Reactive Functional Programming *

Richard B. Kieburtz

Oregon Graduate Institute of Science & Technology
P.O. Box 91000, Portland, OR 97291-1000 USA

October 13, 1997

Abstract

Reactive systems respond to concurrent, possibly unsynchronized streams of input events. Programming reactive systems is challenging without language support for event-triggered actions. It is even more challenging to reason about reactive systems. This paper explores a new conceptual basis for applying functional programming techniques to the design and formal verification of reactive systems. The mathematical foundation for this approach is based upon signature coalgebras and derived proof rules for coinduction. The concepts are illustrated with an example that has been used with the language *Esterel*.

1 Introduction

Reactive systems are characterized by sequences of history-determined reactions to external events. It is known that a non-strict functional programming language can provide a suitable linguistic vehicle for programming reactive systems because streams, modeling temporal sequences of values, can be represented. It is necessary to represent more than streams, however. Current reactive programming languages, such as *Esterel*, *Lustre* and *Statecharts* provide implicit or explicit representations of state, iterative control structures, and parallel threads of activity. Use of these languages has advanced the state of the art of designing reactive systems, however it is not easy to reason about their properties. For ease of reasoning, we should like to have a sound programming logic that is expressive over the terms of the programming language. A principal motivation for this research is to develop in tandem a programming notation well suited to specifying reactive systems, and an associated programming logic.

The control structure needed for reactive programs is inherently iterative, not recursive. The data of interest are infinitary sequences or trees of states, representing the evolution of systems that may never terminate. We have searched for an underlying mathematical structure to model reactive systems. The structure we have found most useful is that of coalgebras, which unfortunately, are not very well understood by most functional programmers.

*The research reported in this paper was supported by the USAF Materiel Command.

1.1 Mathematical models for programming

In the natural sciences and in related fields of engineering, the importance of mathematical models is well appreciated.

- Models abstract away confusing details and focus attention on fundamental concepts.
- They provide a theory in which to reason about properties of nature or complex engineered systems.
- They make precise and quantitative the underlying relationships between directly and indirectly observable phenomena and their controlling parameters.
- They provide structure to help engineers create reliable designs with predictable behavior.

Appropriate mathematical models for computations can serve the same useful functions that they do in other sciences and engineering disciplines. But what models are most appropriate?

The models in common use in programming are cpo models of computational domains. These allow us to calculate solutions to recursive equations, thought to be the universal foundation of functional programming. However, these models don't help much to abstract away detail—they force us to encode it. The theory of cpo models is difficult to apply—its logic is based upon the principle of computational (or fixed-point) induction. Because of encodings, the relationship of controlling parameters to observables is not always clear, although clarifying this relationship is the main claim of functional programming. The structure that this theory provides for the designer of programs is just function definitions written in terms of recursive equations. It seems inadequate.

Our desire to have better mathematical models for programming has led us to seek less universal, more detailed mathematical structures that may guide us to rely less on encodings and more on compositional principles in designing programs. We believe we have found suitable models in structure algebras and their duals, coalgebras. They are not universal but appear to be adequate to model most classes of programs, with the exception of interpreters for programming languages themselves. Interpreters for interesting languages require universal (i.e. Turing complete) models of computation.

With the goals of our research set forth, we ask the reader's patience in looking at an unaccustomed way of formalizing functional programs, and invite her/his assessment of its usefulness. Section 2 of the paper introduces the notation and concepts of programming with coalgebras. Section 3 illustrates application of the concepts to solve a non-trivial example that has previously appeared in the literature, and to verify some properties of the solution. Section 4 presents conclusions.

1.2 Iterative functional programming

In the past few years, several researchers have observed that programming with bounded recursion is algebraic in nature [Bir86, MFP91, Kie94]. Some results of this body of research include the discovery that recursion over typed data structures has a logical counterpart in structural induction, that monads encapsulate effects in particular algebras, and that type-parametric combinators can be embedded in a strict functional programming language to support this style of program construction [KL94]. There is a dual to algebraic programming and it is useful in another style of functional programming, which is the topic of this paper.

Process-oriented programs are iterative. They are controlled by tests of their partial results, driven by external events rather than by the interpretation of data, and are naturally modeled by coalgebras. Because control is derived to meet external demand rather than induced by the structure of arguments, a non-strict evaluation mechanism is needed. The rules of coinduction induced by codatatypes are logical duals of the familiar rules of structural induction that are induced by datatype definitions. Dual to the recursive structure of algebraic programs is the iterative structure of coalgebraic programs, which can be made manifest by embedding a set of type-parametric combinators in a non-strict programming language.

Examples of iterative algorithms are common. They include linear and tree-structured searching, shift-reduce parsing, and both synchronous and asynchronous reactive systems. Before looking at examples, including proof rules, let's introduce a formalism for expressing coalgebraic programs. The notation is used in *DUALITY*, which is a new functional language based upon algebras and coalgebras as its fundamental computational structures. In this paper we shall deal only with the coalgebraic part. An early version of this language has been implemented and is described in a technical report [KL94].

2 Covarieties of coalgebras

A covariety is a class of coalgebras with a common signature. The archetypical example is the covariety of stream coalgebras, whose signature is:

$$\textbf{cosig}\ \mathit{Stream}(a)\{\textbf{type}\ c;\ \ \mathit{str}/c : \{\$\mathit{shd} : a,\ \ \$\mathit{stl} : c\}\};$$

Here *Stream* is the name of the covariety and *str* is the name of its single sort, analogous to a type constructor. Each *Stream*-coalgebra has a type parameter, a, a carrier type, c, and two projectors identified by the symbols *\$shd* and *\$stl*. The projectors are total functions whose domain is the carrier and whose codomain is indicated by the typing given to each projector identifier in the signature. A signature of projectors can be thought of as a generalized record declaration. Binding a type for the carrier and typed functions for the projectors defines a specific coalgebra parameterized by the type variable, a.

Every covariety defined in this way contains a final coalgebra which is unique up to isomorphism. Call the carrier of the final *Stream*-coalgebra $\mathit{str}(a)$ and call its projectors *Shd* and *Stl*. The finality condition asserts that given any coalgebra of the variety *Stream*, with bindings $\{c := t,\ \$\mathit{shd} := f,\ \$\mathit{stl} := g\}$, there is an assignment of the type parameter, $a := t'$, and a unique mapping, $h : t \to \mathit{str}(t')$, which satisfies a homomorphism condition expressed by the following pair of equations:

$$\begin{aligned} \mathit{Stl} \circ h &= h \circ g \\ \mathit{Shd} \circ h &= f \end{aligned}$$

The significance of the final coalgebra is that any *Stream*-coalgebra may be represented as an infinite sequence of values. Because of this property, it is often said that a stream is an infinite list. While that is certainly one way to encode a stream as data, it is by no means the only way. If we accept that a stream is *codata*, then encoding it as data seems unnecessary. A codata object is defined with methods for observing it, rather than with constructors for building its representation.

Every stream is infinite; that is, it is meaningful to iterate the projection operator *Stl* on a stream arbitrarily many times, even though there is no way to witness the entire stream at once. A stream provides a good model for an incrementally readable input file. The projection *Shd* yields the value of the first element of a stream, just as a *get* operation on an open file produces a value from it. The projection *Stl* yields the rest of a stream, but it is not manifested until projections of it are taken. The situation is familiar in non-strict functional languages.

There are infinitely many access paths to elements of a stream. A path is expressed by a well-typed composition of the projectors *Shd* and *Stl*, i.e. $\mathit{Shd} \circ \underbrace{\mathit{Stl} \circ \cdots \circ \mathit{Stl}}_{i \text{ times}}$, for $i \geq 0$.

