Pipit: Reactive Systems in F* (Extended Abstract)

Amos Robinson
amos.robinson@anu.edu.au
Australian National University
Canberra, Australia

Alex Potanin
alex.potanin@anu.edu.au
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 imperative code in a subset of F* which is suitable for compilation and real-time execution on embedded devices. This translation to imperative code preserves types by construction; the proof that the imperative code preserves semantics is ongoing.

CCS Concepts: • **Computer systems organization** → *Embedded software*; **Real-time languages**; • **Theory of computation** → **Program verification**; Modal and temporal logics; • **Software and its engineering** → **Specialized application languages**.

Keywords: Lustre, streaming, reactive, control

ACM Reference Format:
Amos Robinson and Alex Potanin. 2023. Pipit: Reactive Systems in F* (Extended Abstract). In *Proceedings of In submission (TyDe '23)*. ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

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 [6] or Scade [10], and verify that the implementations satisfy the high-level specification using a model-checker, such as Kind2 [7]. These model-checkers can prove many interesting 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 [3]. 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 [1, 16].

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 abstract 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 [19], and using Low*'s C-code generation for real-time execution [20]. Pipit translates its expression language to a transition system for k-inductive proofs, which is verified; verifying the translation to imperative code is ongoing.

In this extended abstract we briefly describe the following preliminary results, which we intend to describe fully in a future publication:

- we motivate Pipit which, as an embedded language, provides syntactic convenience with a small verifiable core language (section 2);
- we demonstrate the use of F*'s existing normalisation and proof automation to prove certain properties with minimal effort (subsection 2.1);
- we describe a key difference between Pipit's core language and Lustre (section 3); and
- we evaluate Pipit by executing a verified controller on an embedded system (section 5).

2 Programming and verifying in Pipit

A common requirement in controllers is to filter an input signal, perhaps using a *finite impulse response* (FIR) filter, which is equivalent to a weighted moving average. An FIR filter takes a vector of coefficients and an input signal; at every point in time, it computes the dot product of the coefficients

TyDe '23, 2023, In submission

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/XXXXXXX.XXXXXXX

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

and the most recent values of the signal. We can implement an FIR filter in Pipit as follows:

```
let fir (coefficients: list ℝ) (signal: stream ℝ): stream ℝ =
  match coefficients with
  | []       → 0
  | c :: cs → (signal · c) + (0 fby (fir cs signal))
```

The coefficient vector is represented by a list of reals, while the signal is a *stream* of reals, and the result is the filtered stream. The implementation starts by looking at the list of coefficients and returns zero if the list is empty. If the list is not empty, then we multiply the most recent value of the stream by the coefficient ($signal \cdot c$); we also take the result of applying the remaining coefficients to the signal stream (fir *cs signal*) and delay it (0 fby ...) before summing the two parts. At the start of execution the delay is initially zero.

Pipit is an *embedded* language, like Bedrock [9]: in this example, the stream type denotes an actual Pipit expression, while the list type and its associated pattern match are part of the F* meta-language. To get the real Pipit program, we need to apply the filter to a concrete list of coefficients:

```
let fir_2 (input: stream ℝ): stream ℝ =
  fir [0.7; 0.3] input
```

Normalising this definition evaluates away all of the lists. The result fits in the core language defined in section 3, for which we can generate real-time imperative code:

```
let fir_2 (input: stream ℝ): stream ℝ =
  (0.7 · input) + (0 fby ((0.3 · input) + (0 fby 0)))
```

The properties that we want to state about reactive programs usually involve some temporal aspect. Rather than defining a separate specification language, we implement computable variants of temporal operators from *past-time* linear temporal logic [14, 18]. We name the past-globally operator *sofar*, as in *the predicate has been true so far*:

```
let sofar (p: stream 𝔹): stream 𝔹 =
  rec p′. p ∧ (true fby p′)
```

This definition takes a stream of predicates p and introduces a recursive stream p'. At each step, the recursive stream p' checks that the current predicate is true (p), and also checks that *sofar* was previously true (true fby p'). If there is no previous value, it defaults to true.

