Thorium: A Language for Bounded Verification of Dynamic Reactive Objects

Kevin Baldor
kevin.baldor@utsa.edu
University of Texas at San Antonio
San Antonio, USA
Southwest Research Institute
San Antonio, USA

Xiaoyin Wang
xiaoyin.wang@utsa.edu
University of Texas at San Antonio
San Antonio, USA

Jianwei Niu
jianwei.niu@utsa.edu
University of Texas at San Antonio
San Antonio, USA

Abstract

Developing reliable reactive software is notoriously difficult – particularly when that software reacts by changing its behavior. Some of this difficulty is inherent; software that must respond to external events as they arrive tends to end up in states that are dependent on the value of that input and its order of arrival. This results in complicated corner cases that can be challenging to recognize. However, we find that some of the complexity is an accident of the features of the programming languages widely used in industry. The loops and subroutines of structured programming are well-suited to data transformation, but poorly capture – and sometimes obscure – the flow of data through reactive programs developed using the inversion-of-control paradigm; an event handler that modifies the data flow tends to be declared closer to the definition of the event that activates it than to the initial definition of the data flow that it modifies. This paper approaches both challenges with a language inspired by the declarative modules of languages SIGNAL and Lustre and the semantics of the SodiumFRP Functional Reactive Programming library with a declarative mechanism for self modification through module substitution. These language features lead to software with a code structure that closely matches the flow of data through the running program and thus makes software easier to understand. Further, we demonstrate how those language features enable a bounded model checking approach that can verify that a reactor meets its requirements or present a *counterexample trace*, a series of states and inputs that lead to a violation. We analyze the runtime performance of the verifier as a function of model size and trace length.

CCS Concepts: • **Software and its engineering** → *Data flow languages.*

REBLS '23, October 23, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0400-0/23/10.
https://doi.org/10.1145/3623506.3623574

Keywords: Bounded Model Checking, Functional Reactive Programming

ACM Reference Format:
Kevin Baldor, Xiaoyin Wang, and Jianwei Niu. 2023. Thorium: A Language for Bounded Verification of Dynamic Reactive Objects. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '23), October 23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3623506.3623574

1 Introduction

Whether in an embedded system or a graphical user interface, we expect reactive software to be responsive and *just work*. Embedded software is generally hidden from the user within a hardware device and that further encourages the expectation that it will work like hardware. For some applications, this isn't unreasonable; the application either performs the same operation indefinitely or can be described as a finite state machine. For those applications, software languages like SIGNAL[22], Lustre[6], and Esterel[4] describe software with declarative syntax that resembles that of hardware description languages like VHDL and Verilog and enables static analysis like that available for hardware. For safety-critical applications, the limitations of such systems are worthwhile for the reliability that they can guarantee, but we are interested in bringing some of the advantages of such languages to applications for which absolute reliability can be sacrificed in exchange for flexibility.

This paper presents a declarative language for describing reactive software in terms of *reactors* that contain state and define how that state will change in response to external input. The key innovation is that the elements of a reactor's state can themselves be reactors and that those sub-reactors can be created and destroyed during the execution of the program. We believe that limiting reconfiguration to swapping out reactors encourages the developer to separate the behavior from the logic that is used to select the desired behavior. We selected its features to provide a minimum viable product for a self-modifying reactive system – desirable features such as containers are left to future research – to evaluate the feasibility of bounded model checking such a system.

The paper is organized as follows: sections 2 and 3 describe the prior work on functional reactive languages and frameworks and describes the technologies from which we synthesized the thorium language; section 4 presents an overview of the language use cases and the model-checking output; section 5 formally defines the syntax and semantics; section 6 describes the process of encoding the semantics of the thorium operators in the Python version of the Z3 Satisfiability Modulo Theories (SMT) library; and section 7 evaluates the runtime performance on a set of simple self-modifying reactors.

2 Background

The Thorium language is a synthesis of existing technologies. The semantics of the language are strongly influenced by a polyglot implementation of Functional Reactive Programming (FRP) called Sodium; we rely on off-the-shelf SMT solvers to perform the actual verification; and the bounded verification is strongly influenced by Alloy[20].

2.1 FRP

Functional Reactive Programming was first introduced in Conal Elliot's FRAN[12]. The immediate benefit was that it provided a mechanism for describing time-varying values in the immutable Haskell language declaratively as a function of a value over time. It is that declarative nature and emphasis of side-effect-free operations that enables our model checking approach.

Over the years, a number of alternate implementations of FRP [12],[27], [11] – and FRP inspired reactive systems [25] – have explored different semantics. We have based our languge features on those of the multi-language SodiumFRP [5] family of libraries because it provides semantics that made reasoning about the behavior of the program with a push-based system that fits well with the industry-dominant, procedural, languages that we wish to target.

SodiumFRP relies on two fundamental concepts: *cells*, which hold a value at all points in time, and *streams* which represent a stream of events and hold a value only at times when the event is taking place. Cells take their name and behavior from the cells of a spreadsheet. A spreadsheet cell can be defined as a function of a number of other cells and whenever any of those cells changes, that cell will automatically change as well. Streams do not fit as well into the spreadsheet metaphor, but they could describe the mouse clicks and keyboard input that initiates changes to the values of the cells.

The primary feature that drew us to the SodiumFRP's semantics is the concept of the transaction. All responses to a set of simultaneous inputs can be thought of as happening synchronously (i.e. instantaneously) from the perspective of the rest of the program. Thus, the execution of the program can be seen as a series of transactions updating all of the reactive values in a reactor simultaneously.

2.2 Bounded Model Checking

The reactors that this investigation supports, as limited as they are, still define infinite-state machines, so traditional model checking procedures that prove properties on finite-state machines cannot be directly applied. However, if we place a limit on the length of the trace (series of transactions) that we will consider, the problem then becomes one of satisfiability: the verification of a given property becomes a matter of searching for an assignment of values to the members of the reactor in each transaction that contradicts the property.

This approach to bounded model checking brings a significant limitation, it can only detect violations of properties that can be contradicted by a finite trace. These are the *safety* (i.e. bad things won't happen) properties of [1] and omits the *liveness* (i.e. good things will continue to happen indefinitely) properties that can be important features of a high-reliability system. In defense of our approach, however, this is also true of any testing and our bounded verification is equivalent to testing against *all* input sets up to the length of the bound.

2.3 Satisfiability Modulo Theories

In the previous section, we claimed that our approach to bounding the problem of verification reduced it to satisfiability. Further, if all of the variables in a reactor were Boolean-valued then it would reduce to Boolean satisfiability and – though the problem is NP-Complete in general – often solved reasonably quickly by the DPLL [9] algorithm of a standard Satisfiability (SAT) solver. But, by employing a SMT solver, we can support a richer set of variable types, such as integers and reals.

