Towards Compositional Feedback in Non-Deterministic and Non-Input-Receptive Systems *

Viorel Preoteasa[1] and Stavros Tripakis[1,2]

[1]Aalto University and [2]University of California, Berkeley

Abstract

Feedback is an essential composition operator in many classes of reactive and other systems. This paper studies feedback in the context of compositional theories with refinement. Such theories allow to reason about systems on a component-by-component basis, and to characterize substitutability as a refinement relation. Although compositional theories of feedback do exist, they are limited either to deterministic systems (functions) or input-receptive systems (total relations). In this work we propose a compositional theory of feedback which applies to non-deterministic and non-input-receptive systems (e.g., partial relations). To achieve this, we use the semantic frameworks of predicate and property transformers, and relations with fail and unknown values. We show how to define instantaneous feedback for stateless systems and feedback with unit delay for stateful systems. Both operations preserve the refinement relation, and both can be applied to non-deterministic and non-input-receptive systems.

1. Introduction

Large and complex systems are best designed using component-based rather than monolithic methodologies. In the component-based paradigm a system is an assembly of interacting components. Each component can itself be a subsystem composed from other components. Arbitrary component hierarchies can be built in this manner, allowing to decompose a complex system into simpler and further simpler subsystems.

Systems constantly evolve, and a key concern in a component-based setting is *substitutability*: when can we replace some component A with a new component A'? The issue is *property preservation*: the replacement must not result in breaking essential properties of the existing system. Moreover, we would like to avoid as much as possible having to re-verify the new system from scratch, since verification is an expensive process. Ideally, a simple check should suffice to guarantee that the new system is correct, assuming that the old system was correct.

* This work was partially supported by the Academy of Finland, the U.S. National Science Foundation (awards #1329759 and #1139138), and by UC Berkeley's iCyPhy Research Center (supported by IBM and United Technologies).

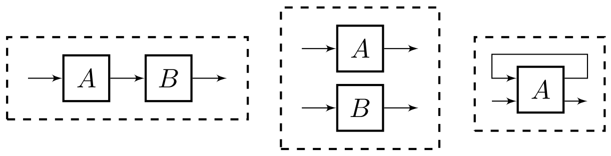

Figure 1: Composition operators: serial composition (left); parallel composition (middle); feedback composition (right).

Compositional theories with refinement provide an elegant framework for reasoning about these problems. In that framework the following notions are central: (a) the notion of component; (b) the notion of composition operators, which allow to obtain new components by composing existing ones; (c) the notion of *refinement*, which is a binary relation between components. Compositionality here comes in many forms, but typically two main theorems are provided by such a theory:

1. *Preservation of refinement by composition*, which can be roughly stated as: if A' refines A and B' refines B, then the composition of A' and B' refines the composition of A and B.
2. *Preservation of properties by refinement*, which can be roughly stated as: if A' refines A and A satisfies property ϕ, then A' also satisfies ϕ.

To see how the above two theorems can be used, consider again the scenario above where A is replaced by A'. We can avoid re-verifying the new system by trying instead to show that A' refines A. If this is true, then by preservation of refinement by composition it follows that the new system (with A') refines the old system (with A). (Here we also implicitly use that refinement is reflexive, which is typically true.) Moreover, by preservation of properties by refinement it follows that if property ϕ holds in the old system then it will also hold in the new system. In this way, we can avoid checking ϕ directly on the new system, which can be expensive when the system is large. Instead, we must show that A' refines A, which is an easier verification problem, as it deals with only two components.

In this paper we look at components as blocks with inputs and outputs. In such a setting, there are three basic composition operators (see Figure 1):

- *Serial composition*, where the output of one component is connected to the input of another component.
- *Parallel composition*, where two components are "stacked on top of each other" without any connections, but encapsulated into a single component.
- *Feedback composition*, where the output of a component is connected into one of its inputs.

Semantically, we want our components to be able to model systems which are both *non-deterministic* and *non-input-receptive*. Non-determinism means that the system may produce more than one outputs for a given input. Non-determinism is well established as a powerful modeling mechanism which allows abstracting from unnecessary details among other things. Non-input-receptiveness means that the system may declare some input values to be *illegal* (always, or at certain times). For example, a component computing the square root of a real number may declare that negative inputs are illegal. Note that this is different from stating that if the input is negative then the output can be arbitrary. The latter corresponds logically to an implication: $x \geq 0 \Rightarrow y = \sqrt{x}$ (here x denotes the input and y the output); whereas the former corresponds to a conjunction: $x \geq 0 \wedge y = \sqrt{x}$. Although less common in the domains of formal methods and verification, non-input-receptive systems are common in programming languages and type theory, where they provide the basis for type checking. Type checking is a key feature of many languages and can be seen as a "lightweight" verification mechanism as the programmer does not have to provide a formal specification of her program (the program only needs to type check). In our setting, non-input-receptiveness is essential in order to be able to model *compatibility* of components during composition. For an extensive discussion of this point, see [7, 27].

In this setting, parallel composition is relatively straightforward to define and can be logically seen as taking the conjunction of the specifications of the two components. Serial composition is a bit trickier and logically involves $\forall$-$\exists$ quantification, which essentially states that any output of the upstream component is a legal input for the downstream component (i.e., there exists for it an output of the downstream component). This *game theoretic* semantics is necessary when the components are both non-deterministic and non-input-receptive, while it collapses to the standard semantics (composition of functions or relations, or logical conjunction) when components are either deterministic or input-receptive [7, 8, 27]. Both parallel and serial composition as defined above are compositional in the sense that they preserve refinement.

Compositional theories of feedback do exist, but they are limited to either deterministic systems, or to input-receptive systems. To our knowledge, compositional feedback for both non-deterministic and non-input-receptive systems has so far remained elusive. The main difficulty has been to come up with a definition of feedback which preserves refinement. That is, if A' refines A, then $\kappa(A')$ refines $\kappa(A)$, where $\kappa(\cdot)$ is the feedback composition operator. An extensive account of some of the difficulties and failed attempts is given in [26], for the framework of *relational interfaces* [27].

To illustrate some of the earlier problems and how they are handled in the new framework, consider the following example, first given in [9]. Let A be a component with two input variables x, u, one output variable y, and input-output relation true, which means that A accepts any input value, and given an input value, it can return any output (because true is satisfied by any (x, u, y) triple). Now let A' be another component with inputs x, u, output y, and relation $x \neq y$. This means that A' accepts any input (x, u), and given such x, returns some y different from x. In the relational interface framework (as well as in our framework), A' refines A. This is because A' has the same legal inputs as A (in fact, every input is legal for both) and A' is "more deterministic" than A.

Now, suppose we apply feedback to both A and A', in particular, we connect output y to input x. Denote the resulting components by B and B', respectively. Both B and B' have a single input, u (x is no longer a free input, since it has been connected to y). What should the relations of B and B' be? Note that we want B' to refine B, otherwise refinement is not preserved by feedback. The straightforward way to define feedback is to add the constraint $x = y$, since y is connected to x, essentially becoming the same "wire". Doing so, we obtain $\mathsf{true} \wedge x = y$, i.e., $x = y$, as the relation of B, and $x \neq y \wedge x = y$, i.e., false, as the relation of B'. This is problematic, since false does not refine $x = y$. Indeed, according to the game-theoretic semantics, refinement is *not* implication, because inputs and outputs must be treated differently. In our example, any input u is legal for B, whereas no input u is legal for B', which means that B' cannot replace B, i.e., does not refine it.

In this paper, we attack the problem of feedback afresh. We adopt the frameworks of *refinement calculus* (RC) [3] and *refinement calculus of reactive systems* (RCRS) [23]. RC is a classic compositional framework for reasoning about sequential programs, while RCRS is an extension of RC to reason about reactive systems. In RC, programs are modeled as (monotonic) *predicate transformers*, i.e., functions from sets of post-states to sets of pre-states. In RCRS, systems are modeled as (monotonic) *property transformers*, which are functions that transform sets of output infinite sequences into sets of input infinite sequences. Both predicate and property transformers are powerful mathematical tools, able to capture both non-deterministic and non-input-receptive systems. In fact, they are more powerful than partial relations, which is the semantic basis of relational interfaces [27]. Therefore one can hope that some of the difficulties in obtaining a compositional definition of feedback in the relational framework can be overcome in the refinement calculi frameworks.

