the measured worst-case execution time was $114\mu s$, while the average was $107\mu s$ with a standard deviation of $2.3\mu s$. Earlier work on fault-tolerant TTCAN [41] describes the required slot sizes — the minimum time between triggers — to achieve bus utilisation at different bus rates. For a 125Kbit/s bus, a slot size of approximately $1{,}500\mu s$ is required to achieve utilisation above 85 per cent. For the maximum CAN bus rate of 1Mbit/s, the required slot size is $184\mu s$. Further evaluation is required to ensure that the complete runtime including the hardware-interop layer is sufficient for full-speed CAN.

Our code generation can be improved in a few ways. A common optimisation in Lustre is to fuse consecutive if-statements with the same condition [5]; such an optimisation seems useful here, as our treatment of optional values introduces repeated unpacking and repacking. Some form of array fusion [37] may also be useful for removing redundant array operations. Our current extraction generates a transition-system with a step function which returns a tuple of the updated state and result. Composing these step functions together results in repeated boxing and unboxing of this tuple; we currently rely on the $F^\star$ normaliser to remove this boxing. In the future, we plan to build on the current proofs to implement a more-sophisticated encoding that introduces less overhead.

6.2 Verification

We have verified a simplified trigger-fetch mechanism, as presented earlier (Section 2). For comparison, we implemented the same logic in the Kind2 model-checker [11]. The restrictions placed on the triggers array — that triggers are sorted by time-mark, that there must be an adequate time-gap between a trigger and its next-enabled, and that a trigger's time-mark must be greater-than-or-equal-to its index — are naturally expressed with quantifiers. The Kind2 model-checker includes experimental array and quantifier support [26]. Due to the experimental nature of these features, we had to work around some limitations: for example, the use of arrays and quantifiers disables IC3-based invariant generation; quantified variables cannot be used in function calls; and the use of top-level constant arrays caused runtime errors that rendered most properties invalid [27].

We were able to verify the Kind2 implementation of the simplified trigger-fetch mechanism for trigger arrays containing up to 16 elements; above that, a 32-size array reported multiple runtime errors and did not terminate after several hours. For reference, the M_TTCAN hardware implementation of TTCAN supports up to 64 triggers [36].

	Kind2				Pipit	
	simple enable-set		full enable-set			
size	wall-clock	CPU	wall-clock	CPU	wall-clock	CPU
1	2s	3s	6s	12s	6s	6s
2	3s	3s	8s	12s	6s	6s
4	5s	11s	12s	24s	6s	6s
8	8s	13s	82s	90s	6s	6s
16	125s	276s	error		6s	6s
32	error		error		6s	6s

Figure 10 Verification time for trigger-fetch; simple enable-set uses a simplified version of the enable-set, while full enable-set uses bitwise arithmetic as in the TTCAN specification.

We made a critical simplification in the Kind2 implementation, which was to modify the trigger-enabled set to be a single cycle index. In the specification, the enabled set is implemented as a cycle-offset and repeat-factor. Checking if a trigger is enabled in the current cycle requires nonlinear arithmetic, which is difficult for SMT solvers. In our Pipit development, we can treat the definition of the cycle set abstractly. However, in the Kind2 development, quantifiers cannot contain function calls, which means that we cannot hide the implementation of the enabled-set check by providing an abstract contract. This limitation also makes the specification quite unwieldy, as functions must be manually inlined.

Figure 10 shows the verification runtime for different sizes of arrays; the Pipit version is parametric in the array size, and is thus verified for all sizes of arrays. We ran these experiments on a 2020 M1 MacBook Air with 16 gigabytes of RAM. Both Kind2 and Pipit developments of the simplified trigger-fetch logic are roughly the same size, on the order of two-hundred lines of code including comments.

We plan to verify the remainder of the TTCAN implementation and publish it separately. Prior work formalising TTCAN has variously modeled the protocol itself [39, 33, 30], instances of the protocol [20], and abstract models of TTCAN implementations [29], but we are unaware of any prior work that has verified an *executable* implementation of TTCAN.

Separately, Pipit has also been used to implement and verify a real-time controller for a coffee machine reservoir control system [38]. The reservoir has a float switch to sense the water level and a solenoid to allow the intake of water. The specification includes a simple model of the water reservoir and shows that the reservoir does not exceed the maximum level under different failure-mode assumptions.

7 Related work

Using existing Lustre tools to verify *and* execute the time-triggered CAN driver from Section 2 is nontrivial. Compiling the triggers array with an unverified compiler such as Lustre V6 [24] or Heptagon [19] is straightforward; however, the verified Lustre compiler Vélus [7] does not support arrays or a foreign-function interface. Recent work on translation validation for LustreC [9] also does not yet support arrays.

Verifying the time-triggered CAN driver is trickier, as the restrictions placed on the triggers array — that triggers are sorted by time-mark, there must be an adequate time-gap between a trigger and its next-enabled, and a trigger's time-mark must be greater-than-or-equal-to its index — naturally require quantifiers. As described in Section 6, the Kind2 does include experimental array and quantifier support, but is limited to verifying small arrays up to 8 or 16 triggers. Additionally, due to the limitations on top-level array definitions, compiling the program with Lustre V6 would result in multiple copies of the entire triggers array on the stack.