Like a SAT solver, an SMT solver answers a yes-or-no question: can a given statement be made to be true by assigning values to its free variables? Where a SAT solver could determine that the statement

$$(a \vee \neg b) \wedge (b \vee c) \wedge (a \vee c)$$

can be satisfied for the truth assignments

a	b	c
true		true
	false	true
true	true	

An SMT solver could be presented with a similar statement

$$\begin{aligned} &((x > 15) \vee \neg(x > 10)) \wedge \\ &((x > 10) \vee (x < 5)) \wedge \\ &((x > 15) \vee (x < 5)) \end{aligned} \tag{1}$$

that is the same logical statement, but with a substituted with $x > 15$ and so on. That is, it remains a satisfiability problem, but one that is further restricted by the theory of integer arithmetic. Not all of the solutions identified by the

SAT solver are actually acceptable once a value of x must be found that satisfies the required truth assignments. In particular, only the latter two assignments are possible.

3 Related Work

Our research is primarily inspired by our perceived difficulties in working with existing reactive frameworks in imperative languages. Since our approach is an extension language that enables verification, there are a number of related languages with comparable goals with which we have less familiarity. We believe that our novelty derives from a combination of the semantics and the support for runtime modification.

3.1 Languages and Libraries for Reactive Programming

Many formalisms, libraries, and languages have been proposed to address the complexities of reactive programs. Harel Statecharts[16] are a turing-complete extension to finite state machines and the Actor model [17] captures the nondeterminism of distributed computing by limiting coordination between actors to message passing with no defined order of arrival or processing. Both statecharts and the actor model lend themselves to direct implementation and many such implementations exist. Statecharts underly the proprietary CREATE [18] library (formerly Yakindu [19]) for C, C++, Java, and Python. Popular actor model implementations include Scala Actors [15], Akka [23], and the signals and slots message passing architecture of the Qt framework [7] tends to result in applications that follow the actor-model. However, it was our exposure to large Qt applications that led to our focus on FRP rather than the actor model. The ability to re-wire the signals and slots from arbitrary points in the application can quickly lead to spaghetti signals. We also objected to its lack of glitch-freedom [11]. That is, if actor A sends a single signal to actors B and C simultaneously and B and C each react by sending a signal to D, D can display a temporarily-inconsistent state. The glitch-freedom provided by FRP-inspired frameworks like Sodium-FRP [5], REScala [27] and especially Distributed REScala [11] inspired our focus on the FRP semantics. That said, the parallel, and similarly-motivated research by Lohstroh et al[24] also claims glitch-freedom in a framework inspired by actor-model semantics.

Ultimately, our primary syntactical inspiration has come from the early reactive languages like Verilog, SIGNAL[22]. Many of the other reactive frameworks support dynamic reconfiguration, but due to implementation as a library within an imperative language, the code that configures – and especially reconfigures – the reactive subsystem obscures the shape of the resulting dataflow. We believe that the single-definition principle provides the greatest potential for making complicated reactive software comprehensible.

3.1.1 Verification of Reactive Systems.

Synchronous programming [13] has been used to design verifiable reactive systems. Halbwachs et al. [14] shows that synchronous programs can be compiled into efficient sequential code and the control structure of the object code is a finite automaton which is synthesized by an exhaustive simulation of a finite abstraction of the program. Jeffrey [21] proposed to extend the type system of functional reactive programming with Linear Temporal Logic which constraints temporal behavior of the reactive program and facilitate the verification of temporal properties. Dimitrova et al. [10] proposed an integration of information flow properties into Linear Temporal Logic to model check information flows in reactive systems. Constant et al. [8] proposed to combine formal verification with conformance testing to detect specification violations in reactive systems. As described earlier, the limitations of the declarative SIGNAL[22] and Esterel[4] languages support automatic analysis [2] and verification[28].

3.1.2 Language Extension for Verification.

In more broader domains, there are other language extensions for developers to use to support automatic verification. Anwar et al. [3] proposed SVOCL, a language extension of System Verilog to express the design verification requirement of the latter. Molotnikov et al. [26] proposed mbeddr, a domain specific C language extension to verify domain-level properties of C language. To the best of our knowledge, thorium represents a novel approach bringing a transactional update model that has much in common with Verilog and other hardware description languages, but with the freedom to operate in the effectively unbounded state-space of software.

4 Thorium Language

Thorium's design was informed by the belief that the restrictions imposed by pure FRP result in reactive code that is easier to reason about. Specifically, the features that we are emphasizing are that each reactive value

1. is either a cell or stream, where
   - cells hold a value at all points in time and
   - streams have a value only at some time instants,
2. is declaratively defined in *one place* in the code using a specialized set of operators on other reactive values, and
3. cannot be observed to differ from that definition from outside of the FRP system.

These restrictions were a fairly natural fit to the language Haskell – in which they were first developed [12] – but are less-easily integrated into the imperative languages that dominate in industry. The most successful effort to bring some of the advantages of FRP to those languages through libraries, the reactive extensions [25], capture the stream logic extremely well, but leave state changes up to the developer. All such libraries suffer from readability challenges due to limitations of the syntax of the host language and

the potential to distribute the logic that injects events into the `observables` across the program. We seek to make pure FRP semantics available to industrial programming languages.

However, we accept that creating a full pure-FRP language that could interact with the outside world would either result in unrealistically modeling the outside world in FRP terms or allowing impurity to seep into the reactive code. Further, industry has programming-language inertia and introducing an entire new language is a daunting task. So, we propose a compromise: we enable a programmer to develop reactor code that operates in a pure-FRP environment and then compile that code into objects callable from industrial languages but that enforces the FRP semantics internally.

For example the reactor `accumulator` defined in figure 1

```
1 reactor accumulator(inc: stream unit) {
2   value: cell int = 0 .. ~value+increment;
3 private:
4   increment: stream int = 1 @ inc;
5 properties:
6   non_negative: G value >= 0;
7   bounded: G (0 <= value and value <= 3);
8 }
```

Figure 1. Accumulator Reactor

defines a reactor that acts as a tally clicker that increments its `value` by one each time that its input, `inc`, is active. This introduces a number of language features that will be explained more fully in section 5, but will be introduced briefly here:

- Line 1 declares the reactor and defines `inc` to be an input of type `stream unit`, meaning that it doesn't carry any information other than whether it is active or not.
- Line 2 declares `value` to be of type `cell int`, meaning that it will have an integer value at all points in time. Further, it is defined using the *hold* syntax "<init> .. <updates>" defining an initial value and the stream of events that will change its value.
- The <updates> portion of the *hold* expression in line 2, `~value+increment`, defines a stream of integers. The ~ operator provides access to the value of its operand immediately before the current transaction within *stream expressions*. It enables the sort of update logic exemplified by this accumulator example while also supporting access to the value of cells in the current state so long as no circular cell definitions are made. This expression says that whenever `increment` is active, this expression will contain the sum of `increment` and the value that `value` held immediately before the transaction in which `increment` became active.