We first consider *stateless* systems and *instantaneous feedback*, i.e., feedback in a single step (Section 5). We model stateless systems using predicate transformers, where instead of a post/pre-state interpretation, we use an output/input interpretation. We often employ *strictly conjunctive* predicate transformers, which correspond to input-output relations over a domain extended with a special value *fail* (denoted $\bullet$), used to model illegal inputs (those return $\bullet$ as output). We further extend these relations with an additional *unknown* value (denoted $\perp$). The unknown value is used to define feedback by means of a fix-point iteration, starting with unknown and potentially converging to known values. The idea to use unknown values and fix-points comes from methods for dealing with instantaneous feedback in deterministic and input-receptive systems (i.e., total functions) [4, 10, 17]. In our case systems are non-deterministic and also may reject some inputs. We therefore need to generalize the fix-point method to relations with fail and unknown. The precise definition is involved (Def. 2), but roughly speaking, for a given input we explore all possible computation paths, including those in which the feedback input stabilizes to a known or unknown value, as well as those in which either the relation returns $\bullet$ or the path does not stabilize. We can then consider an input to the post-feedback system as legal if no execution path fails, and those which stabilize reach a known value.

This instantaneous feedback operator preserves refinement. In the example discussed above, the new definition results in both B and B' being the component Fail which rejects all inputs, since the above procedure does not stabilize in neither the case of A nor A'. This indicates that the two feedback operations are illegal, which is similar to discovering an incompatibility during, say, serial composition.

Relations with fail and unknown have the same modeling power as strictly conjunctive predicate transformers over input/output values with unknowns. We use the two models alternatively, as each is best suited to obtain certain results. In particular, we define the instantaneous feedback operator using relations with fail and unknowns, but we prove that the operator preserves refinement using their predicate transformer representation.

We next turn to the case of *stateful* systems. Syntactically, we model these as *symbolic transition systems*, i.e., relations on input, output, current state, and next state variables. For instance, a simple counter can be modeled by the relation $u' = u + 1$, where u denotes the current state and u' the next state. Semantically, we use predicate transformers on the same variables [23]. We then define *feedback with unit-delay* for such predicate transformers, which corresponds to connecting u' to u via a *unit-delay* component. A unit-delay outputs at step k its input at the previous step $k-1$. Thus, feedback with unit-delay ensures that the next state u' becomes the current state in the next step. The feedback with unit-delay operator is defined in Section 7. That definition is aided by iteration operators over property transformers defined in Section 6.

Combining the two feedback operators generalizes block-diagram languages like Simulink, which are not able to handle non-deterministic and non-input-receptive components.

Our treatment in this paper is mostly semantic. However, following [23], we can use syntactic objects such as symbolic transition systems to represent predicate and property transformers. For example, the formula $u' = u + 1$ represents a counter, as mentioned above; the formula $x \geq 0 \wedge y = \sqrt{x}$, where x is input and y is output, represents a relation with fail and also a predicate transformer. We can also use *temporal logic* (e.g., LTL [21]) to express properties over infinite sequences of input-output values, including liveness properties. For example, a system specified by the LTL formula $\Box\,(x \Rightarrow \Diamond\, y)$ ensures that if the input x is infinitely often true, so will be the output y. Using these representations, most of the constructs presented in this paper can actually be computed as first-order formulas, or as quantified LTL formulas (Section 8).

All the results presented in this paper have been formalized and proved in the Isabelle theorem prover [19]. The Isabelle code is available from `http://users.ics.aalto.fi/viorel/FeedbackIsabelle.zip`. The code has been tested with versions 2015 and 2016-RC1 of Isabelle. An extended version of this paper is also available [24].

2. Related Work

Our idea to use relations with an unknown value is inspired from fix-point based methods such as *constructive semantics*, which are used to reason about instantaneous feedback in cyclic combinational circuits and synchronous languages [4, 10, 17]. These methods deal with monotonic functions over the flat CPO with unknown as the least element. Because they assume total functions, they are limited to deterministic and input-receptive systems. Refinement is also absent from these frameworks.

Fix-point based methods have also been used to reason about non-instantaneous feedback, in systems such as Kahn Process Networks [15], where processes are modeled as continuous functions on streams. Again, these are total functions, and therefore deterministic and input-receptive. Non-deterministic extensions of process networks exist [14] but they do not include a notion of refinement.

Frameworks which allow non-determinism and also include refinement are I/O automata [16], reactive modules [1], and Focus [5]. All three frameworks assume that components are input-receptive. Note that in the example given in the introduction, although components A and A' are input-receptive, and so is B, B' is not. This shows that the straightforward definition of feedback does not preserve input-receptiveness. It is unclear how this example could be handled in the above frameworks.

Interface automata [8] allow non-deterministic as well as non-input-receptive specifications. Composition in this framework is asynchronous parallel composition with input-output label synchronization, and therefore difficult to directly compare with feedback composition in our framework. Also, interface automata are limited to safety whereas our framework can also express liveness properties.

Checking compatibility within the framework of functional programing languages, can be done using type-checking as in the Standard ML language [18], or using more advanced techniques as Refinement Types for ML [11], Dependent Types [28], and Liquid Types [25]. These are techniques based on subtypes and dependent-types that allow checking at compile time of invariants about the computed values of the program. These techniques are in general automated, therefore they are applicable only to certain classes of decidable problems. Our compatibility check for system compositions is more general, and not necessarily decidable. In our case checking compatibility of system compositions is reduced to checking satisfiability of formulas. If the underlying logic of these formulas is decidable, then we can have an automatic compatibility test.

3. Background

We use higher order logic as implemented by the Isabelle theorem prover. Capital letters $X, Y, \ldots$ denote types and small letters denote elements of these types $x \in X$. Bool denotes the type of Boolean values true and false, Nat the natural numbers, and Real the reals. We use $\wedge$, $\vee$, $\Rightarrow$, and $\neg$ for the Boolean operations.

If X and Y are types, then $X \rightarrow Y$ denotes the type of function from X to Y. For function application we use $f.x$ instead of $f(x)$. The function type constructor associates to right, and correspondingly the function application associates to left. That is $X \rightarrow Y \rightarrow Z$ is the same as $X \rightarrow (Y \rightarrow Z)$ and $f.x.y$ is the same as $(f.x).y$. We also use lambda abstraction to construct functions. For example if $x + 2 \cdot y \in \mathsf{Nat}$ is natural expression, then $(\lambda x, y : x + 2 \cdot y) : \mathsf{Nat} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat}$ is the function that maps x and y into $x + 2 \cdot y$. $X \times Y$ denotes the Cartesian product of X and Y.

Predicates are functions with one or more arguments returning Booleans (e.g., a predicate with two arguments has signature $X \rightarrow Y \rightarrow \mathsf{Bool}$), and relations are predicates with at least two arguments. For a relation $r : X \rightarrow Y \rightarrow \mathsf{Bool}$ we denote by $\mathsf{in}.r : X \rightarrow \mathsf{Bool}$ the predicate given by $\mathsf{in}.r.x = (\exists y : r.x.y)$. If r is a relation with more than two arguments, then $\mathsf{in}.r$ is defined similarly by quantifying over the last argument: $\mathsf{in}.r.x.y.z = (\exists u : r.x.y.z.u)$.

We extend point-wise all operations on Booleans to operations on predicates and relations. For example if $r, r' : X \rightarrow Y \rightarrow \mathsf{Bool}$ are two relations then $r \wedge r'$ and $r \vee r'$ are the relations given by $(r \wedge r').x.y = r.x.y \wedge r'.x.y$ and $(r \vee r').x.y = r.x.y \vee r.x.y$, and $(r \leq r') = (\forall x, y : r.x.y \Rightarrow r'.x.y)$. We use $\bot$ and $\top$ as the smallest and greatest predicates (relations): $\bot.x = \mathsf{false}$ and $\top.x = \mathsf{true}$.

For relations $r : X \rightarrow Y \rightarrow \mathsf{Bool}$ and $r' : Y \rightarrow Z \rightarrow \mathsf{Bool}$, we denote by $r \circ r'$ the *relational composition* given by $(r \circ r').x.z = (\exists y : r.x.y \wedge r'.y.z)$. For a relation $r : X \rightarrow Y \rightarrow \mathsf{Bool}$ we denote by $r^{-1} : Y \rightarrow X \rightarrow \mathsf{Bool}$ the *inverse* of r ($r^{-1}.y.x \Leftrightarrow r.x.y$). A relation r is *total* (sometimes called *left-total*) if for all x there is y such that $r.x.y$.