2.1 Bounded input, bounded output

We can now state a *bounded-input-bounded-output* (BIBO) property, which says that if the inputs have always been within some particular range, then the outputs are also within the range:

```
let bibo_2 (n: ℝ_≥0) (input: stream ℝ): stream 𝔹 =
  check (sofar(|input| ≤ n)  ⟹  |fir_2 input| ≤ n)
```

This property states that if the input has always been in the range $[-n, n]$, then the output is also within the range $[-n, n]$. Note that the upper bound n is a nonnegative real rather than a stream of reals, which means that n stays constant across the whole stream. To prove that this property holds, we translate to a transition system and show that the stream is always true. In this case, induction over the transition relation is sufficient to prove the property. There is some boilerplate required to perform the induction, but both base and step cases are automatically proved by F*:

```
let proof_2 (n: ℝ_≥0): Lemma (induct (bibo_2 n)) =
  assert (base_case (bibo_2 n)) by (pipit_simplify ());
  assert (step_case (bibo_2 n)) by (pipit_simplify ())
```

This definition uses F*'s *lemma* syntax to state that the BIBO property holds inductively for any n. The two assertions prove the inductive cases separately, using our *simplify* tactic to ensure that the translation to transition system is normalised away, and any translation artefacts are removed.

If we wish to prove a similar BIBO property for a filter with more coefficients, standard induction over the transition system is not sufficient: the relationship between the stacked delays in the filter and sofar is not clear from a single step of the transition system. One simple automated way to strengthen invariants is via *k-induction* [13], which adds more context by assuming that the property holds for k previous steps of the transition relation. We can define analogous functions fir$_3$ and bibo$_3$ which operate on the coefficients $[0.7; 0.2; 0.1]$, and use k-induction for $k = 2$ as follows:

```
let proof_3 (n: ℝ_≥0): Lemma (induct_k 2 (bibo_3 n)) =
  assert (base_case_k 2 (bibo_3 n))  by (pipit_simplify ());
  assert (step_case_k 2 (bibo_3 n))  by (pipit_simplify ())
```

Although the properties here boil down to simple properties about linear arithmetic, we believe that this example demonstrates a promising way to use F*'s existing proof automation to verify reactive systems.

$$
\begin{array}{lcl}
e, e' & := & v \mid x \mid e\ e' \\
 & \mid & v \texttt{ fby } e \quad \mid \quad e \rightarrow e' \\
 & \mid & \texttt{rec } x.\ e[x] \quad \mid \quad \texttt{check } e \\
 & \mid & \texttt{let } x = e \texttt{ in } e'[x] \\
 & & \\
v & := & n \in \mathbb{N} \mid b \in \mathbb{B} \mid r \in \mathbb{R} \mid \ldots \mid \lambda x.\, e
\end{array}
$$

Figure 1. Pipit core language grammar

3 Core language

The grammar of Pipit is defined in Figure 1. The expression form e includes standard syntax for values (v), variables (x) and applications ($e\ e'$); however, it does not include any form for defining functions except reusing closed functions from the F* meta-language ($\lambda x.\, e$). Most of the expression forms were introduced informally in section 2 and correspond to the clock-free primitives in Lustre [6]. Streams can also be composed together using the *then* notation ($e \rightarrow e'$) which denotes that the value of stream e is used for the first step, followed by the values from stream e' for subsequent steps.

Recursive streams, which can refer to previous values of the stream itself, are defined using the fixpoint operator ($\texttt{rec}\ x.\ e[x]$); 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 ($\texttt{rec}\ x.\ 0\ \texttt{fby}\ (x + 1)$) is well-defined, but stream ($\texttt{rec}\ x.\ x + 1$) is invalid and has no computational interpretation. This form of recursion differs slightly from standard Lustre, which uses a set of mutually-recursive bindings. We use this form to define a substitution-based operational semantics that is syntax-directed, as opposed to the mutually-recursive form in Caspi and Pouzet [6] which is not syntax-directed. Our semantics has a simpler proof of determinism; we believe it has simplified other necessary proofs too and will perform further evaluation. Although we cannot express mutually-recursive bindings in the core syntax here, we can express them as a notation on the surface syntax at the expense of potentially duplicating expressions.

4 Extraction

Pipit can generate executable code which is suitable for real-time execution on embedded devices. The code extraction is implemented in a currently-unverified transform that takes a deeply embedded representation of a Pipit expression and generates a shallow imperative representation of the program. F* can generate C code from a subset of the language called Low* [20]; the result of our translation to imperative code fits in this subset. During code extraction, we use F*'s tactic support [19] to fully normalise the translation to imperative code, conceptually similar to staged compilation.