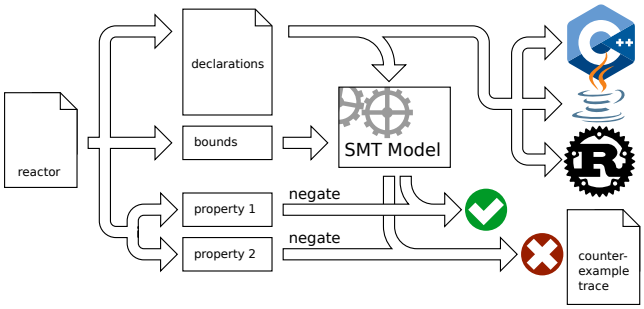

Figure 2. Thorium Workflow

- Line 4 declares `increment` to be of type `stream int` and uses the *snapshot* syntax, "<cell value> @ <event>" that takes a "snapshot" of a the value of a *cell* expression (in this case the constant expression "1") whenever the "<event>" (in this case the `inc` input) stream is active.

That is, for each `inc` input event, the public `value` cell increments by one. Ignoring, for the moment, the properties section, it will result in a C++ class with signature

```
1 class accumulator {
2   accumulator(Stream<int> Unit);
3   Cell<int> value();
4 }
```

Figure 2 illustrates the developer workflow. At present, only the model-checking operations are supported, but we have selected operators and semantics that can be implemented in SodiumFRP [5] so that, regardless of whether the declared properties are valid, a developer will be able to compile a syntactically-valid reactor definition into an executable backend like C++, Java, etc. That artifact can be used for testing and eventual integration into a larger software product. While we believe that the syntax aids program comprehension, our primary value proposition is the ability to provide automated static analysis on the properties declared for a reactor. For the `accumulator` reactor we have asserted that

- non-negative: G value >= 0
  - G(lobally), that is at all points in time, `value` is never negative
- bounded: G (0 <= value and value <= 3)
  - value is never greater than three (an invalid assertion)

The model-checking procedure works as follows:

First, all of the declarations in the reactor are used to produce a single SMT model using the procedure described in section 6. This model represents, through logical constraints, all possible execution traces of length equal to the number of steps specified at the time of verification.

Then, each property is also represented by an SMT model, but with an important caveat. It is negated to produce a

k	0	1	2	3	4	5
inc		unit	unit	unit	unit	unit
value	0	1	2	3	4	5
increment		1	1	1	1	1

Figure 3. `accumulator` counterexample for the `bounded` property. Each column represents the values of each of the declared members of the `accumulator` reactor. `inc` and `increment` are streams of type `unit` and `int`, respectively, and don't carry a value at time $k = 0$.

model of all possible execution traces (of bounded length) for which the property is false. This, in combination with the model of the reactor, produces a model of all possible execution traces that follow the definitions in the reactor and violate the current property.

The desired outcome is that this produces an unsatisfiable model for each inverted property. That means that the definitions of the reactor are sufficient to guarantee that the property will never be violated – at least in an execution trace limited by the specified number of steps. For increased confidence, the programmer can increase the number of steps to be considered, but that will lead to an increase in the time and memory required to complete the analysis as shown in section 7.

When the SMT solver finds an assignment of values that *do* satisfy the model, that represents a property that the reactor definitions fail to enforce. The good news is that, due to the small-step semantics that we model in section 6, that assignment of values maps directly into an execution trace that illustrates one test case for which the reactor definitions are insufficient.

Figure 3 shows the counterexample trace produced for the `bounded` property of the `accumulator` reactor with a verification bound of four steps. The model checker determined that when the `inc` input is active for all four steps, the property is violated. The trace shown in figure 4 shows the full model used to verify the property. The reactor model contains one member for each subexpression in the reactor definition. This is overkill for some subexpressions – constants, in particular, seem like good candidates for optimization – but it simplifies the definition of the reactive operators `hold`, `snapshot`, `filter`, and `merge`.

The final contribution of thorium is the mechanism for supporting runtime reconfiguration. Existing FRP implementations make use of *switch* statements that define a variable of type Cell<T> in terms of a stream of type Stream<Cell<T>> (and corresponding versions for updating variables of type Stream<T>). We believe that this is a potential source of confusion for those with an object-oriented programming background. Instead, thorium supports reactive values – particularly cells – that contain reactor instances. This enables a reactor's behavior to be determined by its sub-reactors.

k	0	1	2	3	4	5
inc		unit	unit	unit	unit	unit
value	0	1	2	3	4	5
increment		1	1	1	1	1
non_negative	True	True	True	True	True	True
bounded	False	False	False	False	False	False
value-1	0	0	0	0	0	0
value-2-1	0	1	2	3	4	5
value-2-2		1	1	1	1	1
value-2		1	2	3	4	5
increment-1	1	1	1	1	1	1
increment-2		unit	unit	unit	unit	unit
non_negative-1-1	0	1	2	3	4	5
non_negative-1-2	0	0	0	0	0	0
non_negative-1	True	True	True	True	True	True
bounded-1-1-1	0	0	0	0	0	0
bounded-1-1-2	0	1	2	3	4	5
bounded-1-1	True	True	True	True	True	True
bounded-1-2-1	0	1	2	3	4	5
bounded-1-2-2	3	3	3	3	3	3
bounded-1-2	True	True	True	True	False	False
bounded-1	True	True	True	True	False	False

Figure 4. The full counterexample trace for the `accumulator` reactor. This is the same counterexample trace as figure 3, but also shows all of the internal state maintained for each of the subexpressions. Any label with a hyphen in the name represents a subexpression of one of the declared reactor members. Note the inefficiency of members like `bounded-1-2-2` that represent constants that could be eliminated. This allowed for a direct translation between the Abstract Syntax Tree (AST) and the SMT model, but will be considered for optimization in future research.

This is functionally equivalent to the *switch* statement, but syntactially resembles object-oriented extension by composition. Currently, the number of sub-reactors is limited by the number of members because container datatypes are not yet supported.

5 Syntax and Semantics

5.1 Declarations

For the purpose of this paper, there are two top-level declarations: datatypes and reactors.

5.1.1 Datatypes. Thorium datatypes are simple Algebraic Data Type (ADT)s providing support for structure-like product types such as

```
datatype Point2D {
    x : real,
    y : real
}
```

enumeration-like sum types such as

```
datatype Shape {
```

```
2    RECT
3  | CIRCLE
4 }
```

and, more interestingly, combining sum types and product types to support definitions like

```
1 datatype Shape2D {
2    Rect{  center: Point2D,  width: real, height: real}
3  | Circle{center: Point2D, radius: real}
4 }
```

Access to the members of product types is accomplished with the "." operator. For example, if `r` is a `Point2D`, the expression `r.x` yields the value of the `x` member of `r`. For sum types, access is accomplished with `match` expressions of the form

⟨*match*⟩ ::= '`match`' ⟨*expr*⟩ '`{`' ⟨*match case*⟩ ('`|`' ⟨*match case*⟩)* '`}`'

⟨*match case*⟩ ::= ⟨*type spec*⟩ => ⟨*expr*⟩

⟨*type spec*⟩ ::= ⟨*ID*⟩(::⟨*ID*⟩)*⟨*match case args*⟩?