For reactive systems we need *infinite sequences* (or *traces*) of values. Formally, if X is a type of values, then a trace from X is an element $\mathbf{x} \in X^\omega = (\mathsf{Nat} \rightarrow X)$. To make the reading easier, we denote infinite sequences by bold face letters $\mathbf{x}, \mathbf{y}, \ldots$ and values by normal italic small letters $x, y, \ldots$. When, for example, both x and $\mathbf{x}$ occur in some formula, we use the convention that the elements of $\mathbf{x}$ have the same type as x. For $\mathbf{x} \in X$, $\mathbf{x}_i = \mathbf{x}.i$, and $\mathbf{x}^i \in X^\omega$ is given by $\mathbf{x}^i_j = \mathbf{x}^i.j = \mathbf{x}_{i+j}$. We consider pair of traces $(\mathbf{x}, \mathbf{y})$ as being the same as traces of pairs $(\lambda i : (\mathbf{x}_i, \mathbf{y}_i))$.

3.1 Linear temporal logic

Linear temporal logic (LTL) [21] is a logic used for specifying properties of reactive systems. In this paper we use a semantic (algebraic) version of LTL as in [23]. This logic is formalized in Isabelle, and the details of this formalization are available in [22]. An LTL formula is a predicate on traces and the temporal operators are functions mapping predicates to predicates. We call predicates over traces (i.e., sets of traces) *properties*.

If $p,\ q \in X^\omega \to \mathsf{Bool}$ are properties, then *always* p, *next* p, and p *until* q are also properties and they are denoted by $\Box\, p$, $\bigcirc\, p$, and $p\ \mathsf{U}\ q$ respectively. The property $\Box\, p$ is true in $\mathbf{x}$ if p is true at all time points in $\mathbf{x}$, $\bigcirc\, p$ is true in $\mathbf{x}$ if p is true at the next time point in $\mathbf{x}$, and $p\ \mathsf{U}\ q$ is true in $\mathbf{x}$ if there is some time in $\mathbf{x}$ when q is true, and until then p is true. Formally we have:

$$(\Box\, p).\mathbf{x} = (\forall n : p.\mathbf{x}^n),\ \ (\bigcirc\, p).\mathbf{x} = p.\mathbf{x}^1$$
$$(p\ \mathsf{U}\ q).\mathbf{x} = (\exists n : (\forall i < n : p.\mathbf{x}^i) \wedge q.\mathbf{x}^n)$$

In addition we define the operator $p\,\mathsf{L}\,q = \neg(p\ \mathsf{U}\ \neg q)$. Intuitively, $p\,\mathsf{L}\,q$ holds if, whenever p holds continuously up to step $n-1$, then q must hold at step n.

If r is a relation on traces we define the temporal operators similarly. For example $(\Box\, r).\mathbf{x}.\mathbf{y}.\mathbf{z} = (\forall n : r.\mathbf{x}^n.\mathbf{y}^n.\mathbf{z}^n)$. If r is a relation on values, then we extend it to traces by $r.\mathbf{x}.\mathbf{y}.\mathbf{z} = r.\mathbf{x}_0.\mathbf{y}_0.\mathbf{z}_0$.

3.2 Refinement calculus of reactive systems

In refinement calculus programs are modeled as *monotonic predicate transformers* [3]. A program S with input type X and output type Y is modeled as a function from $(Y \to \mathsf{Bool}) \to (X \to \mathsf{Bool})$ with a weakest precondition interpretation. If $q : Y \to \mathsf{Bool}$ is a post-condition (set of final states), then $S.q$ is the set of all initial states from which the program always terminates and it terminates in a state from q. We model reactive systems as *monotonic property transformers*, also with a weakest precondition interpretation [23]. A reactive system S with input type X and output type Y is formally modeled as a monotonic function from $(Y^\omega \to \mathsf{Bool}) \to (X^\omega \to \mathsf{Bool})$. If $q : Y^\omega \to \mathsf{Bool}$ is a post-property (a set of output traces), then $S.q$ is the set of input traces for which the execution of S does not fail and it produces an output trace in q. The remaining of this section applies equally to both monotonic predicate transformers and monotonic property transformers, and we will call them simply monotonic transformers.

The point-wise extension of the Boolean operations to predicates, and then to monotonic transformers gives us a *complete lattice* with the operations $\sqsubseteq, \sqcap, \sqcup, \bot, \top$ (observe that $\sqcap$ and $\sqcup$ preserve monotonicity). All these lattice operations are also meaningful as operations on programs and reactive systems. The order of this lattice ($S \sqsubseteq T \Leftrightarrow (\forall q : S.q \le T.q)$) is the *refinement relation*. If $S \sqsubseteq T$, then we say that T *refines* S (or S is *refined* by T). If T refines S then T can replace S in any context.

The operations $\sqcap$ and $\sqcup$ model *demonic* and *angelic non-determinism* or *choice*. The interpretation of $\sqcap$ is that the system $S \sqcap T$ is correct (i.e., satisfies its specification) if both S and T are correct. On the other hand $S \sqcup T$ is correct if one of S and T are correct. Thus, $\sqcap$ and $\sqcup$ represent uncontrollable and controllable non-determinism, respectively.

We denote the bottom and the top of the lattice of monotonic transformers by Fail and Magic respectively. Fail is refined by any monotonic transformer and Magic refines every monotonic transformer. Fail does not guarantee any property. For any q, $\mathsf{Fail}.q = \bot$, i.e., there is no input (sequence) for which Fail will produce an output (sequence) from q. On the other hand Magic can establish any property q ($\mathsf{Magic}.q = \top$). The problem with Magic is that it cannot be implemented.

Serial composition of two systems modeled as transformers S and T is simply the functional composition of S and T ($S \circ T$). For monotonic transformers we denote this composition by $S\ ;\ T$ ($(S\ ;\ T).q = S.(T.q)$). To be able to compose S and T, the type of the output of S must be the same as the type of the input of T.

The system Skip defined by $(\forall q : \mathsf{Skip}.q = q)$ is the neutral element for serial composition:

$$\mathsf{Skip}\ ;\ S = S\ ;\ \mathsf{Skip} = S.$$

We now define two special types of monotonic transformers which will be used to form more general property transformers by composition. For predicate $p : X \to \mathsf{Bool}$ and relation $r : X \to Y \to \mathsf{Bool}$ we define the *assert transformer* $\{p\} : (X \to \mathsf{Bool}) \to (X \to \mathsf{Bool})$, and the *demonic update transformer* $[r] : (Y \to \mathsf{Bool}) \to (X \to \mathsf{Bool})$ as follows:

$$\{p\}.q = (p \wedge q), \qquad [r].q.x = (\forall y : r.x.y \Rightarrow q.y)$$

The assert system $\{p\}$ when executed from a sequence x, behaves as skip when $p.x$ is true, and it fails otherwise. The demonic update statement $[r]$ establishes a post condition q when starting in x if all y with $r.x.y$ are in q. If $r = \bot$, then $[r] = [\bot] = \mathsf{Magic}$.

If R is an expression in x and y, then $[x \leadsto y \mid R] = [\lambda x, y : R]$. For example if R is $z = x + y$, then $[x,\ y \leadsto z \mid z = x + y] = [\lambda(x,\ y),\ z : z = x + y]$ is the system which produces in the output z the value $x + y$ where x and y are the inputs. If e is an expression in x, then $[x \leadsto e] = [x \leadsto y \mid y = e]$, where y is a new variable different of x and which does not occur free in e. For assert statements we introduce similar notation. If P is an expression in x then $\{x \mid P\} = \{\lambda x : P\}$. If P is $x \le y$, then $\{x,\ y \mid x \le y\} = \{\lambda(x, y) : x \le y\}$. The variables x and y are bounded in $[x \leadsto y \mid R]$ and $\{x \mid P\}$.

The *havoc monotonic transformer* is the most non-deterministic statement $\mathsf{Havoc} = [x \leadsto y \mid \mathsf{true}]$ and it assigns some arbitrary value to the output y.

In RCRS we can model (global) specifications of reactive systems, e.g., using LTL as a specification language, as well as concrete implementations, e.g., using symbolic transition systems (relations on input, output, state, and next-state variables) [23]. A global specification may define what the output of a system is when the input is an entire sequence $\mathbf{x}$. A global specification may also impose conditions on the entire input sequence $\mathbf{x}$, including liveness conditions. For example, the formula $\Box \Diamond x$ asserts that the input *must* be true infinitely often. An implementation describes how the system works step-by-step. For instance, an implementation may start with input $\mathbf{x}_0$ and some initial internal state $\mathbf{u}_0$, compute the output $\mathbf{y}_0$ and a new internal state $\mathbf{u}_1$, and then use $\mathbf{x}_1$ and $\mathbf{u}_1$ to compute $\mathbf{y}_1$ and $\mathbf{u}_2$, and so on.