2.1 Generators of codata

We say that the carrier of a final coalgebra is a type of codata, meaning that data values can be gotten from it by projection. Data and codata are distinguished by the type system of *DUALITY*. When a sort of a covariety is used as a type constructor, it designates a type of codata.

If $T(a)$ is a covariety and t is a sort symbol of the signature T, a generator of sort t is a function with a type $c \to t(a)$, where the codomain can be recognized as a type of codata [1]. To determine a generator, we must specify a coalgebra by naming a covariety, a type for the carrier and bindings of functions for the projectors. Coalgebras are first-class objects of *DUALITY*.

Example 2.1 : Integer sequences
Declare a *Stream* coalgebra by:

$$\textbf{coalgebra}\ \mathit{intseq} \doteq \mathit{Stream}\{c := \mathit{int};\ \$\mathit{shd} := \mathit{id},\ \$\mathit{stl} := \mathit{add1}\}$$

where $\mathit{add1} \doteq \lambda n.\, n+1$. To obtain an expression of type $\mathit{str}(\mathit{int})$, we can apply the *DUALITY* combinator, $\mathit{gen}[\mathit{str}]$, to the coalgebra specification, creating a generator for this type,

$$\mathit{gen}[\mathit{str}]\ \mathit{intseq} : \mathit{int} \to \mathit{str}(\mathit{int})$$

The combinator $\mathit{gen}[\mathit{str}]$ is an instance of a higher-order combinator, *gen*, specialized to the sort *str* of the covariety *Stream*. The higher-order combinators in *DUALITY* substitute for the recursion operator found in conventional functional languages. The generator $h \doteq \mathit{gen}[\mathit{str}]\ \mathit{intseq}$

[1] Readers familiar with the notion of *anamorphism* [MFP91] may be tempted to identify generators with anamorphisms. The analogy is false, in general. It would be valid in a computational domain of cpo's, in which the function space encompasses all functions defineable by least fixpoints. The models we consider are based upon ordinary sets, not cpo's, and the function spaces are comprised of partial functions. Moreover, they use two kinds of domains, data (sets) and codata (computations).

is the unique map taking the coalgebra *intseq* to a final *Stream* coalgebra. The homomorphism condition it satisfies is expressed by the pair of equations:

$$\begin{aligned} \mathit{Shd} \circ h &= \mathit{id}_{int} \\ \mathit{Stl} \circ h &= h \circ \mathit{add1} \end{aligned}$$

□

Other applications of *Stream* coalgebra generators define pseudo-random number sequences, sequences of unique identifiers and other enumerated sets.

2.1.1 A proof rule for stream generators

All functional programmers are familiar with proof rules based upon induction. Somewhat less familiar is the dual rule of coinduction. The possible observations of a codata object are enumerable composites of a finite basis of primitive witness functions. The coinduction principle is that the finitely observable properties of an object completely characterize it, even if the object is not finitary. An exposition of coinduction is given by Paulson [Pau93].

To define a proof rule for a stream of elements of type a, generated from a carrier of type t, let P be a two-place, typed predicate symbol whose arguments range over t and a, respectively. We prefer a two-place predicate because it can express the input-output relation of a function. Coinduction extends the domain of the relation to infinitary objects. A proof rule for a stream of elements is:

$$\frac{\begin{array}{l} x_0 \; : \; t \\ f \; : \; t \rightarrow a \\ g \; : \; t \rightarrow t. \quad \forall x \; : \; t.\, P(x_0, x) \Rightarrow P(x_0, g\, x) \end{array}}{P(x_0,\, x_0) \Rightarrow \Box P(x_0,\, \mathit{gen}[\mathit{str}]\{c := t;\; \$\mathit{shd} := f,\; \$\mathit{stl} := g\}\; x_0)}$$

We have used a linear temporal operator, $\Box$ (read as *always*), as a quantifier on the predicate P in the consequent of the rule to express that the proposition $P(x_0, x)$ is asserted for every element, x, of the generated stream.

2.2 Coalgebra homomorphisms define iteration schemes

A compelling reason to consider coalgebras is that *coalgebra homomorphisms*, i.e. the structure-preserving maps between coalgebras of a given covariety, conform directly to an iteration scheme for computation. Thus coalgebras afford a mechanism to prescribe specific control structure for a computation and to communicate this structure to program analysis and translation software.

For the covariety *Stream*, the related iteration scheme is linear search. For more complex covarieties, the iteration schemes are more specialized, including algorithm schemes such as binary search and shift-reduce parsing.

A coalgebra homomorphism is composed of two parts: a coalgebra specification, such as *intseq* in Example 2.1, and a *control* that selects among the projectors of the coalgebra. The body of a control has the form of a conditional or a case expression. A control and a coalgebra specification are combined by an *DUALITY* combinator, *cohom*, suitably specialized to a sort of the covariety. This forms a limit of the specified coalgebra, determined by the control. It is, of course, necessary to confirm that such limits exist, in each case.

Example 2.2 : Sequential search
We shall define a generic sequential search function, give a necessary and sufficient condition for its termination, and give a hypothetical rule of logic to conclude a property of a search. Let a be a type, $p : a \to bool$ and $r : a \to a$. Define a sequential search combinator, *while*, by

$$\begin{aligned} while(p, r) \doteq cohom[str]Stream\ &\{c := a;\ \$shd := id_a,\ \$stl := r\} \\ &(\lambda x.\ \textbf{let}\ u = \$shd\, x\ \textbf{in} \\ &\qquad \textbf{if}\ \neg p\ u\ \textbf{then}\ u\ \textbf{else}\ \$stl\, x) \end{aligned}$$

The control is expressed as a lambda abstraction enclosed in parentheses. The expressions on the arms of a conditional (or case expression) that forms the return expression of the control must be applications of projectors of the coalgebraic variety, or as in this example, identifiers bound to such applications in a local definition. A control expression should not be confused with a function declaration; in particular, the types of the expressions on the arms of a conditional or case are not all of a common type.

A function composed with $cohom[str]$ satisfies a set of conditional equations such as the ones given below for the sequential search combinator:

$$\begin{aligned} p\ x = tt \quad &\Rightarrow \quad while(p, r)\ x = while(p, r)\ (r\ x) \\ p\ x = f\!f \quad &\Rightarrow \quad while(p, r)\ x = id_a\ x \end{aligned}$$

The right-hand sides of the equations are formed by substitution into the control expression. The bindings of the projectors *$shd* and *$stl* are taken from the coalgebra declaration, and in addition, the defined combinator is recursively applied to every projector expression whose codomain type has been specified to be the carrier. In this example, the declaration $\$stl : c$ in the signature dictates that $while(p, r)$ is applied to $(r\ x)$, gotten from the binding of *$stl*. It is not misleading to imagine the combinator expressions of *DUALITY* translated in this way into recursive function definitions in a conventional language. However, the patterns of recursion so obtained are rigidly constrained to tail recursion.

Of course, it is useful to have a more compact declaration for such a useful combinator as $while(\_, \_)$ and it is often made a language primitive. We have used it here as the simplest illustration of coalgebraic program construction with explicit control. Before leaving the example, we should call the reader's attention to another aspect of the coalgebraic declaration. The data transformation and the control are separated, and each is a first-class entity in *DUALITY*. The data transformation is fully specified by the coalgebra, which could be used in other declarations with a different control. The control specification could be used with other *Stream* coalgebras.