⟨*match case args*⟩ ::= '`(`' ⟨*ID*⟩ ('`,`' ⟨*ID*⟩) '`)`'

More concretely, assuming that `shape` is of type `stream Shape2D`, we can define a stream of area measurements with the expression

```
1 match shape { Shape2D::Rect(c,w,h) => w*h
2             | Shape2D::Circle(c,r) => 3.14159*r*r
3             };
```

5.1.2 Reactors.

Reactors define a set of reactive members in terms of reactive inputs and, optionally, define properties that must hold for those reactive values.

⟨*reactor*⟩ ::= '`reactor`' ⟨*ID*⟩ ⟨*params*⟩? '`{`' ⟨*member*⟩* ('`private:`' ⟨*member*⟩*)? ('`properties:`' ⟨*property*⟩*)? '`}`'

⟨*params*⟩ ::= '`(`' ⟨*param*⟩ ('`,`' ⟨*param*⟩)* '`)`'

⟨*param*⟩ ::= ⟨*ID*⟩ '`:`' ⟨*rtype*⟩

⟨*member*⟩ ::= ⟨*ID*⟩ '`:`' ⟨*rtype*⟩ = ⟨*expr*⟩ '`;`'

⟨*rtype*⟩ ::= '`cell`' ⟨*type*⟩ | '`stream`' ⟨*type*⟩

⟨*type*⟩ ::= int | real | bool

⟨*property*⟩ ::= ⟨*ID*⟩ ⟨*LTL*⟩

⟨*LTL*⟩ ::= ('G'|'F'|'P') ⟨*LTL*⟩ | ⟨*LTL*⟩ ('U'|'S'|'=>') ⟨*LTL*⟩ | ⟨*boolean expr of reactor members*⟩

⟨*expr*⟩ ::= ⟨*standard expression syntax*⟩

5.2 Reactive Values

Syntax: As shown above, reactive values within a reactor can be declared with an `<ID>` and `<rtype>`, so the code

```
1    c : cell int ...
2    s : stream real ...
```

declares a cell `c` and a stream `s`, respectively. Reactive values are also declared by any expression that operates on reactive values as shown in the following subsections.

Semantics: Thorium interprets reactive values as functions from dense time to values. By dense time, we mean that for any two time instants, there is always some time instant between them for which the interpretation of the reactive values remains valid.

Reactive values are either cells or streams.

$$r \models \text{Reactive<T>} \implies r \models \text{Cell<T>} \text{ or } r \models \text{Stream<T>}$$

Constants of type T can be treated as Cell<T> when considering operations on reactive types.

$$c \models \text{Cell<T>}$$

c is a total function from dense time to type T. i.e., $c : \mathbf{R}^+ \to \mathbf{T}$

$$s \models \text{Stream<T>}$$

s is a partial function from dense time to type T. i.e., $s : \mathbf{R}^+ \to T \cup \{\bot\}$ where $\bot$ denotes the lack of value.

5.3 Reactive Operators

Thorium supports the FRP operators filter, merge, snapshot, and hold. The first two control streams of events – filter conditionally blocks events and merge converts two event streams into a single stream. The latter two, convert cells to streams and vice versa, respectively.

5.3.1 Filter.

The filter operator takes a stream and only allows those events from that stream for which a corresponding condition is true. Unlike some other filter definitions, the condition need not depend on the value of the event. The condition can even be a cell.

Syntax:

⟨*filter expr*⟩ ::= ⟨*value: expr*⟩ '`if`' ⟨*condition: expr*⟩

where `value` ⊨ Stream<T> and `condition` ⊨ Reactive<bool>

Semantics:

$$\mathtt{s} = \mathtt{v} \text{ if } \mathtt{c} \text{ defines } \mathtt{s} \models \text{Stream<T>}$$

$$\forall t \in \mathbf{R}^+ . s_t = \begin{cases} \bot & \text{if } \mathtt{c}_t \in \{\bot, \mathit{false}\} \\ \mathtt{v}_t & \text{otherwise} \end{cases}$$

5.3.2 Merge.

Since all of the updates within a transaction are considered to be simultaneous, there is no automatic serialization of events within streams. The programmer must specify how the reactor is to behave for all combinations of input streams. This is accomplished by using the merge operator to prioritize which stream should take precedence.

Syntax:

⟨*merge expr*⟩ ::= ⟨*s1: expr*⟩ '`|`' ⟨*s2: expr*⟩

where `s1`, `s2` ⊨ Stream<T>

Semantics: `s` = `s1` | `s2` defines `s` ⊨ Stream<T>

$$\forall t \in \mathbf{R}^+.s_t = \begin{cases} \texttt{s1}_t & \text{if } \texttt{s1}_t \neq \bot \\ \texttt{s2}_t & \text{otherwise} \end{cases}$$

5.3.3 Snapshot.

Snapshot provides a conversion from a cell to a stream whenever a stream has an event. Regardless of the type of the trigger stream, the snapshot operator only checks for the presence or absence of an event.

Syntax:

⟨*snapshot expr*⟩ ::= ⟨*value: expr*⟩ '`@`' ⟨*trigger: expr*⟩

where `value` ⊨ Cell<T_1> and `trigger` ⊨ Stream<T_2>

Semantics:

`s` = `v` @ `t` defines `s` ⊨ Stream<T_1>

$$\forall t \in \mathbf{R}^+.\texttt{s}_t = \begin{cases} \bot & \text{if } \texttt{t}_t = \bot \\ \texttt{v}_t & \text{otherwise} \end{cases}$$

5.3.4 Hold.

The hold operator creates a cell that has the value of the most recent event on an `updates` stream. Since a cell must have a value at all points in time and there might never be an event in the `updates` stream, an `init` value is provides that will be the value of the hold expression until the first event arrives.