Example 1 (System modeling with property and predicate transformers)**.** We can model a global reactive system Sum that for every input sequence $\mathbf{x} \in \mathsf{Nat}^\omega$ computes the sequence $sum.\mathbf{x} \in \mathsf{Nat}^\omega$, where $sum.\mathbf{x}.n = \mathbf{x}_0 + \mathbf{x}_1 + \ldots + \mathbf{x}_{n-1}$. We model Sum as a monotonic property transformer:

$$Sum = [\mathbf{x} \leadsto sum.\mathbf{x}]$$

We can also model a step of an implementation of Sum as a predicate transformer:

$$StepSum = [u, x \leadsto u', y \mid u' = u + x \wedge y = u] \qquad (1)$$

The relation $u' = u + x \wedge y = u$ describes a symbolic transition system: u is the current state, u' the next state, x is the input, and y the output. $StepSum$ works iteratively starting at initial state $\mathbf{u}_0$, and the first element of the input $\mathbf{x}$, and assuming that $\mathbf{u}_0 = 0$, it computes $\mathbf{u}_1 = \mathbf{u}_0 + \mathbf{x}_0 = \mathbf{x}_0$ and $\mathbf{y}_0 = \mathbf{u}_0 = 0 = sum.\mathbf{x}.0$, then using $\mathbf{u}_1$ and second element $\mathbf{x}_1$ of $\mathbf{x}$, it computes $\mathbf{u}_2 = \mathbf{u}_1 + \mathbf{x}_1 = \mathbf{x}_0 + \mathbf{x}_1$ and $\mathbf{y}_1 = \mathbf{u}_1 = \mathbf{x}_0 = sum.\mathbf{x}.1$, and so on. We will show later in Example 11 how to obtain the Sum property transformer by connecting in $StepSum$ the output u' to the input u in feedback with a unit delay.

The assert and demonic choice predicate transformers satisfy the following properties [3]:

Theorem 1. *For p, p' and r, r' of appropriate types we have*

1. $\{p\}\ ;\ [r]\ ;\ \{p'\}\ ;\ [r'] = \{x \mid p.x \wedge \forall y : r.x.y \Rightarrow p'.y\}\ ;\ [r \circ r']$
2. $\{p\}\ ;\ [r] \sqsubseteq \{p'\}\ ;\ [r'] \Leftrightarrow (\forall x : (p.x \Rightarrow p'.x) \wedge (\forall y : p.x \wedge r'.x.y \Rightarrow r.x.y))$

3.3 Product and fusion of monotonic transformers

For two predicates p and q their *product* is denoted $p \times q$ and is given by

$$(p \times q).(s, s') = p.s \wedge q.s'$$

The *product* (parallel composition) of two monotonic transformers S and T is denoted by $S \times T$ and is given by

$$(S \times T).q.(s, s') = (\exists p, r : p \times r \leq q \wedge S.p.s \wedge T.r.s')$$

If $\mathcal{S} = \{S_i\}_{i \in I}$, is a collection of monotonic transformers, then the *fusion* of $\mathcal{S}$, denoted $\mathsf{Fusion}.\mathcal{S}$, is a monotonic transformer given by

$$\mathsf{Fusion}.\mathcal{S}.q.s = \big(\exists p : (\bigwedge_{i \in I} p_i) \leq q \wedge (\forall i : S_i.p_i.s)\big)$$

When $\mathcal{S}$ contains just two transformers S and T, we write $\mathsf{Fusion}(S, T)$ instead of $\mathsf{Fusion}(\{S, T\})$.

Note that product and fusion are similar, but not the same. The result of the product $A \times B$ is a component with separate inputs and outputs for A and B (see Figure 1, parallel composition), whereas the result of $\mathsf{Fusion}(A, B)$ is a component with “shared” input and output.

Theorem 2. *Fusion and product satisfy the following [2]*

1. $\mathsf{Fusion}.(\{\{p_i\}\ ;\ [r_i]\}_{i \in I}) = \{\bigwedge_i p_{i \in I}\}\ ;\ [\bigwedge_{i \in I} r_i]$
2. $S \sqsubseteq S'$ and $T \sqsubseteq T$' implies $S \times T \sqsubseteq S' \times T'$
3. $(\forall i : S_i \sqsubseteq T_i)$ implies $\mathsf{Fusion}.\mathcal{S} \sqsubseteq \mathsf{Fusion}.\mathcal{T}$
4. $(\{p\}\ ;\ [r]) \times (\{p'\}\ ;\ [r']) = \{x, y \mid p.x \wedge p'.y\}\ ;\ [x, y \leadsto x', y' \mid r.x.x' \wedge r'.y.y']$

4. Relations with Fail and Unknown

In this section we formally introduce relations with fail and unknown values. As mentioned in the introduction, relations with fail and unknown and strictly conjunctive predicate transformers have the same expressive power, but each is better suited in defining, explaining, or proving certain results.

4.1 Relations with fail

First we introduce relations augmented with a special element $\bullet$ which we call “fail”. Relations with fail are used in the semantics of imperative programming languages to model non-termination in the context of non-determinism [20]. In this paper $\bullet$ is used to model non-input-receptive systems: when the output is $\bullet$, it means that the input is illegal.

We assume that $\bullet$ is a new element that does not belong to ordinary types. We define the type $A^\bullet = A \cup \{\bullet\}$. An element $x \in (A \times B)^\bullet$ is either $\bullet$ or it is a pair (a, b), with $a \in A$ and $b \in B$. A *relation with fail* from A to B is an element from $A^\bullet \to B^\bullet \to \mathsf{Bool}$. We assume that relations with fail map $\bullet$ only to $\bullet$, i.e., $R. \bullet .\bullet$ is true, and $R. \bullet .x$ is false for all $x \neq \bullet$. If R is a relation with fail, and $R.x.\bullet$, then x is *an illegal input* for R.

Example 2 (Division)**.** $\mathsf{Div} : (\mathsf{Real}^2)^\bullet \to \mathsf{Real}^\bullet \to \mathsf{Bool}$ is a relation with fail modeling division:

$$\mathsf{Div}.(x, y).\bullet = (y = 0) \text{ and } \mathsf{Div}.(x, y).z = (y \neq 0 \wedge z = x/y).$$

$\wedge_\perp$	$\perp$	false	true
$\perp$	$\perp$	false	$\perp$
false	false	false	false
true	$\perp$	false	true

Table 1: AND gate with unknown

Because $\mathsf{Div}.(x, 0).\bullet$ is true, we have that $(x, 0)$ is an illegal input for Div, for any x.

If x is an illegal input for R, then it is not important if R also maps x to some proper output y ($R.x.y = \mathsf{true}$, with $y \neq \bullet$). If R' is a relation almost identical to R except that $R'.x.y = \mathsf{false}$, then we consider R and R' as modeling the same system. We do not restrict relations with fail to relations that map illegal inputs only to $\bullet$ because we want to model the *demonic choice* of relations by union. If R and R' are of the same type, then $R \vee R'$ models the demonic choice of R and R'. If x is illegal for R, then x is illegal also for $R \vee R'$, even if x is legal for R'.

The relation fail is defined by $(\forall x, y : \mathsf{fail}.x.y = (y = \bullet))$. All inputs are illegal for fail. Similarly to predicate transformers we use the syntax $(x \leadsto e)$ for the relation R that satisfies $R. \bullet .\bullet$ and $(\forall x \neq \bullet : R.x.e)$, where e is an expression containing x.

Relations with fail are closed under relation composition (serial composition), and union (demonic choice).

Definition 1. For $R, R' : A^\bullet \to B^\bullet \to \mathsf{Bool}$ two relations with fail of the same type, we define the *refinement* $R \sqsubseteq R'$ by $(\forall x : R.x. \bullet \vee (R'.x \leq R.x))$.

4.2 Unknown values

We denote by $\perp$ a special element called *unknown* which is not part of ordinary types: Nat, Bool, Real, ... For a set A we define a partial order on $A_\perp = A \cup \{\perp\}$ so that, for $a, b \in A_\perp$:

$$a \leq b \;\Leftrightarrow\; a = \perp \vee a = b$$

This order is the *flat complete partial order* (flat CPO) [6].