3 Finite-state reactive systems

Finite-state systems are naturally modeled by multi-sorted coalgebras. The states of a system correspond to sorts of a coalgebra; the carrier in each sort is comprised of the state variables, and the projectors in each sort are the possible reactions in the corresponding state. Many of these reactions take the system to another state. Traditional functional programming languages have not been easy to use in describing reactive systems because the sequences of possible reactions often seem to require complex mutual recursion for their specification. Formulating a reactive system as a coalgebra is easy because the use of multiple sorts provides a natural and detailed structure for the specification.

We shall illustrate the technique with an example of a synchronous reactive system previously used to illustrate programming in *Esterel* [BG88].

Example 3.1 : The Reflex game
The Reflex game is a coin-operated machine on which a player measures the time constant of her reflexes. After depositing a coin to start the game, she can depress a Ready button to signify that she is prepared to start a trial. When she receives a Go signal from the machine, she depresses a Stop button as quickly as she can. The machine times her response in several trials, then displays the average response time. There are several illegal moves that must be accounted for. If the Stop button is depressed after the player is ready but before Go has been signaled, this action is interpreted as cheating and terminates the game. If either the Ready or the Stop button is depressed when it is not expected, a warning bell sounds, but the game is not interrupted. A coin drop always restarts the game, even when this event occurs during the progress of a previous game.

The game also depends upon timing signals emitted by a clock. Clock ticks must be counted to measure the player's latency. Also, the Go signal is emitted after a randomly determined number of clock ticks following depression of the Ready button by the player. And if a player fails to respond within a predetermined interval when a response is expected, the game times out.

The events that the machine must react to are a coin drop, depression of the Ready and Stop buttons, and ticks of the clock. We assume that these events never occur exactly simultaneously, or that they can be separated in a sequence.

Analysis of the Reflex game shows that the machine can be described as having five major states:

quiet,	when no game is in progress,
start,	when awaiting a Ready event to start a trial,
wait,	when the player is awaiting a Go signal from the machine,
react,	when he machine awaits a Stop event,
end,	when the machine pauses to display the response time of the player.

The machine responds to events differently in each of these five states. Some of the responses are transitions from one state to another. Figure 1 is a state transition diagram for the reflex game machine.

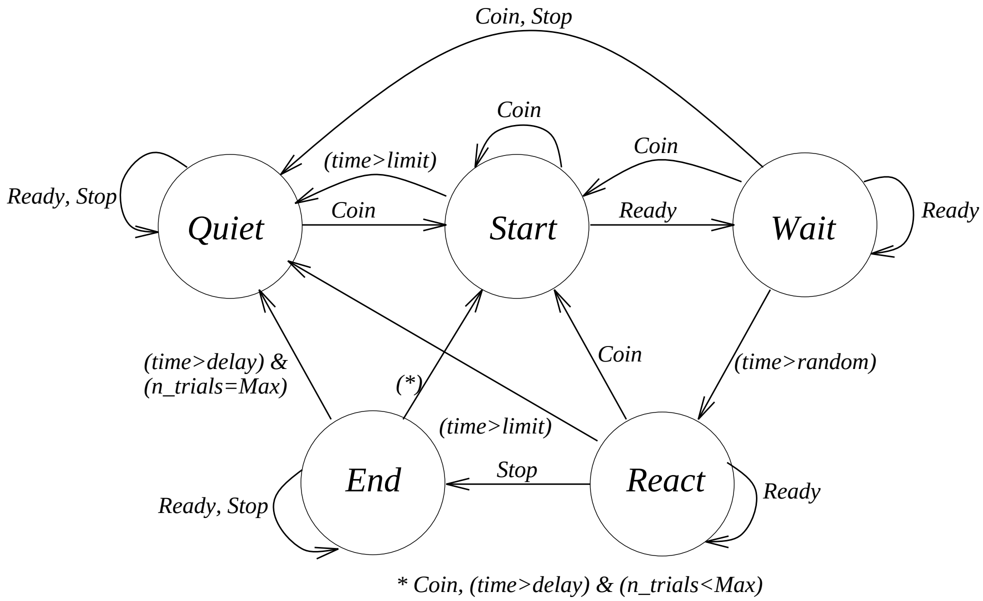

Figure 1—Major states of the Reflex game

In the solution of this problem as an *Esterel* program, the states of the game are not manifest but are implicit in the control. The control consists of a nested loop structure, triggered by events, that takes the machine through the possible sequences of state transitions. Although *Esterel* provides intuitive syntax for coding even-driven nested loop structures, it is still challenging to get them right. This represents the state-of-the-art in programming reactive systems.

The Reflex game can be modeled by a multi-sorted coalgebra. We associate a separate sort with each of the major states of the game. In each of these states, we identify the possible reactions and name them. The reactions in each state become the projectors of the corresponding sort. The codomain type of each projector is the carrier that corresponds to the game state to which the reaction leads. No explicit recursion or iteration is involved in programming the game in this way.

The output of the Reflex game will be modeled as a sequence of states. A state will include state variables and output signals produced by a reaction. However, these are details that will appear in a coalgebra for the game. None of these details are manifested in a covariety. A signature for the covariety is:

$$
\begin{array}{l}
\textbf{cosig}\ \mathit{Reflex}(a)\ \{\textbf{type}\ q, s, w, r, t; \\
\qquad\qquad \begin{array}{lcl}
\mathit{quiet}/q & : & \{\$\mathit{coin} : s,\ \$\mathit{noop} : q\}, \\
\mathit{start}/s & : & \{\$\mathit{reveal} : a,\ \$\mathit{ready} : w,\ \$\mathit{renew} : s,\ \$\mathit{warn} : s,\ \$\mathit{timeout} : q,\ \$\mathit{tick} : s\}, \\
\mathit{wait}/w & : & \{\$\mathit{reveal} : a,\ \$\mathit{renew} : s,\ \$\mathit{warn} : w,\ \$\mathit{abort} : q,\ \$\mathit{tick} : w,\ \$\mathit{go} : r\}, \\
\mathit{react}/r & : & \{\$\mathit{reveal} : a,\ \$\mathit{react} : t,\ \$\mathit{renew} : s,\ \$\mathit{warn} : r,\ \$\mathit{tick} : r,\ \$\mathit{timeout}\, q\}, \\
\mathit{end}/t & : & \{\$\mathit{reveal} : a,\ \$\mathit{warn} : t,\ \$\mathit{renew} : s,\ \$\mathit{tick} : t,\ \$\mathit{tock} : s,\ \$\mathit{finish} : q\}\}
\end{array}
\end{array}
$$

The projectors *\$reveal* do not correspond to state transitions of the game, but are instead actual projections of the machine state.

To specify the game, we shall specify a *Reflex*-coalgebra, binding data transformation functions associated with state transitions to each of the projector symbols. To determine all trajectories of play, we shall generate a game tree, which will be codata, of course. To simulate a game, we shall interpret a sequence of externally caused events (coin drops, clock ticks and button pushes) as control for the projectors in each state.