Syntax:

⟨*hold expr*⟩ ::= ⟨*init: expr*⟩ '`..`' ⟨*updates: expr*⟩

where `init` ⊨ Cell<T> and `trigger` ⊨ Stream<T>

Semantics:

`c` = `i` .. `u` defines c ⊨ Cell<T>, such that

$$\forall t \in \mathbf{R}^+.c_t = \begin{cases} \texttt{u}_{t'} & \max\, t' \text{ such that } t' <= t \wedge \texttt{u}_{t'} \neq \bot \\ \texttt{i} & \text{otherwise} \end{cases}$$

5.4 Function Application

Arithmetic and logical operators as well as datatype construction can be treated as function application. Functions in Thorium are defined in terms of base (non-reactive) datatypes, so some sort of *lift* operation must take place. When all arguments are cells, this is fairly straightforward: the result is of type Cell<T> where T is the return type of the function and it is always equal to the result of calling the function with the value of each argument; if any of the arguments is a stream, then the result is of type Stream<T> where T is the return type of the function and it is only active when all of its inputs are active and at that time it contains the result of calling the function with the value of each argument.

5.4.1 Function application on cells.

For a function

$$f : T_1, T_2, \ldots, T_p \rightarrow T$$

and parameters

$$c_1, c_2, \ldots, c_p \in \text{Cell<}T_1\text{>}, \text{Cell<}T_2\text{>}, \ldots, \text{Cell<}T_p\text{>},$$

$c = f(c_1, c_2, \ldots, c_p)$ defines c ⊨ Cell<T> such that

$$\forall t \in \mathbf{R}^+.c_t = f(c_{1_t}, c_{2_t}, \ldots, c_{k_t})$$

5.4.2 Function application on streams.

for a function

$$f : T_1, T_2, \ldots, T_p \rightarrow T$$

and parameters

$$r_1, r_2, \ldots, r_p \in \text{Reactive<}T_1\text{>}, \text{Reactive<}T_2\text{>}, \ldots, \text{Reactive<}T_p\text{>}$$

for which at least one parameter is a stream,

$s = f(r_1, r_2, \ldots, r_p)$ defines s ⊨ Stream<T>, such that

$$\forall t \in \mathbf{R}^+.s_t = \begin{cases} \bot & \text{if } \exists k \in [1, p].r_{k_t} = \bot \\ f(r_{1_t}, r_{2_t}, \ldots, r_{k_t}) & \text{otherwise} \end{cases}$$

5.5 Composition

The previous sections described the semantics for a reactor composed of basic datatypes. To capture applications that change their behavior in response to input events, thorium reactors support members of reactive types carrying reactors. When declaring a reactor that takes a reactive argument, there are two ways that it could be interpreted: The constructor should either capture the specific stream that existed at the time of the reactor's creation or remember how the stream was defined and update the input stream if the result of that definition changes. This is most important when dealing with the situation when the output of one reactor(A) feeds the input of another(B). If reactor A is replaced with a new instance, should B continue to receive the output of the instance at the time of B's creation or begin consuming the output of the new instance. We have opted for the latter and find that it best reflects the behavior when dealing with non-reactor members. See the Pipeline reactor in figure 5.

6 SMT Mapping

In the original formulation of FRP, both cells and streams are interpreted as functions from time to a value. The primary innovation of thorium is its emphasis on the reactor, rather than the cell and stream, as the fundamental reactive element. In the runtime, this is seen in that the behavior of a cell or stream in a reactor can only change by replacing a subreactor with a new one with a different behavior. In the thorium model-checker, this is reflected in the fact that all execution traces are mappings from time instants to an entire reactor state rather than an individual cell or stream.

This slightly complicates the definition of the operators. Instead of an expression like $A = B + C$ mapping to

$$\forall k, A[k] = B[k] + C[k],$$

it is more accurately represented as

$$\forall k, Trace[k].A = Trace[k].B + Trace[k].C$$

Further, we avoid all use of quantification, so the $\forall k$ is actually implemented as a series of assertions for each k in the specific reactor's start state k_0 (not necessarily state 0) and k_K, the final state in the bounded model. This approach enables the representation of multiple instances of the same

reactor type created at different times over the duration of the bounded model checker.

The model for the trace for a reactor that is created at time k_0 is produced by defining a member for every subexpression in the reactor. Then the value of each subexpression member is defined at each time index k as a function of its child sub-expressions and the semantics of the subexpression. The following subsections describe how we represent thorium's semantics in the Python binding of the Z3 SMT solver. The first describes our representation for reactive values and the remainder describe the process for defining those reactive values for a selection of the supported operators.

6.1 Reactive Values

As introduced earlier, our FRP semantics are defined on two reactive value kinds: the cell and the stream: A `cell T` has a value of type T at all points in time and a `stream T` has a value of type T only at points of time in which an event arrives on that stream. Conceptually, a reactive value of type T is a function from real-valued time t to a value of type T (or *nothing* in the case of a stream that does not have an event at time t). However, in our Z3 implementation we use the `Array` construct rather than an actual `Function` because our semantics only allows changes in value in discrete transactions, so real-valued time is unnecessary and, unlike functions, arrays can be composed in Z3. This is necessary to support composable reactors since each reactor instance defines a trace of state transitions. Using an array rather than a function does not materially impact our definitions because we explicitly assert the definition of each reactive value at each time instant.

This is quite straightforward for cells: the Z3 model simply represents them as their fundamental datatype. That is, a `cell int` member becomes an `Int` in Z3. However, for streams, we need a way to explicitly assert that the stream is inactive at those states for which it has no value. Were we to represent it with its fundamental data type and simply leave its value undefined at those states, Z3 would be free to assume any value. Instead, we produce a new datatype Stream<T> that either contains an event with a value of type T or nothing (when the stream is not active). In SMTLib2, this is represented as

```
(declare-datatypes (T)
   ((Stream (event (value T))
            (nothing))))
```

Since the Python API lacks support for generic datatypes, each type specialization must be declared explicitly. This can be automated with a function like

```
def declare_stream(type:str , T):
  stream = z3.Datatype(f'stream-{type}')
  stream.declare('event', ('value', T)))
  stream.declare('nothing')
  return stream.create()
```

Unfortunately, this means that directly accessing the value held in a stream differs from that in a cell and thus complicates the definition of all operators that can operate on combinations of stream and cell values. To mitigate this, we make use of the `ReactiveValue` class that:

1. presents each member of a reactor within a trace as a function from time to a value
2. abstracts away the differences between streams and cells where that distinction is unimportant
3. allows all streams to be checked for emptiness with a uniform syntax regardless of the underlying Z3 type.
4. supports optionally treating the presence of a value in a stream as a Boolean when desirable. e.g., snapshot triggers and within the "active()" pseudo-function.

```
class ReactiveValue:
  def __init__(self, trace, accessor, thorium_type,
       z3_type):
    self.trace = trace
    self.accessor = accessor
    self.thorium_type = thorium_type
    self.z3_type = z3_type

  def isStream(self):
    return isinstance(self.thorium_type, Stream)

  def isNothing(self, k):
    if self.isStream():
      return self(k) == self.z3_type.nothing
    return False

  def isActive(self, k):
    if self.isStream():
      return self(k) != self.z3_type.nothing
    return True

  def isTrue(self, k):
    if self.isStream():
      return z3.If(self.isNothing(k), False, self[k])
    return self(k)

  def setValue(self, k, value):
    if self.isStream():
      return self(k) == self.z3_type.event(value)
    return self(k) == value

  def __call__(self, k):
    return self.accessor(self.trace[k])

  def __getitem__(self,k):
    if self.isStream():
      return self.z3_type.value(self(k))
    return self(k)
```

6.2 Reactive Operators

Because they are the fundamental thorium operations, we present the SMT mapping for all of the reactive operators defined in section 5.3: *filter, merge, snapshot*, and *hold*.