We extend two partial orders on A and B to a partial order on the Cartesian product $A \times B$ by

$$(a, b) \leq (a', b') \Leftrightarrow a \leq a' \wedge b \leq b'$$

$\perp$ is the smallest element of $A_\perp$, and for the Cartesian product $A = A_{1\perp} \times \ldots \times A_{n\perp}$we denote also by $\perp$ the tuple $(\perp, \ldots, \perp) \in A$.

For partial order $(A, \leq)$, an element $m \in A$ is *maximal* if there is no element in A greater than m:

$$\mathsf{maximal}.m = (\forall x \in A : m \not\lneq x)$$

Lemma 1. *An element of $(a, b) \in A_\perp \times B_\perp$ is maximal if and only if $a \neq \perp$ and $b \neq \perp$.*

The context will distinguish the cases when $\perp$ is an element of $A_\perp$, or the smallest predicate or relation. However, in all cases $\perp$ denotes the smallest element of some partially ordered set.

Example 3 (AND gate with unknown)**.** Consider the function $\wedge_\perp : \mathsf{Bool}_\perp \to \mathsf{Bool}_\perp \to \mathsf{Bool}_\perp$ defined in Table 1. It models a logical AND gate which returns false if at least one of its inputs is false, even when the other input is unknown. We will return to this example in Example 6.

4.3 Relations with fail and unknown

We use unknown values ($\perp$) as the starting value of the feedback variable in an iterative computation of the feedback, and we use $\bullet$ to represent “failure” (e.g. illegal inputs). To see why we distinguish the two, suppose we used the same value (say $\perp$) to represent both these concepts. Suppose $R.1.\perp$ is true for some relation R. This

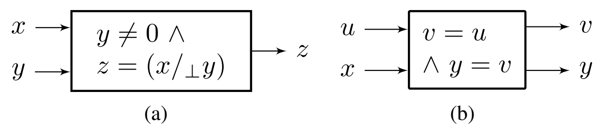

Figure 2: (a) Divide, (b) Identity

could mean that 1 is an illegal input for R, or that given 1 as input, the output is unknown. Now, suppose we have another relation R' that maps unknown inputs to 0, and we compose R and R' in series. What should the composition $R \circ R'$ produce given input 1? Is 1 illegal for $R \circ R'$, or does $R \circ R'$ map 1 to 0, or both? To avoid such ambiguities, we separate the two concepts and values.

Example 4 (Division with fail and unknown)**.** Continuing Example 2, we extend relation Div with unknown. If the input y is zero, then $\mathsf{Div}.(x, y)$ fails, otherwise if x is zero, then the result z is zero, even if y is unknown. The complete definition of Div is:

$$\begin{aligned}
&\mathsf{Div}.(x, y).\bullet = (y = 0)\\
&\mathsf{Div}.(x, y).z = (y \neq 0) \wedge ((x = 0 \wedge z = 0)\\
&\quad \vee (x \neq 0 \wedge y = \perp \wedge z = \perp) \vee (x = \perp \wedge z = \perp)\\
&\quad \vee (x \neq \perp \wedge y \neq \perp \wedge z = x/y))
\end{aligned}$$

Div is illustrated in Figure 2a, where we use $/_\perp$ for the operator $/$ extended to unknown values as defined above.

5. Instantaneous Feedback

In this section we introduce the definition of instantaneous feedback for a relation $R : (A \times B)^\bullet \to (A \times C)^\bullet \to \mathsf{Bool}$, as shown in Figure 3(a), where $A = A_{1\perp} \times \ldots \times A_{n\perp}$. The intention is to feed output v back into input u, as shown in Figure 3(b). The fact that we require R to have an extra input x and an extra output y is without loss of generality, as we can model absence of these wires by having their type (B or C) be the *unit type*, i.e., the type that has the empty tuple () as its only element.

The idea of the instantaneous feedback calculation is outlined next: We start with $\perp = (\perp, \ldots, \perp) \in A$, and some $x \in B$, and we construct the following tree: The nodes of the tree are labeled with elements in $(A \times (C \cup \{?\}))^\bullet$. The root of the tree is $(\perp, ?)$, and for every node of the tree of the form $(u, ?)$ we create new children nodes as follows:

1. If $R.(u, x).\bullet$ then we create the child $\bullet$.
2. If there exists v, y such that $u < v$ and $R.(u, x).(v, y)$, then we create the child $(v, ?)$.
3. If there exists y such that $R.(u, x).(u, y)$, then we create the child node (u, y).

We thus obtain a tree with leaves of the form $\bullet$, $(u, ?)$, or (u, y). Note that the depth of this tree is finite, because whenever we create a child $(v, ?)$ of $(u, ?)$ we require $u < v$, and we cannot have a strictly increasing infinite sequence of A elements (because A is a product of flat CPOs). That is, if all u values become known (non-$\perp$), then all children of $(u, ?)$ are leaves.

The *instantaneous feedback* of R, denoted $\mathsf{InstFeedback}.R$ is a relation from $B^\bullet$ to $(A \times C)^\bullet$ such that: $\mathsf{InstFeedback}.R.x.(u, y)$ is true if (u, y) is a leaf in the tree; and $\mathsf{InstFeedback}.R.x.\bullet$ is true if $\bullet$ is a leaf in the tree. The type of $\mathsf{InstFeedback}.R$ is shown in Figure 3(b).

Next we define formally $\mathsf{InstFeedback}.R$. In addition we define also another version of the feedback ($\mathsf{fb\_hide} : B^\bullet \to C^\bullet$) in which we remove the output component u if u is maximal (i.e., if it contains no unknowns), or we make the feedback fail otherwise. The type of fb_hide is shown in Figure 3(c).

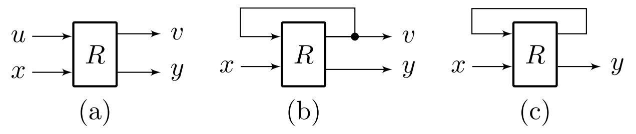

Figure 3: (a) relation R, (b) $\mathsf{InstFeedback}.R$, (c) $\mathsf{fb\_hide}.R$

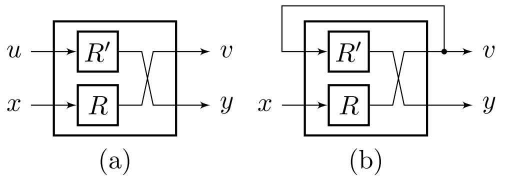

Figure 4: (a) cross product, (b) feedback of cross product

Definition 2. For a relation $R : (A \times B)^\bullet \to (A \times C)^\bullet \to \mathsf{Bool}$ we define $\mathsf{InstFeedback}.R : B^\bullet \to (A \times C)^\bullet \to \mathsf{Bool}$, $\mathsf{fb\_end} : (A \times C)^\bullet \to C^\bullet \to \mathsf{Bool}$, and $\mathsf{fb\_hide} : B^\bullet \to C^\bullet$:

$$\begin{aligned}
&\mathsf{InstFeedback}.R.x.(u, y)\\
&\quad = \exists n, u_0, \ldots, u_n : u_0 = \perp \wedge u_n = u \wedge R.(u, x).(u, y)\\
&\quad\quad \wedge \big(\forall i < n : u_i < u_{i+1} \wedge (\exists z : R.(u_i, x).(u_{i+1}, z))\big)\\
&\mathsf{InstFeedback}.R.x.\bullet\\
&\quad = \exists n, u_0, \ldots, u_n : u_0 = \perp \wedge R.(u_n, x).\bullet\\
&\quad\quad \wedge \big(\forall i < n : u_i < u_{i+1} \wedge (\exists z : R.(u_i, x).(u_{i+1}, z))\big)\\
&\mathsf{fb\_end}.(u, y).z = \mathsf{maximal}.u \wedge z = y\\
&\mathsf{fb\_end}.(u, y).\bullet = \neg\mathsf{maximal}.u\\
&\mathsf{fb\_hide}.R = \mathsf{InstFeedback}.R \circ \mathsf{fb\_end}
\end{aligned}$$

One of the main results of this paper is that the feedback operation preserves refinement:

Theorem 3. *If* $R, R' : (A_\perp \times B)^\bullet \to (A_\perp \times C)^\bullet \to \mathsf{Bool}$*, then*

$$R \sqsubseteq R' \Rightarrow \mathsf{InstFeedback}.R \sqsubseteq \mathsf{InstFeedback}.R'.$$

The next main result of this paper is that serial composition can be expressed as parallel composition followed by feedback. To state this, we first define the *cross product* of two relations, which is essentially their parallel composition, but with their outputs swapped, as shown in Figure 4a. The swapping is done so that we can apply feedback to the right "wires", as shown in Figure 4b.