Minor states of the game are determined by the values of three integer-valued state components. These can be packaged as fields of a record type,

$$\mathbf{record}\ State\{time,\ total\_time,\ trial\_number\ :\ int\};$$

Further, there are signals delivered to the actuators that implement the machine. These unvalued signals can be represented by a set of elements of an enumerated type,

$$\begin{aligned}\mathbf{type}\ Signals = \mathbf{set\ of}\ [&game\_over\_on \mid game\_over\_off \mid go\_on \mid go\_off \mid \\ &tilt\_on \mid tilt\_off \mid ring\_bell \mid bump\_random];\end{aligned}$$

There is one integer-valued signal which sends values to the display. The state components and the signals are conveniently packaged as fields of a record type. We declare

$$\mathbf{record}\ Game\{state\ :\ State;\ sigs\ :\ Signals;\ display\ :\ int\};$$

It is convenient to define a pair of constants of type *Game*,

$$\begin{aligned}
&\mathbf{def}\ initial_game = \{state := \{time := 0;\ total_time := 0;\ trial_number := 0\};\\
&\qquad\qquad\qquad\qquad sigs := [game_over_off,\ go_off,\ tilt_off];\\
&\qquad\qquad\qquad\qquad display := 0\};\\
&\mathbf{def}\ tilt_game = \{\{state := \{time := 0;\ total_time := 0;\ trial_number := 0\};\\
&\qquad\qquad\qquad\quad sigs := [game_over_on,\ go_off,\ tilt_on];\\
&\qquad\qquad\qquad\quad display := 0\};
\end{aligned}$$

There are also three integer constants, *Time_limit*, *Delay* and *Max_trials*, and a stream, *random*, which is a randomly generated sequence of positive integers of bounded size, supplied by the machine. As a notational abbreviation, let $R \oplus \{X := e\}$ denote the record whose fields have the values of the corresponding fields in the record R, except for field X, which has the value of e.

The next task is to define a coalgebra by specifying the projectors of each sort. These correspond to the possible transitions from each state.

$$
\begin{aligned}
\mathit{reflex} \doteq\ & \textbf{coalgebra}\ \mathit{Reflex}\ \{q, s, w, r, t := \mathit{state};\\
& \quad \mathit{quiet} : \{\$\mathit{coin} := \lambda s.\ \mathit{initial\_game},\\
& \qquad\qquad \$\mathit{noop} := \mathit{id}_{\mathit{Game}}\},\\
& \quad \mathit{start} : \{\$\mathit{reveal} := \lambda s.\ s.\mathit{state},\\
& \qquad\qquad \$\mathit{ready} := \lambda s.\ s \oplus \{s.\mathit{state} \oplus \{\mathit{time} := 0\},\ \mathit{sigs} := [\mathit{bump\_random}]\},\\
& \qquad\qquad \$\mathit{renew} := \lambda s.\ \mathit{initial\_game},\\
& \qquad\qquad \$\mathit{warn} := \lambda s.\ s \oplus \{\mathit{sigs} := [\mathit{ring\_bell}]\},\\
& \qquad\qquad \$\mathit{timeout} := \lambda s.\ \mathit{tilt\_game},\\
& \qquad\qquad \$\mathit{tick} := \lambda s.\ s \oplus \{s.\mathit{state} \oplus \{\mathit{time} := s.\mathit{time} + 1\}\}\},\\
& \quad \mathit{wait} : \{\$\mathit{reveal} := \lambda s.\ s.\mathit{state},\\
& \qquad\qquad \$\mathit{renew} := \lambda s.\ \mathit{initial\_game},\\
& \qquad\qquad \$\mathit{warn} := \lambda s.\ s \oplus \{\mathit{sigs} := [\mathit{ring\_bell}]\},\\
& \qquad\qquad \$\mathit{abort} := \lambda s.\ \mathit{tilt\_game},\\
& \qquad\qquad \$\mathit{tick} := \lambda s.\ s \oplus \{s.\mathit{state} \oplus \{\mathit{time} := s.\mathit{time} + 1\}\},\\
& \qquad\qquad \$\mathit{go} := \lambda s.\ s \oplus \{s.\mathit{state} \oplus \{\mathit{time} := 0\};\ \mathit{sigs} := [\mathit{go\_on}]\}\},\\
& \quad \mathit{react} : \{\$\mathit{reveal} := \lambda s.\ s.\mathit{state},\\
& \qquad\qquad \$\mathit{react} := \lambda s.\ \{\mathit{state} := \{\mathit{time} := 0;\\
& \qquad\qquad\qquad\qquad\qquad \mathit{total\_time} := s.\mathit{state}.\mathit{total\_time} + s.\mathit{state}.\mathit{time};\\
& \qquad\qquad\qquad\qquad\qquad \mathit{trial\_number} := s.\mathit{state}.\mathit{trial\_number} + 1\};\\
& \qquad\qquad\qquad\qquad \mathit{sigs} := [\mathit{go\_off}];\ \mathit{display} := s.\mathit{state}.\mathit{time}\},\\
& \qquad\qquad \$\mathit{renew} := \lambda s.\ \mathit{initial\_game},\\
& \qquad\qquad \$\mathit{warn} := \lambda s.\ s \oplus \{\mathit{sigs} := [\mathit{ring\_bell}]\},\\
& \qquad\qquad \$\mathit{tick} := \lambda s.\ s \oplus \{s.\mathit{state} \oplus \{\mathit{time} := s.\mathit{state}.\mathit{time} + 1\}\},\\
& \qquad\qquad \$\mathit{timeout} := \lambda s.\ \mathit{tilt\_game}\},\\
& \quad \mathit{end} : \{\$\mathit{reveal} := \lambda s.\ s.\mathit{state},\\
& \qquad\qquad \$\mathit{renew} := \lambda s.\ \mathit{initial\_game},\\
& \qquad\qquad \$\mathit{warn} := \lambda s.\ s \oplus \{\mathit{sigs} := [\mathit{ring\_bell}]\},\\
& \qquad\qquad \$\mathit{tick} := \lambda s.\ s \oplus \{s.\mathit{state} \oplus \{\mathit{time} := s.\mathit{state}.\mathit{time} + 1\}\},\\
& \qquad\qquad \$\mathit{tock} := \lambda s.\ \{\mathit{state} := \{\mathit{time} := 0\},\\
& \qquad\qquad\qquad \mathit{display} := 0\},\\
& \qquad\qquad \$\mathit{finish} := \lambda s.\ s \oplus \{\mathit{sigs} := [\mathit{game\_over\_on}];\\
& \qquad\qquad\qquad\qquad \mathit{display} := s.\mathit{state}.\mathit{total\_time}/\mathit{Max\_trials}\}\}\}
\end{aligned}
$$

A generator composed from this coalgebra, for instance, $\mathit{gen}[\mathit{quiet}]\,\mathit{reflex} : \mathit{Game} \to \mathit{quiet}$, when applied to a value of the state variables generates, in response to demand, an infinite game tree rooted on the quiescent game state. Paths in the game tree incorporate all major-state transitions allowed by the rules of the game, and in addition, some that are not allowed. The game tree includes some paths that do not correspond to feasible trajectories of the actual game, because transitions in the game tree are unconstrained by conditions on the state variables that govern the progress of the actual game (i.e. the rules of the game).

To obtain a function that accurately simulates the game, the coalgebra must be composed with a control that responds to input events and reads state variables to determine a game path. The control is defined as a cluster of five expressions, one for each sort, as the game's response to events depends upon the major state that it occupies.