6.2.1 Filter. This function enforces the semantics defined in section 5.3.1. Informally, that the `result` will be equal to `value` if the `value` and `condition` are active and `condition` is *true*. It returns a set of assertions that are added to the SMT solver by the caller.

```
1 def filter(k0        : int, # initial state
2            kK        : int, # final state
3            result    : ReactiveValue,
4            value     : ReactiveValue,
5            condition : ReactiveValue):
6     yield result.isNothing(k0-1)
7     for k in range(k0, kK+1):
8         active = z3.And(condition.isActive(k),
9                         value.isActive(k),
10                        condition[k])
11        yield z3.If(active,
12                    result.setValue(k,value[k]),
13                    result.isNothing(k))
```

6.2.2 Snapshot. This function enforces the semantics of section 5.3.3. Informally, the `result` takes on the value of `cell` only when `stream` is active.

```
1 def snapshot(k0        : int, # initial state
2              kK        : int, # final state
3              result    : ReactiveValue,
4              cell      : ReactiveValue,
5              stream    : ReactiveValue):
6     yield result.isNothing(k0-1)
7     for k in range(k0, kK+1):
8         yield z3.If(stream.isNothing(k),
9                     result.isNothing(k),
10                    result.setValue(k, cell[k]))
```

6.2.3 Merge. This function enforces the semantics of section 5.3.2. Informally, the it prioritizes `s1` by only allowing `s2` to pass through if `s1` is not active.

```
1 def merge(k0     : int, # initial state
2           kK     : int, # final state
3           result : ReactiveValue,
4           s1     : ReactiveValue,
5           s2     : ReactiveValue):
6     yield result.isNothing(k0-1)
7     for k in range(k0, kK+1):
8         yield result(k) == z3.If(s1.isNothing(k),
9                                  s2(k),
10                                 s1(k))
```

6.2.4 Hold. This function enforces the semantics of section 5.3.4. Informally, the it begins with the value `init` and only changing (to the value carried by the event in `update`) when `update` is active. Even if `init` is a cell that might change at some future time, the semantics are ensured by only taking the value of init at time k_0 and then defining `result` at future times in terms of the previous value of `result` or the value of `update`. It places the value of `init` in the pre-history of the `result` (time $k_0 - 1$) so that if the `update` is active at time k_0, it will replace the value of `init`.

```
1 def hold(k0     : int, # initial state
2          kK     : int, # final state
3          result : ReactiveValue,
4          init   : ReactiveValue,
5          update : ReactiveValue):
6     yield result[k0-1] == init[k0]
7     for k in range(k0, kK+1):
8         yield result[k] == z3.If(update.isNothing(k),
9                                  result[k-1],
10                                 update[k])
```

The for loop on line 3 iterates through all states at which streams may be active, i.e. $[k_0, k_K]$. At each state, line 4 sets the value of `result` depending on whether the update stream has a value. If not, then the value of `result` will be unchanged, i.e. equal to its value in the previous state. Otherwise, it takes on the value held in the update stream.

6.3 Temporal Logic

6.3.1 Globally. The Linear Temporal Logic (LTL) operators present an additional complication. Strictly speaking, they are defined on infinitely-long traces. More importantly, for the future-time LTL operators, *globally*(G), *eventually*(F), and *until*(U), they can describe properties that cannot be satisfied by a finite trace.

```
1 def globally(k0     : int, # initial state
2              kK     : int, # final state
3              result : ReactiveValue,
4              arg    : ReactiveValue):
5     for k in range(k0, kK+1):
6         yield result[k] == z3.And(arg.isTrue(k),
7                                   result[k+1])
8     # optimistic semantics
9     yield result[kK+1] == True
```

In the case of *globally*, this is fairly naturally addressed by the so-called *optimistic* semantics. We require that the property holds at each state within the bound, but trust that it will hold for any state not explicitly checked.

6.3.2 Since. The *since* operator, though the most complicated of the past-time LTL operators, requires no such accommodations. It can be completely defined on a finite trace. The interpretation of pSq is that, for it to be true at time k, either q must be true at time k or p must be true and have been true since the last time that q was true. This is recursively defined with

```
def since(k0     : int, # initial state
          kK     : int, # final state
          result : ReactiveValue,
          p      : ReactiveValue,
          q      : ReactiveValue):
    yield z3.Not(result[k0-1])
    for k in range(k0, kK+1):
        yield result[k] == z3.Or(q.isTrue(k),
                                 z3.And(p.isTrue(k),
                                        result[k-1]))
```

6.4 Apply

One of the most important – and complex – operators in thorium is `apply`. It is used to implement function calls, `struct` instantiation, and all basic operators. When all arguments are cells (line 16), its operation is fairly straightforward. It defines the value of `result` at each time instant `k` as the value returned by the function `f` when passed the value of each of the arguments at that time instant.

```
def apply(k0     : int, # initial state
          kK     : int, # final state
          result : ReactiveValue,
          f      : callable,
          args   : List[ReactiveValue]):
  stream_args = [arg for arg in args if arg.isStream()]
  for k in range(k0, kK+1)
      values = [arg[k] for arg in args]
      if stream_args:
          active = z3.And(*[arg.isActive(k)
                            for arg in stream_args])
          yield z3.If(active,
                      result.setValue(k, f(*values)),
                      result.isNothing(k)))
      else: # all cells
          yield result.setValue(k, f(*values))
```

However, when any input is a stream, it ensures that the output will only have a value when all of the inputs are active.

6.4.1 Sub-Reactors. Each reactor state is represented as a Z3 structure containing the value of all of its members. The naive approach would be to simply use that datatype as the type of the sub-reactor member. This was our original approach when composition was only supported during the initial state. Now that we support arbitrary replacement of reactors, we have switched to a new approach. All members of type reactor are represented as an integer. That integer represents the index of the reactor in a set of "heap" arrays (one for each reactor type). Whenever members of a sub-reactor must be used in an expression of the larger reactor, a special accessor function is created to ensure that the the index into the heap array is updated whenever the instance is replaced in the parent reactor.