Definition 3. The cross product of two relations with fail $R : A^\bullet \to B^\bullet \to \mathsf{Bool}$ and $R : C^\bullet \to D^\bullet \to \mathsf{Bool}$ is a relation with fail $R \otimes R' : (C \times A)^\bullet \to (B \times D)^\bullet \to \mathsf{Bool}$, given by

$$\begin{aligned}
R \otimes R'.(u, x).\bullet &= R.x.\bullet \vee R'.u.\bullet\\
R \otimes R'.(u, x).(v, y) &= R.x.v \wedge R'.u.y
\end{aligned}$$

We can now prove that the feedback of the cross product of two relations is the same as their serial composition:

Theorem 4. *If* $R : A_\perp^\bullet \to B_\perp^\bullet \to \mathsf{Bool}$*,* $R' : B_\perp^\bullet \to C_\perp^\bullet \to \mathsf{Bool}$*,* R' *is total and* $\neg R'.\perp.\bullet$ *then*

$$\begin{aligned}
&\mathsf{InstFeedback}.(R \otimes R').x.\bullet &&= (R \circ R').x.\bullet\\
&\mathsf{InstFeedback}.(R \otimes R').x.(u, y) &&= R.x.u \wedge R'.u.y\\
&\mathsf{fb\_hide}.(R \otimes R').x.\bullet &&= (R \circ R').x.\bullet \vee R.x.\perp\\
&\mathsf{fb\_hide}.(R \otimes R').x.y &&= (\exists u : u \neq \perp \wedge R.x.u \wedge R'.u.y)
\end{aligned}$$

The condition $\neg R'.\perp.\bullet$ is needed because if R' maps unknown ($\perp$) to fail ($\bullet$), then the feedback iteration stops and the whole left term maps some external input x to fail. On the other hand, if R maps x only to $a \notin \{\perp, \bullet\}$, and R' maps a only to $b \neq \bullet$, then, the

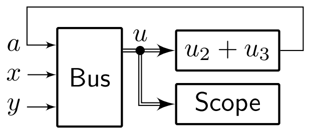

Figure 5: Bus followed by function block in feedback

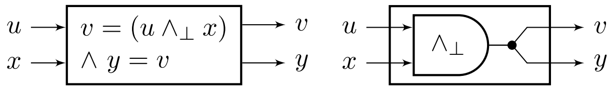

Figure 6: AND gate

serial composition does not fail for the same x. This assumption is reasonable, since all it states is that the unknown value must be legal, at least in contexts when we want to use a component in feedback.

We next illustrate our instantaneous feedback operator with a number of examples.

Example 5 (A seemingly algebraic loop in a Simulink diagram)**.** The diagram shown in Figure 5 is inspired from real-world Simulink models. This particular pattern is found in an automotive fuel control system benchmark by Toyota [12, 13]. In the picture, a, x, and y are inputs to a *Bus* block which outputs them as a tuple. This tuple is then fed into a function block which performs addition on the 2nd and 3rd elements of the tuple (u_2 and u_3). The output of the function is fed back into the bus. But there is no algebraic loop because the function does not depend on the 1st element of the tuple, i.e., a. The output of the bus is fed also into a *Scope* block which is supposed to print all three elements of the tuple on the screen. We can model the bus and the scope as the identity function, and we have:

$$\mathsf{Bus}.(a,x,y).(u_1,u_2,u_3) = (a = u_1 \wedge x = u_2 \wedge y = u_3)$$
$$\mathsf{Scope}.(u_1,u_2,u_3).(a,x,y) = (a = u_1 \wedge x = u_2 \wedge y = u_3)$$
$$\mathsf{Fun}.(u_1,u_2,u_3).v = (v = u_2 + u_3)$$
$$\mathsf{Sys} = \mathsf{Bus} \circ (u \leadsto u, u) \circ (\mathsf{Fun} \times \mathsf{Scope})$$

where Sys is the system without the feedback connection. If we apply the feedback we obtain, as expected:

$$\mathsf{InstFeedback.Sys} = (x, y \leadsto x + y, x, y)$$

Example 6 (Instantaneous feedback of AND gate)**.** Suppose we connect in feedback the AND gate function from Table 1. To apply feedback, we again add an extra output $y = v$ (see Figure 6), and we get:

$$\mathsf{AND}.(u,x).(v,y) = (v = y = u\ \wedge_\perp\ x)$$
$$\mathsf{InstFeedback.AND}.x.(v,y) =$$
$$(v = y) \wedge (x = \mathsf{false} \Rightarrow y = \mathsf{false}) \wedge (x \neq \mathsf{false} \Rightarrow y = \perp)$$
$$\mathsf{fb\_hide.AND}.x.y = (x = \mathsf{false} \Rightarrow y = \mathsf{false})$$
$$\wedge\ (x = \mathsf{true} \Rightarrow y = \bullet)$$

The result of the feedback is a component which outputs false when its free input x is false. On the other hand, if x is true or unknown, then the component fails.

The result above is consistent with the result obtained using classic theories of instantaneous feedback for total functions (or *constructive semantics*) [4, 10, 17]. This will generally be the case for deterministic systems, as the tree described in the instantaneous feedback calculation above, becomes a single path in those cases. Therefore, for the special case of total functions, our semantics specializes to the constructive semantics.

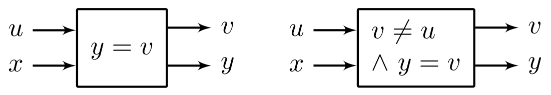

Figure 7: (a) Havoc, (b) Different

Example 7 (Instantaneous feedback of ID)**.** One might wonder what might happen if we connect the identity function in feedback. Since our feedback operator takes relations with two input and two output wires, we define the identity operator

$$\mathsf{ID}.(u,x).(v,y) = (u = v = y)$$

which ignores input x, and assigns both outputs v and y to its other input u (see Figure 2b). Output y allows to record the result of the feedback when we hide the feedback variables (u and v). As might be expected, the result of applying feedback is Fail:

$$\mathsf{InstFeedback.ID}.x.(v,y) = (v = y = \perp)$$
$$\mathsf{fb\_hide.ID} = \mathsf{fail}$$

The result is fail because the feedback variable is assigned the value unknown. Again, this is consistent with the result obtained using constructive semantics.

Example 8 (Instantaneous feedback of TRUE)**.** The power of our framework is that it applies not just to total functions as the examples above, but also to partial relations. Here, as an example, we consider the (total) relation true, discussed in the introduction. This system is also sometimes called *havoc*: it accepts any input, but can produce any output non-deterministically (Figure 7a – note that the equality $v = y$ is only on the outputs). Applying feedback, we get:

$$\mathsf{TRUE}.(u,x).(v,y) = (v = y)$$
$$\mathsf{InstFeedback.TRUE}.x.(v,y) = (v = y)$$
$$\mathsf{fb\_hide.TRUE} = \mathsf{fail}$$

As we should expect, the result of the new definition is fail. This is very different from the straightforward definition discussed in the introduction, and eliminates the problems presented there. The result is fail because again, as in the case of ID, the feedback variable can be assigned any value, including unknown.

Example 9 (Instantaneous feedback of NEQ)**.** Continuing the previous example and also the one of the introduction, consider the relation *different* (component A' in the introduction) which accepts any input and returns any output different from that input (see Figure 7b). Formally, we need also to define how the relation behaves with unknown values. Doing so, and applying feedback, we get:

$$\mathsf{NEQ}.(u,x).(v,y) = (v = y)$$
$$\wedge\ (u = \perp \Rightarrow v = \perp) \wedge (u \neq \perp \Rightarrow u \neq v \wedge v \neq \perp)$$
$$\mathsf{InstFeedback.NEQ}.x.(v,y) = (v = y = \perp)$$
$$\mathsf{fb\_hide.NEQ} = \mathsf{fail}$$

Again, as should be expected, the result of the feedback is fail. Therefore the refinement TRUE $\sqsubseteq$ NEQ is preserved by feedback, solving the problem identified in the introduction. Feedback of NEQ results in fail because we must choose output v to be different from input u, and when starting with unknown, we can only choose unknown.