Since the codomain of the simulation is a function type, each component of the control is a curried abstraction on two arguments. Since the final result is a stream of game states, the body of each component of the control has the form of a *Stream*-generator. The *$shd* projector defined in each game state translates each of the possible input events into a state transition event. In some cases, the translation is conditioned by the elapsed time recorded in a game state. Here is a definition of the control:

$$
\begin{array}{ll}
\mathbf{def}\ \mathit{transition} \doteq & (\mathit{quiet}: (\lambda s.\ \mathit{gen}[\mathit{str}]\ \mathit{Stream}\{c := \mathit{str}(\mathit{event});\\
& \quad \mathit{\$shd} := \lambda es.\ \mathbf{let}\ e = \mathit{Shd}\ es\ \mathbf{in}\\
& \qquad \mathbf{case}\ e\ \mathbf{of}\\
& \qquad\quad \mathit{Coin} \Rightarrow \mathit{\$coin}\ s\\
& \qquad \mid\ \mathit{Ready} \Rightarrow \mathit{\$noop}\ s\\
& \qquad \mid\ \mathit{Stop} \Rightarrow \mathit{\$noop}\ s\\
& \qquad \mid\ \mathit{Tick} \Rightarrow \mathit{\$noop}\ s\\
& \qquad \mathbf{end}\\
& \quad \mathit{\$stl} := \mathit{Stl}\}),\\
& \mathit{start}: (\lambda s.\ \mathit{gen}[\mathit{str}]\ \mathit{Stream}\{c := \mathit{str}(\mathit{event});\\
& \quad \mathit{\$shd} := \lambda es.\ \mathbf{let}\ e = \mathit{Shd}\ es\ \mathbf{in}\\
& \qquad \mathbf{case}\ e\ \mathbf{of}\\
& \qquad\quad \mathit{Coin} \Rightarrow \mathit{\$renew}\ s\\
& \qquad \mid\ \mathit{Ready} \Rightarrow \mathit{\$ready}\ s\\
& \qquad \mid\ \mathit{Stop} \Rightarrow \mathit{\$warn}\ s\\
& \qquad \mid\ \mathit{Tick} \Rightarrow \mathbf{let}\ v = \mathit{\$reveal}\ s\ \mathbf{in}\\
& \qquad\qquad\qquad \mathbf{if}\ v.\mathit{time} < \mathit{Time\_limit}\ \mathbf{then}\ \mathit{\$tick}\ s\\
& \qquad\qquad\qquad \mathbf{else}\ \mathit{\$timeout}\ s\\
& \qquad \mathbf{end}\\
& \quad \mathit{\$stl} := \mathit{Stl}\}),\\
& \mathit{wait}: (\lambda s.\ \mathit{gen}[\mathit{str}]\ \mathit{Stream}\{c := \mathit{str}(\mathit{event});\\
& \quad \mathit{\$shd} := \lambda es.\ \mathbf{let}\ e = \mathit{Shd}\ es\ \mathbf{in}\\
& \qquad \mathbf{case}\ e\ \mathbf{of}\\
& \qquad\quad \mathit{Coin} \Rightarrow \mathit{\$renew}\ s\\
& \qquad \mid\ \mathit{Ready} \Rightarrow \mathit{\$warn}\ s\\
& \qquad \mid\ \mathit{Stop} \Rightarrow \mathit{\$abort}\ s\\
& \qquad \mid\ \mathit{Tick} \Rightarrow \mathbf{let}\ v = \mathit{\$reveal}\ s\ \mathbf{in}\\
& \qquad\qquad\qquad \mathbf{if}\ v.\mathit{time} < \mathit{random}\ \mathbf{then}\ \mathit{\$tick}\ s\\
& \qquad\qquad\qquad \mathbf{else}\ \mathit{\$go}\ s\\
& \qquad \mathbf{end}\\
& \quad \mathit{\$stl} := \mathit{Stl}\}),
\end{array}
$$

$$
\begin{array}{l}
\mathit{react} : (\lambda s.\ \mathit{gen}[\mathit{str}]\ \mathit{Stream}\{c := \mathit{str}(\mathit{event}); \\
\qquad\qquad \$\mathit{shd} := \lambda es.\ \mathbf{let}\ e = \mathit{Shd}\ es\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{case}\ e\ \mathbf{of} \\
\qquad\qquad\qquad\quad \mathit{Coin} \Rightarrow \$\mathit{renew}\, s \\
\qquad\qquad\qquad \mid\ \mathit{Ready} \Rightarrow \$\mathit{warn}\ s \\
\qquad\qquad\qquad \mid\ \mathit{Stop} \Rightarrow \$\mathit{react}\, s \\
\qquad\qquad\qquad \mid\ \mathit{Tick} \Rightarrow \mathbf{let}\ v = \$\mathit{reveal}\, s\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\quad \mathbf{if}\ v.\mathit{time} < \mathit{Time\_limit}\ \mathbf{then}\ \$\mathit{tick}\, s \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \$\mathit{timeout}\, s \\
\qquad\qquad\qquad \mathbf{end} \\
\qquad\qquad \$\mathit{stl} := \mathit{Stl}\}), \\
\mathit{end} : (\lambda s.\ \mathit{gen}[\mathit{str}]\ \mathit{Stream}\{c := \mathit{str}(\mathit{event}); \\
\qquad\qquad \$\mathit{shd} := \lambda es.\ \mathbf{let}\ e = \mathit{Shd}\ es\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{case}\ e\ \mathbf{of} \\
\qquad\qquad\qquad\quad \mathit{Coin} \Rightarrow \$\mathit{renew}\, s \\
\qquad\qquad\qquad \mid\ \mathit{Ready} \Rightarrow \$\mathit{warn}\ s \\
\qquad\qquad\qquad \mid\ \mathit{Stop} \Rightarrow \$\mathit{warn}\ s \\
\qquad\qquad\qquad \mid\ \mathit{Tick} \Rightarrow \mathbf{let}\ v = \$\mathit{reveal}\, s\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\quad \mathbf{if}\ v.\mathit{time} < \mathit{Delay}\ \mathbf{then}\ \$\mathit{tick}\, s \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathbf{let}\ v = \$\mathit{reveal}\, s\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{if}\ v.\mathit{trial\_number} < \mathit{Max\_trials} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \$\mathit{tock}\, s \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \$\mathit{finish}\, s \\
\qquad\qquad\qquad \mathbf{end} \\
\qquad\qquad \$\mathit{stl} := \mathit{Stl}\}))
\end{array}
$$

A simulator for the Reflex game is the function

$$\mathit{cohom}[\mathit{quiet}]\ \mathit{reflex}\ \mathit{transition}\ :\ \mathit{Game} \rightarrow \mathit{str}(\mathit{event}) \rightarrow \mathit{str}(\mathit{Game}).$$

□

The domain constraints needed for the example of the Reflex game can be established by structural (Hindley-Milner) type checking. Note that the simulator is not expected to terminate, in the usual sense, but rather to make finite progress in response to each external event that it receives. Finite progress is assured by observing (in the control code) that every state transition event occurs in response to an external event. There are no spontaneous state transitions, and therefore no infinite sequence of spontaneous transitions that could block finite progress.

Programming the Reflex game in terms of coalgebras is straightforward once the type of the solution and of the component functions has been determined. The structure of the covariety morphisms does not allow guesswork.

3.1 A verification logic for the Reflex game simulator

One of the most significant advantages of formulating a finite-state systems such as the Reflex game simulator as a coalgebra morphism is that the coalgebraic structure induces a complementary deductive logic in which properties of the system can be proved by coinduction. As we shall see, the coinduction rules induced by the coalgebraic structure provide a fine-grained decomposition of proof obligations that must be discharged to establish a conjectured property. We believe this structure will make verification significantly easier by removing most of the guesswork. We expect it to be amenable to the application of automatic proof discovery methods.

Corresponding to each carrier in the coalgebra signature declaration, we shall declare a predicate symbol whose interpretation will characterize a specific property in the major state (or sort) to which the carrier is bound. For the reflex game example, these will be unary[2] predicates, each relating an external event stream and a game state. The game state will be a minor state of the major state that the predicate describes.

A coinduction rule for a coalgebra is formulated as a sequent clause. In the consequent are clauses for each sort of a multi-sorted coalgebra; in the antecedent are sets of hypotheses for each sort. The hypotheses for a given sort will correspond one-for-one to the projectors definedF for that sort. The structure of a coinduction rule is induced directly by the signature of a coalgebra.