7 Evaluation

7.1 Reconfigurable Computation Pipeline

We begin with a reactor that exercises arbitrary reconfiguration of sub-reactors. It implements a simple computational pipeline for which any of the stages may be changed during execution. In this case, each of the stages performs the same operation, multiplication by the configured coefficient. A more realistic computational pipeline would perform different operations in each of the stages, but this is simple enough to reason about and allowed us to contrive a property that could only be violated by reconfiguring each of the processing stages, though they can be reconfigured in arbitrary order.

```
1  reactor Mult( c: cell int, in : stream int) {
2      out: stream int = c * in;
3  }
4
5  datatype CFG { S1: int
6               | S2: int
7               | S3: int }
8
9  reactor ReconfigurablePipeline(cfg: stream CFG,
10                                 in: stream int) {
11   s1: cell Mult = Mult(1, in)
12       .. match cfg { CFG::S1(c) => Mult(c, in) };
13   s2: cell Mult = Mult(1, s1.out)
14       .. match cfg { CFG::S2(c) => Mult(c, s1.out) };
15   s3: cell Mult = Mult(1, s2.out)
16       .. match cfg { CFG::S3(c) => Mult(c, s2.out) };
17   out: stream int = s3.out;
18 private:
19   C: cell int = 0 .. match cfg { CFG::S1(c) => c
20                                | CFG::S2(c) => c
21                                | CFG::S3(c) => c };
22 properties:
23     bounded: (G (0<=C and C<10))
24                  => not F ( (in==1) and
25                             (out/in > 100) );
26 }
```

Figure 5. Reconfigurable Pipeline

Lines 1-3 define the reactor Mult that is configured with a coefficient that will be multiplied with the input stream to produce the output stream. Lines 5-7 define the datatype CFG as a sum type that will contain one of the three values S1, S2, or S3. This will be used in the Pipeline reactor to determine which of the stages will be reconfigured. In the definition of the Pipeline, lines 11-16 define three stages such that `s2` consumes the output of `s1` and `s3` consumes the output of `s2`. It is important to note that the expression `s2.out` will automatically update when `s2` is updated in response to a `cfg` input that contains `CFG::S2` as defined

on line 13. Finally, lines 23-25 define the property, `bounded`, that asserts that if the coefficient of any of the `cfg` inputs is non-negative and less than ten, then the product of the configured coefficients will be less than 100. This is not true, of course, but thanks to the limitation on the coefficients, none of the three coefficients can be left at the initial value of one.

ReconfigurablePipeline

k	0	1	2
cfg	CFG::S3(8)	CFG::S1(3)	CFG::S2(8)
in			1
s1	Mult-0	Mult-2	Mult-2
s2	Mult-4	Mult-4	Mult-7
s3	Mult-9	Mult-9	Mult-9
out			192
C	8	3	8

Mult-0

k	0	1	2
c	1	1	1
in			1
out			1

Mult-1

k	0	1	2
c	-1	-1	-1
in			1
out			-1

Mult-2

k	1	2
c	3	3
in		1
out		3

Mult-3

k	2
c	-1
in	1
out	-1

Mult-4

k	0	1	2
c	1	1	1
in			3
out			3

Mult-5

k	0	1	2
c	16	16	16
in			3
out			48

Mult-6

k	1	2
c	-2	-2
in		3
out		-6

Mult-7

k	2
c	8
in	3
out	24

Mult-8

k	0	1	2
c	1	1	1
in			24
out			24

Mult-9

k	0	1	2
c	8	8	8
in			24
out			192

Mult-10

k	1	2
c	6	6
in		24
out		144

Mult-11

k	2
c	1
in	24
out	24

Figure 6. Counterexample trace for the ReconfigurablePipeline

Initially, we developed a two-stage version of the pipeline. The performance looked quite promising, generally taking less than a minute to find a contradiction, so we increased the number of stages to the three shown above. We were struck by the increased processing time; our initial attempts took thirty minutes to an hour.

7.2 Performance Baseline for Reconfigurable Pipeline

To evaluate the overhead of the reactor composition, we created a baseline implementation that performed all of the same operations, but without the reactor composition. It uses the CFG datatype to determine which coefficient will be updated in each step.

```
1 datatype CFG { S1: int
2              | S2: int
3              | S3: int }
4
5 reactor BaselinePipeline(cfg: stream CFG,
6                           in: stream int) {
7   coef1: cell int = 1 .. match cfg {CFG::S1(c) => c};
8   coef2: cell int = 1 .. match cfg {CFG::S2(c) => c};
9   coef3: cell int = 1 .. match cfg {CFG::S3(c) => c};
10  out: stream int = coef1 * coef2 * coef3 * in;
11 private:
12  C: cell int = 0 .. match cfg {CFG::S1(c) => c
13                                |CFG::S2(c) => c
14                                |CFG::S3(c) => c
15                                };
16 properties:
17  bounded: (G (0<=C and C<10))
18               => not F ( (in==1) and
19                          (out/in > 100) );
20 }
```

BaselinePipeline

k	0	1	2
cfg	CFG::S1(2)	CFG::S3(7)	CFG::S2(8)
in	0		1
coef1	2	2	2
coef2	1	1	8
coef3	1	7	7
out	0		112
C	2	7	8

Figure 7. Counterexample trace for the BaselinePipeline

The `BaselinePipeline` – the version of the pipeline without reactor composition – was substantially faster, taking between one and ten seconds rather than the thirty minutes to an hour. Clearly, the current approach to modeling sub-reactor instances leaves room for improvement.

7.3 Sequential Pipeline

To evaluate the impact of the ability to reconfigure the stages in arbitrary order in the first reactor, we tried one more variant: The `SequentialPipeline`. Unlike the arbitrarily reconfigurable pipeline, the sequential pipeline always builds the pipeline in order, i.e., the first event in the `cfg` stream will be used to configure the first stage; the second will configure the second stage, and so on.

```
reactor Mult( c: cell int, in : stream int) {
    out: stream int = c * in;
}

reactor SequentialPipeline(cfg: stream int,
                           in: stream int) {
    s1: cell Mult = Mult(1, in)
            .. Mult(cfg, in) @ cfg if stage==1;
    s2: cell Mult = Mult(1, s1.out)
            .. Mult(cfg, s1.out) @ cfg if stage==2;
    s3: cell Mult = Mult(1, s2.out)
            .. Mult(cfg, s2.out) @ cfg if stage==3;
    out: stream int = s3.out;
private:
    stage: cell int = 0 .. ~stage + 1 @ cfg;
    C: cell int = 0 .. cfg;
properties:
    bounded: (G (0<=C and C<10)) =>
                not F ( (in==1) and
                        (out/in > 100) );
}
```

7.4 Performance Comparisons

We found in our early experiments that

- the number of stages had a significant effect on runtime and
- the runtime for a given number of stages was highly variable.

To measure both the growth in runtime and the variability, we produced three versions each of the `BaselinePipeline`, `ReconfigurablePipeline`, and `SequentialPipeline` in which each pipeline contained one, two, or three stages. Each pipeline was verified ten times with a bound of three time steps for each, producing the plot in figure 8.