Example 10 (Instantaneous feedback of a non-deterministic and non-input-receptive system)**.** Consider relation with fail and unknown ParNonDet parameterized by $a \in \mathsf{Nat}$, and having inputs

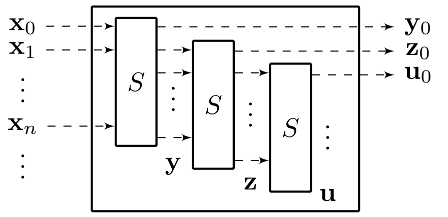

Figure 8: $\mathsf{IterateNextOmega}.S$

$u \in \mathsf{Nat}_\perp$ and $x \in \mathsf{Nat}$ and output $v \in \mathsf{Nat}_\perp$. ParNonDet is defined as:

$$\begin{aligned} \mathsf{ParNonDet}.a.(u, x).v = (u = a \Rightarrow v = \bullet) \\ \wedge (u \neq a \Rightarrow v = x + 1 \vee v = x + 2) \end{aligned}$$

that is, all inputs where $u = a$ are illegal, and if $u \neq a$ then the output v is non-deterministically $x+1$ or $x+2$. Note that if $u = \perp$ then $u \neq a$ (because a is a natural number) so in this case also, v is $x + 1$ or $x + 2$. Applying instantaneous feedback, we get:

$$\begin{aligned} &\mathsf{InstFeedback}.(\mathsf{ParNonDet}.a).x.v = \\ &\quad (x + 1 \neq a \wedge x + 2 \neq a \Rightarrow v = x + 1 \vee v = x + 2) \\ &\quad \wedge (x + 1 = a \Rightarrow v = \bullet \vee v = x + 2) \\ &\quad \wedge (x + 2 = a \Rightarrow v = \bullet \vee v = x + 1) \end{aligned}$$

that is, inputs where $x + 1 = a$ or $x + 2 = a$ become illegal, and for legal inputs, the output v is again non-deterministically chosen as $x + 1$ or $x + 2$.

6. Iteration Operators

In Section 7 we will define feedback with unit delay for arbitrary predicate transformers. In this section we introduce some auxiliary operators to aid that definition.

For a property transformer S with input and output of the same type we want to construct another property transformer $\mathsf{IterateNextOmega}.S$ which works in the following way. It starts by executing S on the input sequence $\mathbf{x}$ to obtain the output $\mathbf{y}$, then it executes S on the suffix $\mathbf{y}^1$ ($\mathbf{y}$ starting from position 1 instead of 0) to obtain $\mathbf{z}$, then it executes S on $\mathbf{z}^1$, and so on. The output of $\mathsf{IterateOmega}.S$ is the sequence $\mathbf{y}_0,\ \mathbf{z}_0,\ \cdots$. The operator IterateNextOmega is illustrated in Figure 8. In the figure, we use multiple incoming arrows to represent *sequences* of values, and not separate input "wires".

The first step in defining $\mathsf{IterateNextOmega}.S$ is to define a property transformer $\mathsf{SkipNext}.S$ which for some input $\mathbf{x}$, executes S on $\mathbf{x}^1$and leaves the first component $\mathbf{x}_0$ of $\mathbf{x}$ unchanged.

Definition 4. If S is a property transformer and $n \in \mathsf{Nat}$, then $\mathsf{SkipNext}.S$ and $\mathsf{SkipTop}.n$ are property transformers given by

$$\begin{aligned} \mathsf{SkipNext}.S &= [\mathbf{x} \leadsto \mathbf{x}_0, \mathbf{x}^1]\ ;\ (\mathsf{Skip} \times S)\ ;\ [z, \mathbf{w} \leadsto z \cdot \mathbf{w}] \\ \mathsf{SkipTop}.n &= [\mathbf{x} \leadsto \mathbf{y} \mid \mathsf{eqtop}.n.\mathbf{x}.\mathbf{y}] \end{aligned}$$

where $\mathsf{eqtop}.n.\mathbf{x}.\mathbf{y} = (\forall i < n : \mathbf{x}_i = \mathbf{y}_i)$ and $z \cdot \mathbf{w}$ denotes the new sequence obtained by prepending element z at the head of sequence $\mathbf{w}$.

Figure 9 gives a graphical illustration of these transformers. In the figure we use double arrows to represent sequences.

Intuitively, $\mathsf{SkipNext}.S$ takes the sequence $\mathbf{x}$, removes its "head" $\mathbf{x}_0$, applies S, and then propends to the output the head $\mathbf{x}_0$. $\mathsf{SkipTop}.n$ takes some sequence $\mathbf{x}$ and outputs a sequence $\mathbf{y}$ which has the same first n elements as $\mathbf{x}$, and is arbitrary after that.

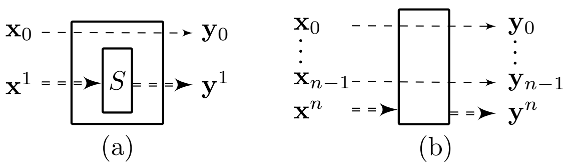

Figure 9: (a) $\mathsf{SkipNext}.S$, (b) $\mathsf{SkipTop}.n$

Then, $\mathsf{IterateNextOmega}.S$ would intuitively be the infinite serial composition

$$\mathsf{IterateNextOmega}.S = S\ ;\ \mathsf{SkipNext}.S\ ;\ \ldots\ ;\ \mathsf{SkipNext}^n.S\ ;\ \ldots$$

Formally, however, we cannot define an infinite serial composition. In order to solve this problem, we construct the finite compositions

$$S\ ;\ \mathsf{SkipNext}.S\ ;\ \ldots\ ;\ \mathsf{SkipNext}^n.S\ ;\ \mathsf{SkipTop}.n$$

for all n, and then we take the infinite fusion of all these property transformers. The details are omitted due to lack of space, and can be found in the extended version of this paper [24]. In [24] we also show how to calculate $\mathsf{IterateNextOmega}.S$ for the special case when the property transformer S is of the form $\{p\}\ ;\ [r]$. This result is used to prove Theorem 6 in the section that follows.

7. Feedback with Unit Delay

In Section 5 we defined an instantaneous feedback operator, for stateless systems. In this section we turn our attention to feedback for systems with state. We model a stateful system as a predicate transformer S with inputs u and x and outputs v and y, such that u models the *current* state, and v the *next* state (thus, u and v must have the same type). We will connect v to u in feedback via a unit delay. By doing so, the next state becomes the current state in the next step.

The result of the feedback-with-unit-delay operator applied to a predicate transformer as above is a property transformer that takes as input an infinite sequence $\mathbf{x}$ and produces an infinite output sequence $\mathbf{y}$, for a given initial state $\mathbf{u}_0$. The system starts by executing S on the first component $\mathbf{x}_0$ of $\mathbf{x}$, and on $\mathbf{u}_0$. This results in the values $\mathbf{u}_1$ and $\mathbf{y}_0$. In the next step, the second value $\mathbf{x}_1$of $\mathbf{x}$ and $\mathbf{u}_1$ are used as input for S resulting in $\mathbf{u}_2$ and $\mathbf{y}_1$, and so on. This computation is depicted in Figure 10a.

We achieve this computation by lifting S to a special property transformer and then applying the IterateNextOmega operator.

Definition 5. For a predicate transformer S as described earlier, we define the property transformer $\mathsf{AddDelay}.S$ with input a tuple of infinite sequences $\mathbf{u}, \mathbf{x}, \mathbf{y}$ and output of the same type:

$$\begin{aligned} &\mathsf{AddDelay}.S = [\mathbf{u}, \mathbf{x}, \mathbf{y} \leadsto (\mathbf{u}_0, \mathbf{x}_0), (\mathbf{u}_0, \mathbf{x})]\ ;\ (S \times \mathsf{Skip})\ ; \\ &\quad [(a, b), (u, \mathbf{x}) \leadsto \mathbf{u}', \mathbf{x}', \mathbf{y}' | \mathbf{u}'_0 = u \wedge \mathbf{u}'_1 = a \wedge \mathbf{y}'_0 = b \wedge \mathbf{x}' = \mathbf{x}] \end{aligned}$$