Each clause in the consequent of a coinduction rule extends the interpretation of a predicate to encompass all of the states of the corresponding sort in a potentially infinite tree or sequence. There will be one such clause for each sort of a multi-sorted coalgebra, allowing characterization of a property specified at each of the major states of a finite-state model, throughout all minor states that are reachable from a given initial state.

Each individual clause of an antecedent implies the transfer of a property under a projection. For instance, referring to the Reflex game, a clause that implies the transfer of a property via the transition *\$ready* is:

$$\forall u : s,\ es : str(event).\, S(u) \Rightarrow W(\$ready\ u)$$

where S and W are the predicate symbols associated with sorts s and w, respectively. Upon substituting the binding for the transition *\$ready* as given in the declaration of the coalgebra *reflex*, the clause becomes

$$\forall u : s,\ es : str(event).\, S(u) \Rightarrow W(u \oplus \{u.state \oplus \{time := 0\}\})$$

When the coinduction rule is for a general coalgebra morphism (a *cohom*), each hypothetical implication must be qualified by a guard for the transition that can be read from the declaration of the control for the morphism. Again referring to the reflex game, the clause above, extended as a hypothetical clause for the simulator, becomes

[2] In this example, we chose not to include an initial state as a parameter of the predicate because the initial state is to be fixed in each game. More generally, a two-place predicate, parametric on an initial state, would allow the simulator to be characterised as a function from an arbitarily specified initial state to the ensuing behavior.

$$\forall u : s,\ es : str(event).\,(transition.start\ u\ (Shd\ es) = \$ready\ u) \Rightarrow$$
$$S(u) \Rightarrow W(u \oplus \{u.state \oplus \{time := 0\}\})$$

Further substituting the guard clause by its binding in the declaration of *transition*, the hypothetical implication now relates the transition to the occurrence of an external event:

$$\forall u : s,\ es : str(event).\,(Shd\ es = Ready) \Rightarrow S(u) \Rightarrow W(u \oplus \{u.state \oplus \{time := 0\}\})$$

Following this recipe, we find the following coinduction rule for the Reflex game simulator:

$$
\begin{array}{l}
\forall u : q,\ es : str(event).\,(Shd\ es = Coin) \Rightarrow Q(u) \Rightarrow S(initial\_state)\\
\forall u : q,\ es : str(event).\,(Shd\ es = Ready) \Rightarrow Q(u) \Rightarrow Q(u)\\
\forall u : q,\ es : str(event).\,(Shd\ es = Stop) \Rightarrow Q(u) \Rightarrow Q(u)\\
\forall u : q,\ es : str(event).\,(Shd\ es = Tick) \Rightarrow Q(u) \Rightarrow Q(u)\\[6pt]
\forall u : s,\ es : str(event).\,(Shd\ es = Coin) \Rightarrow S(u) \Rightarrow S(initial\_state)\\
\forall u : s,\ es : str(event).\,(Shd\ es = Ready) \Rightarrow S(es, u) \Rightarrow W(u \oplus \{u.state \oplus \{time := 0\}\})\\
\forall u : s,\ es : str(event).\,(Shd\ es = Stop) \Rightarrow S(es, u) \Rightarrow S(u \oplus \{sigs := [ring\_bell]\})\\
\forall u : s,\ es : str(event).\,(Shd\ es = Tick) \Rightarrow (u.state.time < Time\_limit) \Rightarrow\\
\qquad\qquad\qquad\qquad S(u) \Rightarrow S(u \oplus \{u.state \oplus \{time := s.time + 1\}\})\\
\forall u : s,\ es : str(event).\,(Shd\ es = Tick) \Rightarrow (u.state.time \geq Time\_limit) \Rightarrow\\
\qquad\qquad\qquad\qquad S(u) \Rightarrow Q(tilt\_game)\\[6pt]
\forall u : w,\ es : str(event).\,(Shd\ es = Coin) \Rightarrow W(u) \Rightarrow S(initial\_state)\\
\forall u : w,\ es : str(event).\,(Shd\ es = Ready) \Rightarrow W(u) \Rightarrow W(u \oplus \{sigs := [ring\_bell]\})\\
\forall u : w,\ es : str(event).\,(Shd\ es = Stop) \Rightarrow W(u) \Rightarrow Q(tilt\_game)\\
\forall u : w,\ es : str(event).\,(Shd\ es = Tick) \Rightarrow (u.state.time < C\_random) \Rightarrow\\
\qquad\qquad\qquad\qquad W(u) \Rightarrow W(u \oplus \{u.state \oplus \{time := s.time + 1\}\})\\
\forall u : w,\ es : str(event).\,(Shd\ es = Tick) \Rightarrow (u.state.time \geq C\_random) \Rightarrow\\
\qquad\qquad\qquad\qquad W(u) \Rightarrow W(u \oplus \{u.state \oplus \{time := 0\};\ sigs := [go\_on]\})\\[6pt]
\forall u : r,\ es : str(event).\,(Shd\ es = Coin) \Rightarrow R(u) \Rightarrow S(initial\_state)\\
\forall u : r,\ es : str(event).\,(Shd\ es = Ready) \Rightarrow R(u) \Rightarrow R(u \oplus \{sigs := [go\_on]\})\\
\forall u : r,\ es : str(event).\,(Shd\ es = Stop) \Rightarrow R(u) \Rightarrow\\
\qquad\qquad\qquad\qquad S(u \oplus \{state \oplus \{time := 0;\\
\qquad\qquad\qquad\qquad\qquad\qquad total\_time := s.state.total\_time + s.state.time;\\
\qquad\qquad\qquad\qquad\qquad\qquad trial\_number := s.state.trial\_number + 1\};\\
\qquad\qquad\qquad\qquad\qquad signs := [go\_off];\ display := s.state.time\})\\
\forall u : r,\ es : str(event).\,(Shd\ es = Tick) \Rightarrow (u.state.time < Time\_limit) \Rightarrow\\
\qquad\qquad\qquad\qquad R(u) \Rightarrow R(u \oplus \{u.state \oplus \{time := u.time + 1\}\})\\
\forall u : r,\ es : str(event).\,(Shd\ es = Tick) \Rightarrow (u.state.time \geq Time\_limit) \Rightarrow\\
\qquad\qquad\qquad\qquad R(u) \Rightarrow Q(tilt\_game)
\end{array}
$$

$$
\begin{array}{l}
\forall u : t,\ es : str(event).\,(Shd\,es = Coin) \Rightarrow T(u) \Rightarrow S(initial\_state) \\
\forall u : t,\ es : str(event).\,(Shd\,es = Ready) \Rightarrow T(u) \Rightarrow T(u \oplus \{sigs := [ring\_bell]\}) \\
\forall u : t,\ es : str(event).\,(Shd\,es = Stop) \Rightarrow T(u) \Rightarrow T(u \oplus \{sigs := [ring\_bell]\}) \\
\forall u : t,\ es : str(event).\,(Shd\,es = Tick) \Rightarrow (u.state.time < Delay) \Rightarrow \\
\qquad\qquad T(u) \Rightarrow T(u \oplus \{u.state \oplus \{time := s.time + 1\}\}) \\
\forall u : t,\ es : str(event).\,(Shd\,es = Tick) \Rightarrow (u.state.time \geq Delay) \wedge (u.trial\_number < Max\_trials) \Rightarrow \\
\qquad\qquad T(u) \Rightarrow S(u \oplus \{state \oplus \{time := 0\} \\
\qquad\qquad\qquad display := 0\})) \\
\forall u : t,\ es : str(event).\,(Shd\,es = Tick) \Rightarrow (u.state.time \geq Delay) \wedge (u.trial\_number \geq Max\_trials) \Rightarrow \\
\qquad\qquad T(u) \Rightarrow Q(u \oplus \{sigs := [game\_over\_on]; \\
\qquad\qquad\qquad display := s.state.total\_time/Max\_trials\})) \\
\hline
\forall u_0 : Game,\ es : str[event].\,Q(u_0) \Rightarrow \Box Q(cohom[quiet]\ reflex\ transition\ u_0\ es) \\
\forall u_0 : Game,\ es : str[event].\,S(u_0) \Rightarrow \Box S(cohom[start]\ reflex\ transition\ u_0\ es) \\
\forall u_0 : Game,\ es : str[event].\,W(u_0) \Rightarrow \Box W(cohom[wait]\ reflex\ transition\ u_0\ es) \\
\forall u_0 : Game,\ es : str[event].\,R(u_0) \Rightarrow \Box R(cohom[react]\ reflex\ transition\ u_0\ es) \\
\forall u_0 : Game,\ es : str[event].\,T(u_0) \Rightarrow \Box T(cohom[end]\ reflex\ transition\ u_0\ es)
\end{array}
$$