The y-axis of figure 8 is logarithmic, so we are seeing exponential growth in runtime as a function of the number of stages. It is interesting that the variability in runtime was highest in the two-stage experiments. We believe that this was caused by the fact that all of the tests were performed with a bound of three steps. Three steps are required to identify a violation when there are three stages, so there isn't much room for variability. The one-stage case doesn't present any options to the solver about which stage to update. Regardless, this suggests that it may be valuable to check for counterexamples in shorter traces first because giving the solver unnecessary degrees of freedom to find the counterexample increases the variance in solve time.

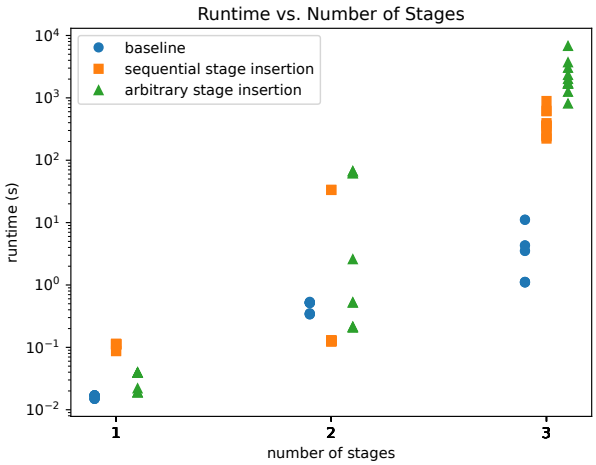

Figure 8. Growth in pipeline verification time vs. number of stages

8 Conclusion and Future Work

We present a first attempt at a verifier for a language that supports dynamic reactive programs. We believe that the syntax that we have developed makes the intent and behavior of the reactor more clear than most library-based reactive frameworks and the synchronous semantics enable a bounded model checking procedure that identifies errors far more reliably than testing – provided that the properties are properly specified.

The performance results suggest that the next step is to search for optimizations. Our first attempt will be to condition each of the assertions for a reactor on the condition that leads to the creation of that reactor instance. Hopefully that will free the solver to focus on the free variables that might impact the result.

The most significant future work for the language will be the introduction of containers so that pipelines might be built with an arbitrary number of stages – though it seems that the practicality of doing so will be contingent on bringing down the cost of reactor composition.

For the present, despite its limitations, we have established that our selected language features and semantics enable model checking of actual code potentially avoiding an error prone process of manually translating from a modeling language.

References

[1] Bowen Alpern and Fred B. Schneider. 1987. Recognizing Safety and Liveness. *Distributed Computing* 2 (1987), 117–126.

[2] Hafiz Muhammad Amjad, Kai Hu, Jianwei Niu, Noor Khan, Loïc Besnard, and Jean-Pierre Talpin. 2019. Translation Validation of Code Generation from the SIGNAL Data-Flow Language to Verilog. In *2019 15th International Conference on Semantics, Knowledge and Grids (SKG)*. 153–160. https://doi.org/10.1109/SKG49510.2019.00034

[3] Muhammad Waseem Anwar, Muhammad Rashid, Farooque Azam, and Muhammad Kashif. 2017. Model-based design verification for embedded systems through SVOCL: an OCL extension for SystemVerilog. *Design Automation for Embedded Systems* 21, 1 (2017), 1–36.

[4] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

[5] Stephen Blackheath. 2016 (accessed 9 January 2017). Sodium. https://github.com/SodiumFRP/sodium.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. 1987. LUSTRE: A Declarative Language for Real-Time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 178–188. https://doi.org/10.1145/41625.41641

[7] The Qt Company. 2022. Qt|Cross-platform software for embedded and desktop. http://qt.io.

[8] Camille Constant, Thierry Jéron, Hervé Marchand, and Vlad Rusu. 2007. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering* 33, 8 (2007), 558–574.

[9] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-proving. *Commun. ACM* 5, 7 (July 1962), 394–397. https://doi.org/10.1145/368273.368557

[10] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N Rabe, and Helmut Seidl. 2012. Model checking information flow in reactive systems. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 169–185.

[11] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 361–376. https://doi.org/10.1145/2660193.2660240

[12] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. *SIGPLAN Not.* 32, 8 (Aug. 1997), 263–273. https://doi.org/10.1145/258949.258973

[13] Nicolas Halbwachs. 1998. Synchronous programming of reactive systems. In *International Conference on Computer Aided Verification*. Springer, 1–16.

[14] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. 1994. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology (AMAST'93)*. Springer, 83–96.

[15] Philipp Haller and Stephen Tu. (accessed 6 Mar 2023). Scala Actors API. https://docs.scala-lang.org/overviews/core/actors.html.

[16] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8 (1987), 231–274.

[17] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA) *(IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[18] Inc. Itemis. 2023. Itemis CREATE - state machines made easy. https://www.itemis.com/en/products/itemis-create/.

[19] Inc. Itemis. (Last updated 2021). Yakindu. https://github.com/Yakindu/statecharts.

[20] Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[21] Alan Jeffrey. 2012. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming languages meets program verification*. 49–60.

[22] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. 1991. Programming real-time applications with SIGNAL. *Proc. IEEE* 79, 9 (1991), 1321–1336. https://doi.org/10.1109/5.97301

[23] Inc Lightbend. 2011-2023(accessed 6 Mar 2023). Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala. https://akka.io/.

[24] Andrés Goens Patricia Derler Jeronimo Castrillon Edward A. Lee Marten Lohstroh, Iñigo Incer Romeo and Alberto Sangiovanni-Vincentelli. 2019. Reactors: A Deterministic Model for Composable Reactive Systems. In *Model-Based Design of Cyber Physical Systems (CyPhy)*. https://www.icyphy.org/publications/2019_LohstrohEtAl3/

[25] Erik Meijer. 2010. Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues. In *ACM SIGPLAN Commercial Users of Functional Programming* (Baltimore, Maryland) *(CUFP '10)*. ACM, New York, NY, USA, Article 11, 1 pages. https://doi.org/10.1145/1900160.1900173

[26] Zaur Molotnikov, Markus Völter, and Daniel Ratiu. 2014. Automated domain-specific C verification with mbeddr. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 539–550.

[27] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: bridging between object-oriented and functional style in reactive applications. In *MODULARITY*.

[28] Yahui Song and Wei-Ngan Chin. 2021. A Synchronous Effects Logic for Temporal Verification of Pure Esterel. In *Verification, Model Checking, and Abstract Interpretation: 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17–19, 2021, Proceedings* (Copenhagen, Denmark). Springer-Verlag, Berlin, Heidelberg, 417–440. https://doi.org/10.1007/978-3-030-67067-2_19

Received 2023-07-21; accepted 2023-08-29