5 Evaluation

To demonstrate the feasibility of Pipit, we have implemented and verified a simple controller. This system controls a water-flow solenoid to fill the reservoir of a coffee machine and includes multiple safeguards to reduce the risk of flooding. The controller has two boolean inputs: the *stop* switch and the *low level* indicator; it returns a boolean indicating whether to engage the solenoid. The stop switch indicates whether the reservoir's lid is open or closed; the system should never operate while the lid is open as water could spill out. The controller should not allow water to flow for more than a minute as this may indicate a leak; if so, the controller enters a terminal error state. Finally, to avoid switching the solenoid too often, the controller waits for ten seconds of low water level before trying to engage:

```
let reservoir (stop low: stream 𝔹): stream 𝔹 =
  let try    = true_for TEN_SECONDS (not stop ∧ low) in
  let error  = any (true_for ONE_MINUTE try) in
  let engage = try ∧ not error in
  check (engage ⟹ not stop);
  check (engage ⟹ low);
  engage
```

Predicate *true_for t* is true if a signal has been true for time *t*; *any* is true if a signal has ever been true. As with the previous examples, the two properties can be automatically verified. Pipit generates real-time C code for this example[2].

6 Related work

Although the FIR filter from section 2 is quite simple, verifying it *and* executing it with existing Lustre tools is nontrivial. Lustre itself does not support lists, as dynamically-allocated data structures are not well-suited to real-time execution. To write this filter in Lustre we would either need to unroll the lists ourselves or reformulate the program to use arrays. However, Vélus does not support arrays [3]; Kind2 uses a custom syntax for arrays with no compiler support [7]; and the Lustre V6 compiler does support arrays [15], but its model-checker Lesar cannot reason about integers or reals [21].

In terms of model-checking reactive systems, recent work uses SMT solvers to check inductive proofs [7, 13] or to check refinement types [8]. These model-checkers have definite 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 [12] and PDR [5, 11]. However, these model-checkers do not provide much assurance that the semantics they use for proofs matches the compiled code. We believe that once Pipit's imperative code generation is verified, Pipit will have a stronger assurance case.

The embedded language Copilot generates real-time C code for runtime monitoring and supports model-checking properties [17], but suffers from the same semantic gap.

Early work embedding a denotational semantics of Lucid Synchrone in an interactive theorem prover focussed on the semantics itself, rather than proving programs [2]. There is ongoing work to construct a denotational semantics of Vélus for program verification [4]. We believe that the hybrid SMT approach of F* will allow for a better mixture of automated proofs with manual proofs; however, the trusted computing base of Pipit is much larger than Vélus, as we depend on all of F*, Low*'s C code extraction, the SMT solver, as well as our currently-unverified imperative code generator.

7 Conclusion

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. To verify large programs, we also need some way to separately prove smaller pieces which can then be composed together, such as contracts [7]. 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 [6].

[2] For a video of the controller in action, see https://youtu.be/6IybbQFPOl8

References

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

[2] Sylvain Boulmé and Grégoire Hamon. 2001. A clocked denotational semantics for Lucid-Synchrone in Coq. *Rap. tech., LIP6* (2001).

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

[4] Timothy Bourke, Paul Jeanmaire, and Marc Pouzet. 2022. Towards a denotational semantics of streams for a verified Lustre compiler. *TYPES22.* https://types22.inria.fr/files/2022/06/TYPES_2022_slides_28.pdf

[5] Aaron R Bradley. 2011. 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.

[6] Paul Caspi and Marc Pouzet. 1995. A functional extension to Lustre. *Intensional Programming I* (1995).

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

[8] 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. 2022. 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.*

[9] Adam Chlipala. 2013. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming.*

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

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

[12] Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. 2014. 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.

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

[14] Nicolas Halbwachs, Jean-Claude Fernandez, and A Bouajjanni. 1993. An executable temporal logic to express safety properties and its connection with the language Lustre. In *Sixth International Symp. on Lucid and Intensional Programming.*

[15] Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. 2016. The Lustre V6 reference manual. *Verimag, Grenoble, Dec* (2016).

[16] Kind2. 2023. Integer division rounds to negative infinite. Github issues. https://github.com/kind2-mc/kind2/issues/978

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

[18] Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. 1985. *The glory of the past.* Springer.

[19] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, et al. 2019. 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.

[20] 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. 2017. Verified low-level programming embedded in F*. *Proc. ACM program. lang.* 1, ICFP (2017).

[21] Pascal Raymond. 2008. Synchronous program verification with Lustre/Lesar. *Modeling and Verification of Real-Time Systems* (2008).

Received 1 June 2023