The temporal operator $\Box$ (*always*) in the consequent formulas expresses precisely the sense in which the predicate over game states is extended by coinduction to a predicate over a stream of game states.

The textual extent of this coinduction rule is imposing, but keep in mind that it is amenable to mechanical calculation from the three parts of the formal declaration of the Reflex game simulator: the signature, the coalgebra specification and the control. The significance of the coinduction rule for verification is that given a conjectured proposition of a property of the potentially infinite behaviors of the simulator as an intended consequent, the rule yields a finite set of finitary propositions that must be discharged to prove the proposition. It accomplishes a logical destructuring that is essential to constructing a formal proof, and it does so in a way that is amenable to mechanization.

3.1.1 Proving safety properties of the Reflex game

The consequents of the coinduction rule for the Reflex game assert invariant properties of states of the game. These are its so-called *safety* properties.

A simple safety property is that in every state, u,

$$Q_t(u) \dot{\equiv} \frac{C\_random \leq Time\_limit \qquad Delay \leq Time\_limit}{u.state.time \leq Time\_limit}$$

In this clause, the antecedent conditions relate the values of constants of the Reflex game. Without these relations, the property does not hold. To prove this property of a game started in the quiescent state, we formulate the conjectured property as a consequent:

$$
\begin{array}{l}
\forall u_0 : Game,\ es : str[event].\ u_0.state.time \leq Time\_limit \Rightarrow \\
\qquad\qquad \Box(state.time \leq Time\_limit)(cohom[quiet]\ reflex\ transition\ u_0\ es)
\end{array}
$$

for which we seek a proof by Reflex game coinduction.

In the coinduction rule stated in the preceding section, choose $Q \equiv S \equiv W \equiv R \equiv T \equiv Q_t$ and attempt to discharge each of the antecedents of the rule. Most of the antecedent clauses discharge trivially, either because they do not refer to the *time* parameter explicitly or they set it to zero. There are five antecedent clauses in which the *time* parameter is incremented, however. Each of these clauses is guarded by a condition that *time* is strictly less than one of the constants *Time_limit, C_random* or *Delay.* Using the antecedent condition relating the latter two constants to *Time_limit*, it can be established that the implicand in each of the clauses is satisfied and the clause is discharged. Thus the property is proved to hold for every game state reachable from an initial state that satisfies the property, by condinduction.

A related safety property that can be established is

$$Q_{tt}(u) \doteq u.state.total\_time \leq Max\_trials * Time\_limit$$

We can also state safety properties consequent to a restriction on the external event stream. For instance we might assert

$$\forall u_0 : Game,\ es : str(event).\ \Box(next\ event \neq Coin) es \Rightarrow Q(u_0) \wedge (u_0.state.time = 0) \Rightarrow$$
$$\Box(state.time = 0)(cohom[quiet]\ reflex\ transition\ u_0\ es)$$

for a game started from the quiescent state. To prove this assertion, we must infer from the temporal logic assertion, $\Box(next\ event \neq Coin)\ es$, the proposition $\forall es : str(event).\ Shd\ es \neq Coin$. The derived proposition can then be used to restrict the domain of antecedent clauses in a proof using the Reflex game coinduction rule.

3.2 Liveness properties of the Reflex game

Liveness properties, which assert that a state or state sequence with the property is eventually reached in every unfolding of the game, require for their proof a specific measure of progress. A monotonically increasing count of any external event can provide a suitable measure, provided that the timing event occurs *almost everywhere* in the event stream. Occuring almost everywhere means that at every point in the stream, the next timing event will occur after at most a finite number of non-timing events. The almost everywhere restriction assures that the observation of advancing time is never obscured by an infinite stream of non-timing events.

In the Reflex game, intuition tells us that the *Tick* event is a reasonable choice for the timing event. We assume that it occurs almost everywhere in every possible stream of external events. This assumption is essential; it cannot be proved from weaker assumptions.

A liveness property we should like to prove is that every game started by a coin drop eventually terminates. We adopt for our definition of termination that either (a) the game enters the quiescent state or (b) another coin is dropped. Note that we have no direct characterization of the quiescent state (or any other major state) in terms of the program's state variables. The *time* attribute of the game state is only locally monotonic. It is monotonic with respect to transitions from any state to itself, but is reset to zero on many transitions from one state to another. Although the attribute *trial_number* is monotonic throughout all transitions that do not enter the quiescent state, we cannot use this attribute to characterize the quiescent state, because it is not incremented immediately upon leaving the quiescent state.

States can be characterized by sets of initial sequences of event streams, but this is not very convenient. It is more convenient to introduce a pseudo-variable, *game_over : bool*, which is set to *true* when the display event *game_over_on* is signalled, and to *false* when *game_over_off* is signalled. The quiescent state is then characterized by a *true* value of *game_over*.

The property we have described can be decomposed into two independent clauses,

$$\begin{aligned}
&\forall u_0 : Game,\ es : str(event).\ \Box(next\ event \neq Coin)es \Rightarrow \\
&\qquad\qquad \Diamond(state.game_over = true)(cohom[quiet]\ reflex\ transition\ u_0\ es) \\
&\forall u_0 : Game,\ es : str(event).\ \Box(next\ event \neq Coin)es \Rightarrow (u_0.state.game_over = true) \Rightarrow \\
&\qquad\qquad \Box(state.game_over = true)(cohom[quiet]\ reflex\ transition\ u_0\ es)
\end{aligned}$$

The second clause is a safety property. We shall attend to the first clause.

The most successful way yet developed to verify temporal properties of a finite state system uses model checking of temporal logic formulas[EC82, CGL92]. The safety and liveness properties of the Reflex game example can obviously be verified by symbolic model checking. We describe a variant of the stardard technique that uses symbolic inference to check monotonicity properties of state variables over transition paths. We have not yet implemented this method.

An ordered, symbolic binary decision diagram (BDD)[Bry86, Bry92] can be used to construct a proof of a liveness property. Boolean pseudo-variables are introduced to represent the boolean-typed expressions on which local control decisions are based. The nodes of the BDD correspond to major states of the Reflex game, split in cases in which more than one boolean condition controls transitions from the state. The boolean control expressions for each state are identified by inspection of the control specification.

$$\begin{array}{ll}
(start) & x_1 \doteq state.time < Time_limit \\
(wait) & x_2 \doteq state.time < C_random \\
(react) & x_3 \doteq state.time < Time_limit \\
(end) & x_4 \doteq state.time < Delay \\
(end') & x_5 \doteq state.trial_number < Max_trials
\end{array}$$

The pseudo-variables are ordered by $x_1 < x_2 < x_3 < x_4 < x_5$.