Other model-checkers for Lustre such as Lesar [35], JKind [16] and the original Kind [21] do not support quantifiers. It may be possible to encode the quantifiers as fixed-size loops, but ensuring that these loops do not affect the execution or runtime complexity of the generated code does not appear to be straightforward.

These model-checkers have definite usability advantages over the general-purpose-prover approach offered here: they can often generate concrete counterexamples and implement counterexample-based invariant-generation techniques such as ICE [18] and PDR [8, 14]. However, even when the problem can be expressed, these model-checkers do not provide much assurance that the semantics they use for proofs matches the compiled code. In the future, we would like to investigate integrating Pipit with a model-checker via an unverified extraction: such an extraction may allow some of the usability benefits such as counterexamples and invariant generation. If this integration were used solely for debugging and suggesting candidate invariants, then such a change would not expand the trusted computing base.

Recent work has also introduced a form of refinement types for Lustre [12]. Rather than using transition systems, this work generates self-contained verification conditions based on the types of streams. Such a type-based approach promises to allow abstraction of the implementation details. However, for general-purpose functions such as *count_when* from Section 2, it is not clear how to give it a specification that actually *abstracts* the implementation: a simple specification that the result is within some range would hide be insufficient for verifying the rest of the system. For this function, the best specification is likely to include a re-statement of the implementation itself.

The embedded language Copilot generates real-time C code for runtime monitoring [28]. Recent work has used translation validation to show that the generated C code matches the high-level semantics [40]. Copilot supports model-checking via Kind2; however, the model-checking has a limited specification language and does not support contracts.

Early work embedding a denotational semantics of Lucid Synchrone in an interactive theorem prover focussed on the semantics itself, rather than proving programs [4]. There is ongoing work to construct a denotational semantics of Vélus for program verification [6]. We believe that the hybrid SMT approach of F* will allow for a better mixture of automated proofs with manual proofs. Compared to Vélus alone, the trusted computing base of Pipit is larger: we depend on all of F*, Low*'s C code extraction and the Z3 SMT solver.

The deferred aspect of our proofs is similar to the deferred proofs of verification conditions for imperative programs, such as [32]. However, such verification conditions are *syntactically* deferred so that the verification condition can be proved later; in our case, the verification conditions are *semantically* deferred, so that more knowledge of the enclosing program can be exploited in the proof. In imperative programs, this sort of extra knowledge is generally provided explicitly as loop invariants, and non-looping statements have their weakest precondition computed automatically. In Lustre-style reactive languages such as ours, programs tend to be composed of many nested recursive streams, which perform a similar function to loops. Explicitly specifying an invariant for each recursive stream would be cumbersome; deferring the proof allows such invariants to be implicit.

8 Conclusion

TODO: *requires rewrite* ***TODO:*** *future work: clocks*

Our preliminary results show that F*'s proof automation and code extraction are suitable for verifying reactive systems and executing them in real-time; these results still require further work. Next, we intend to verify the imperative code generation. Finally, we need to evaluate Pipit on larger control systems before extending the language to support more features, such as Lustre's clocks for describing partially-defined streams [10].

We are interested in further pursuing the intersection of model-checking with interactive theorem proving. A smart contract called Djed [42] currently uses a mixture of Kind2 [11] and manual Isabelle/HOL proofs to show that the contract is well-behaved. In future work, we would like to further investigate whether Pipit's integration of streaming proofs with F*'s automated proof system would be able to provide similar proofs, without introducing any semantic gap between the two systems.

Our current array support is limited: constant arrays provide a pure index function, while we only support fixed-size mutable arrays by wrapping bit-vectors. Array support is an obvious direction for future work; integrating with a verified array-fusion system such as [37] would be an interesting and useful extension.

References

1 ISO/CD 11898-4. Road vehicles - Controller area network (CAN) - Part 4: Time triggered communication. Standard, International Organization for Standardization, 2000.

2 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

3 Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 121–130, 2008.

4 Sylvain Boulmé and Grégoire Hamon. A clocked denotational semantics for Lucid-Synchrone in Coq. *Rap. tech., LIP6*, 2001.

5 Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

6 Timothy Bourke, Paul Jeanmaire, and Marc Pouzet. Towards a denotational semantics of streams for a verified Lustre compiler. 2022. URL: `https://types22.inria.fr/files/2022/06/TYPES_2022_slides_28.pdf`.

7 Timothy Bourke, Basile Pesin, and Marc Pouzet. Verified compilation of synchronous dataflow with state machines. *ACM Transactions on Embedded Computing Systems*, 22(5s):1–26, 2023.

8 Aaron R Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings 12*. Springer, 2011.

9 Lélio Brun, Christophe Garion, Pierre-Loïc Garoche, and Xavier Thirioux. Equation-directed axiomatization of lustre semantics to enable optimized code validation. *ACM Transactions on Embedded Computing Systems*, 22(5s):1–24, 2023.

10 Paul Caspi and Marc Pouzet. A functional extension to Lustre. *Intensional Programming I*, 1995.

11 Adrian Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The Kind 2 model checker. In *Computer Aided Verification*, 2016.

12 Jiawei Chen, José Luiz Vargas de Mendonça, Shayan Jalili, Bereket Ayele, Bereket Ngussie Bekele, Zhemin Qu, Pranjal Sharma, Tigist Shiferaw, Yicheng Zhang, and Jean-Baptiste Jeannin. Synchronous programming and refinement types in robotics: From verification to implementation. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems*, 2022.

13 Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6: A formal language for embedded critical software development. In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–11. IEEE, 2017.

14 Niklas Eén, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2011.

15 Thomas Fuehrer, Bernd Mueller, Florian Hartwich, and Robert Hugel. Time triggered CAN (TTCAN). *SAE transactions*, pages 143–149, 2001.

16 Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani. The JKind model checker. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*, pages 20–27. Springer, 2018.

17 Emilio Jesús Gallego Arias, Pierre Jouvelot, Sylvain Ribstein, and Dorian Desblancs. The W-calculus: a synchronous framework for the verified modelling of digital signal processing algorithms. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, pages 35–46, 2021.

18 Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 2014.

19 Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler. *ACM SIGPLAN Notices*, 47(5), 2012.

20 Xiaoyun Guo, Toshiaki Aoki, and Hsin-Hung Lin. Model checking of in-vehicle networking systems with CAN and FlexRay. *Journal of Systems and Software*, 161:110461, 2020.

21 George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *2008 Formal Methods in Computer-Aided Design*. IEEE, 2008.

22 Florian Hartwich, Thomas Führer, Bernd Müller, and Robert Hugel. Integration of time triggered CAN (TTCAN_TC). *SAE Transactions*, pages 112–119, 2002.

23 Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 107–124. IEEE, 2022.

24 Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. The Lustre V6 reference manual. *Verimag, Grenoble, Dec*, 2016.

25 Kind2. Integer division rounds to negative infinite. Github issues, 2023. URL: `https://github.com/kind2-mc/kind2/issues/978`.

26 Kind2. *Kind2 user documentation*, 2.1.1 edition, 2023. URL: `https://kind.cs.uiowa.edu/kind2_user_doc/doc.pdf`.

27 Kind2. Top-level array definition causes runtime failures. Github issues, 2024. URL: `https://github.com/kind2-mc/kind2/issues/1043`.

28 Jonathan Laurent, Alwyn Goodloe, and Lee Pike. Assuring the guardians. In *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*. Springer, 2015.

29 Gabriel Leen and Donal Heffernan. Modeling and verification of a time-triggered networking protocol. In *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICNICONSMCL'06)*, pages 178–178. IEEE, 2006.

30 Xin Li, Jian Guo, Yongxin Zhao, and Xiaoran Zhu. Formal modeling and verifying the TTCAN protocol from a probabilistic perspective. *Journal of Circuits, Systems and Computers*, 28(10):1950177, 2018.

31 Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan

Protzenko, et al. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *Programming Languages and Systems: 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings*. Springer International Publishing Cham, 2019.

32 Liam O'Connor. Deferring the details and deriving programs. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development*, pages 27–39, 2019.

33 Can Pan, Jian Guo, Longfei Zhu, Jianqi Shi, Huibiao Zhu, and Xinyun Zhou. Modeling and verification of CAN bus with application layer using UPPAAL. *Electronic Notes in Theoretical Computer Science*, 309:31–49, 2014.

34 Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, et al. Verified low-level programming embedded in F*. *Proc. ACM program. lang.*, 1(ICFP), 2017.

35 Pascal Raymond. Synchronous program verification with Lustre/Lesar. *Modeling and Verification of Real-Time Systems*, 2008.

36 Robert Bosch GmbH. *M_TTCAN Time-triggered Controller Area Network User's Manual*, 3.3.0 edition, 2019. URL: `https://www.bosch-semiconductors.com/media/ip_modules/pdf_2/m_can/mttcan_users_manual_v330.pdf`.

37 Amos Robinson and Ben Lippmeier. Machine fusion: merging merges, more or less. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, pages 139–150, 2017.

38 Amos Robinson and Alex Potanin. Pipit: Reactive systems in F*. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development*, 2023.

39 Indranil Saha and Suman Roy. A finite state analysis of time-triggered CAN (TTCAN) protocol using Spin. In *2007 International Conference on Computing: Theory and Applications (ICCTA'07)*, pages 77–81. IEEE, 2007.

40 Ryan G Scott, Mike Dodds, Ivan Perez, Alwyn E Goodloe, and Robert Dockins. Trustworthy runtime verification via bisimulation (experience report). *Proceedings of the ACM on Programming Languages*, 7(ICFP):305–321, 2023.

41 Michael Short and Michael J Pont. Fault-tolerant time-triggered communication using CAN. *IEEE transactions on Industrial Informatics*, 3(2):131–142, 2007.

42 Joachim Zahnentferner, Dmytro Kaidalov, Jean-Frédéric Etienne, and Javier Díaz. Djed: a formally verified crypto-backed autonomous stablecoin protocol. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2023.