A nondeterministic BDD for the Reflex game is shown in Figure 2(a). The solid arcs indicate transitions possible when the value of the controlling pseudo-variable is positive; the dashed arcs represent transitions possible on negative values of the control variables.

Notice that there are multiple positive (or negative) arcs from some nodes. This BDD represents a nondeterministic FSA because transitions of the Reflex machine also depend upon external events which have not been represented in the control expressions bound to pseudo-variables. Note also that in constructing Figure 2(a), transitions that require the *Coin* event have been omitted because the *Coin* event is precluded by the antecedent clause of the liveness assertion. Nondeterminism allows us to represent with the BDD all of the transitions possible with event sequences that are restricted only by the assumptions that *Tick* events occur almost everywhere and *Coin* events are never present. The liveness property that we seek can be proved if we can show that the BDD of Figure 2(a) can be reduced to the single node, **1**.

3.2.1 Reducing a BDD with repeated nodes

We shall describe (informally) how to reduce the BDD of Firgure 2(a), which represents the asserted liveness property of the Reflex game. Notice that this BDD contains paths from ancestor nodes to leaf nodes that carry the same labels. We call these *repeated nodes*. A path from an ancestor to a repeated node occurrence represents a loop in the state transition diagram. To reduce the BDD, each of these paths must be shown to be only finitely extensible as it is elaborated by repeating transitions of the state machine.

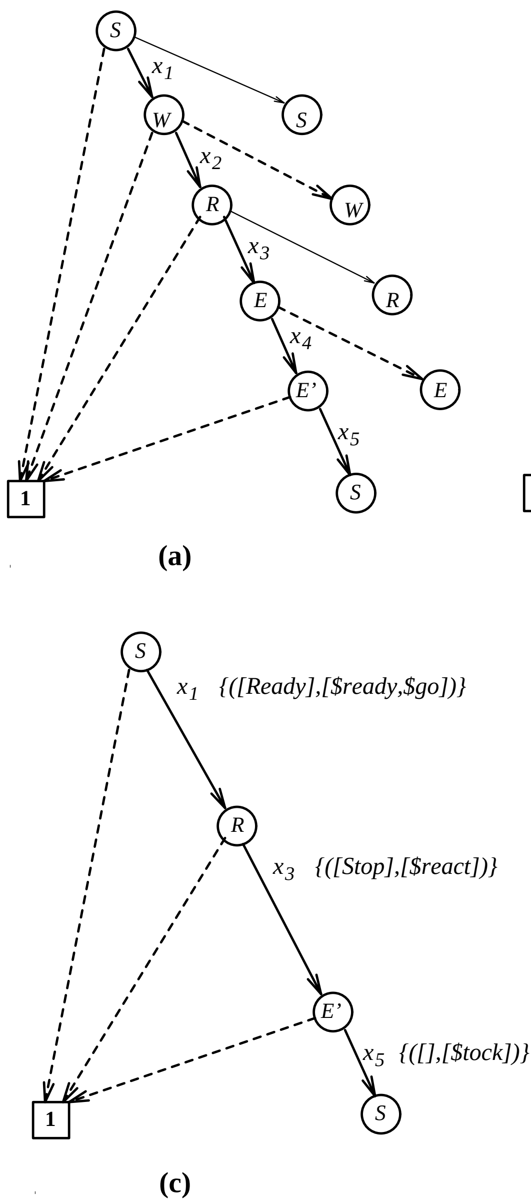

Figure 2

To establish that a path to a repeated node is only finitely extensible, we examine the control expressions bound to the pseudo-variables that label the arcs of the path. In our example, these expressions are less-than inequalities. If we can show that the value of the program variable on the left of the inequality grows to reach the expressed bound after a finite number of repetitions of the path then the corresponding pseudo-variable will eventually become 0. Sufficient conditions for this to occur are that the program variable is (1) monotonic with respect to executions of the transition function that corresponds to the path and (2) increasing almost everywhere in any sequence of repetitions of the path.

Consider the path from the first occurrence of a node labeled *wait* to its repeated occurrence, controlled by pseudo-variable x_2. The program variable that appears in the corresponding inequality is *state.time*. The path controlled by x_2 represents only the transitions *\$warn* and *\$tick*, which are enabled by external events *Ready* and *Tick*, respectively. Inspection of the coalgebra specification *Reflex* reveals that a *\$warn* transition does not change *state.time*, while a *\$tick* transition increments its value. Thus the path satisfies condition (1), monotonicity of the program variable. Furthermore, since the *Tick* event occurs almost everywhere in an external event stream, the *\$tick* transition will occur almost everywhere in a sequence of transitions from state *wait* to itself, thus the path also satisfies condition (2). This argument proves the temporal assertion $\Diamond(x_2 = 0)$.

This allows the BDD of Figure 2(a) to be reduced by removing the repeated occurrence of node *wait* and the arc leading to it. Similar reasoning justifies removal of the repeated occurrence of nodes *ready*, *end*, and the first repeated occurrence of node *start* leaving the BDD depicted in Figure 2(b). The paths not controlled by a pseudo-variable in this BDD are not of interest and can be reduced to single arcs.

In Figure 2(c), the arcs have been labelled with sets of the pairs of event sequences and corresponding transition actions represented by an interpretation of the arc in the *Reflex* coalgebra. The set of transition sequences from the root node to its repeated occurrence is gotten by taking the cartesian product of the sequece of set-valued labels. As the labels on individual arcs are all singletons, so is the composite label, which is $\{([\mathit{Ready}, \mathit{Stop}], [\$\mathit{ready}, \$\mathit{go}, \$\mathit{react}, \$\mathit{tock}])\}$.

We examine the expression x_1, x_3 and x_5 in the context of the path transition sequence. The program variable *state.time* is not monotonic with respect to the transition sequence, hence we cannot conclude that repeated extensions of the path would cause the variables x_1 or x_3 to assume zero values. However, the program variable *state.trial_number* is monotonically increasing over this transition sequence, thus the arc labeled by x_5 and the repeated node *start* can be eliminated. The resulting BDD, depicted in Figure 2(d), is reducible to the singleton node **1** as its canonical form. This constitutes a proof of the conjectured liveness property.

4 Conclusions

We have introduced a of functional programming notation that does not depend upon explicit recursion in definitions but uses instead the structure of signature coalgebras. The important contributions of this notation and the mathematical structures that underlie it are:

- The structure of a signature coalgebra provides a framework in which control and data transformation are separately specified. The major states of a system structure the design of a program.

- All familiar iteration schemes can be modeled by varieties of coalgebras.

- Each variety of coalgebra has associated with it proof rules that virtually dictate the form of proofs of safety properties of algorithms constructed with coalgebras of the variety.

- Liveness properties are verified through a hybrid deduction scheme in which temporal logical inference is used in conjunction with symbolic model checking.

References

[BG88] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. Technical Report 842, INRIA, May 1988.

[Bir86] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO Series F*. Springer-Verlag, 1986.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CGL92] E. M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 124–175. Springer Verlag, 1992.

[EC82] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronizations skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[Kie94] Richard B. Kieburtz. Programming with algebras. Technical report, Oregon Graduate Institute of Science & Technology, July 1994.

[KL94] Richard B. Kieburtz and Jeffrey Lewis. Algebraic Design Language—Preliminary definition. Technical report, Pacific Software Research Center, Oregon Graduate Institute of Science & Technology, January 1994.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, August 1991.

[Pau93] Lawrence C. Paulson. Co-induction and co-recursion in higher-order logic. Technical Report TR 304, Computing Laboratory, Cambridge University, December 1993.