The property transformer $\mathsf{AddDelay}.S$ takes as input the sequences $\mathbf{u}, \mathbf{x}, \mathbf{y}$, executes S on input $\mathbf{u}_0$ and $\mathbf{x}_0$ producing the values a, and b, and then sets the output $\mathbf{u}', \mathbf{x}', \mathbf{y}'$ where $\mathbf{u}'_0 = \mathbf{u}_0$, $\mathbf{u}'_1 = a$, $\mathbf{y}'_0 = b$, and $\mathbf{x}' = \mathbf{x}$. The components $\mathbf{u}'_2, \mathbf{u}'_3, \ldots$ and $\mathbf{y}'_1, \mathbf{y}'_2, \ldots$ are assigned arbitrary values. The transformer $\mathsf{AddDelay}.S$ returns the value of the second output component y at time zero, but adds a unit delay to the first output component v. $\mathsf{AddDelay}.S$ also adapts the input and the output of S to be of the same type such that we can apply IterateNextOmega to $\mathsf{AddDelay}.S$. Figure 10b gives a graphical representation of $\mathsf{AddDelay}.S$. In this figure the input $\mathbf{x}$ is split in two components $\mathbf{x}_0$ and $\mathbf{x}^1 = (\mathbf{x}_1, \mathbf{x}_2, \ldots)$, input $\mathbf{u}$ is split similarly in $\mathbf{u}_0$ and $\mathbf{u}^1$, output $\mathbf{u}'$ is split in $\mathbf{u}'_0$, $\mathbf{u}'_1$, and $\mathbf{u}'^2$, output $\mathbf{x}'$ is equal to input $\mathbf{x}$, and output $\mathbf{y}'$ is split in $\mathbf{y}_0$ and $\mathbf{y}^1$. From this figure we can also

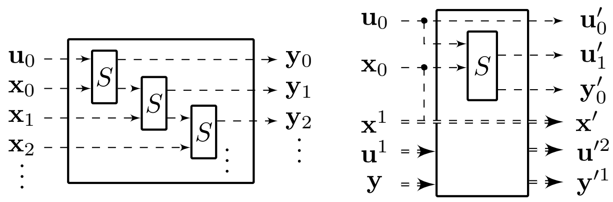

Figure 10: (a) Unit delay feedback of S, (b) $\mathsf{AddDelay}.S$

see that the input components $\mathbf{u}^1$ and $\mathbf{y}$ are not used (they are not connected inside $\mathsf{AddDelay}.S$), and the output components $\mathbf{u}'^2$ and $\mathbf{y}'^1$ are assigned arbitrary values (they are not connected).

Finally we can introduce the transformer $\mathsf{DelayFeedback}.init.S$:

Definition 6. If $init$ is a predicate on the initial value of the feedback variable u and S is a predicate transformer as described above, then unit delay feedback is given by

$$\begin{aligned}\mathsf{DelayFeedback}.init.S = [\mathbf{x} \rightsquigarrow \mathbf{u}, \mathbf{x}, \mathbf{y} \mid init.\mathbf{u}_0] \ ; \\ \mathsf{IterateNextOmega}(\mathsf{AddDelay}.S) \ ; \ [\mathbf{u}, \mathbf{x}, \mathbf{y} \rightsquigarrow \mathbf{y}]\end{aligned}$$

The intuition behind the above definition is that if we plug $\mathsf{AddDelay}.S$ (Figure 10b) into Figure 8, using the tuple $(\mathbf{u}, \mathbf{x}, \mathbf{y})$ instead of $\mathbf{x}$, then we obtain the desired system of Figure 10a.

A major result of this section is that the operators $\mathsf{AddDelay}$ and $\mathsf{DelayFeedback}$ preserve refinement:

Theorem 5. *If S is a predicate transformer as above then*

1. $S \sqsubseteq S' \Rightarrow \mathsf{AddDelay}.S \sqsubseteq \mathsf{AddDelay}.S'$
2. $S \sqsubseteq S' \Rightarrow \mathsf{DelayFeedback}.S \sqsubseteq \mathsf{DelayFeedback}.S'$

The results presented so far in this section apply to any predicate transformer S with the structure described above (inputs u, x and outputs v, y). In practice, we often describe transformers as symbolic transition systems, that is, using a predicate $init$ specifying the set of initial states, a predicate p specifying the set of legal inputs, and a relation r specifying the transition and output relations. If S is obtained from a symbolic transition system, that is, if S has the form $\{u, x \mid p.u.x\} \ ; \ [u, x \rightsquigarrow u', y \mid r.u.u'.x.y]$, then we can prove the following:

Theorem 6. *We have*

$$\begin{aligned}&\mathsf{DelayFeedback}.init.(\{u, x \mid p.u.x\}; [u, x \rightsquigarrow u', y \mid r.u.u'.x.y]) \\ &\quad = \{\mathbf{x} \mid \forall \mathbf{u} : init.\mathbf{u}_0 \Rightarrow (\mathsf{in}.r \,\mathsf{L}\, p).\mathbf{u}.\mathbf{u}^1.\mathbf{x}\} \ ; \\ &\qquad [\mathbf{x} \rightsquigarrow \mathbf{y} \mid \exists \mathbf{u} : init.\mathbf{u}_0 \wedge \Box\, r.\mathbf{u}.\mathbf{u}^1.\mathbf{x}.\mathbf{y}]\end{aligned}$$

This theorem shows that the unit delay feedback of such a predicate transformer is identical to the property transformer semantics of the corresponding symbolic transition system [23], which provides a sanity check of our construction. The condition $(\forall \mathbf{u} : init.\mathbf{u}_0 \Rightarrow (\mathsf{in}.r \,\mathsf{L}\, p).\mathbf{u}.\mathbf{u}^1.\mathbf{x})$ is false on input $\mathbf{x}$, if there exists $\mathbf{u}_0, \ldots, \mathbf{u}_n$ and $\mathbf{y}_0, \ldots, \mathbf{y}_{n-1}$ such that $(\forall i < n : r.\mathbf{u}_i.\mathbf{u}_{i+1}, \mathbf{x}_i, \mathbf{y}_i)$ is true and $p.\mathbf{u}_n.\mathbf{x}_n$ is false. That is, if with $\mathbf{x}$ it is possible to reach a false assertion, then the system will fail from $\mathbf{x}$, even if non-failing choices are possible.

The intuitive behavior of a system as in Theorem 6 was explained in the beginning of this section. Here we elaborate on this intuition further, explaining in particular how it works for non-deterministic and non-input-receptive systems. The system starts with some initial state $\mathbf{u}_0$ and input $\mathbf{x}_0$ (the first element of $\mathbf{x}$) and it checks the input condition $p.\mathbf{u}_0.\mathbf{x}_0$. If the condition is false, then $\mathbf{x}_0$ is an illegal input at state $\mathbf{u}_0$, and execution fails. Otherwise, the system computes non-deterministically the output $\mathbf{y}_0$ and the next state $\mathbf{u}_1$, such that $r.\mathbf{u}_0.\mathbf{u}_1.\mathbf{x}_0.\mathbf{y}_0$ is true. This ends the first execution step. The system then proceeds with the next step, where it uses $\mathbf{u}_1$ and $\mathbf{x}_1$ to test again the input condition p, and so on. At any point during execution, if $p.\mathbf{u}_i.\mathbf{x}_i$ becomes false then the system fails. Otherwise the system outputs non-deterministically the infinite sequence $\mathbf{y}_0, \mathbf{y}_1, \cdots$.

Example 11. [Stateful systems in feedback] Consider the following relations modeling systems with state (u and u' represent current and next state variables, respectively):

$$R_1 \;=\; [u, x \rightsquigarrow u', y \mid u' = u \wedge y = u] \qquad (2)$$
$$R_2 \;=\; [u, x \rightsquigarrow u', y \mid u' = u + 1 \wedge y = u] \qquad (3)$$

In R_1, the next state is equal to the current state. In R_2, the next state increments the current state by 1. We will also use the relation $StepSum$ defined in (1). In all three cases the output y is equal to the state. Let Zero be the predicate given by $\mathsf{Zero}.u = (u = 0)$. We compose the above systems by connecting u' to u in feedback with unit delay. In all three cases we use the predicate Zero as $init$, and in all three cases we use input-receptive components, i.e., where $p = \top$. We get:

1. $\mathsf{DelayFeedback}.\mathsf{Zero}.R_1 = [\mathbf{x} \rightsquigarrow \mathbf{y} \mid \forall i : \mathbf{y}_i = 0]$. Since the initial state is 0, the output is always 0.
2. $\mathsf{DelayFeedback}.\mathsf{Zero}.R_2 = [\mathbf{x} \rightsquigarrow \mathbf{y} \mid \forall i : \mathbf{y}_i = i]$. The output at the i-th step equals i.
3. $\mathsf{DelayFeedback}.\mathsf{Zero}.StepSum = [\mathbf{x} \rightsquigarrow sum.\mathbf{x}]$. The output is the summation of the sequence of inputs seen so far.

The three examples above are all input-receptive and deterministic. Our framework can also handle non-input-receptive and non-deterministic systems, as illustrated next:

Example 12 (A deterministic but non-input-receptive system)**.** Let $R_3 = \{u, x \mid u \leq a\} \ ; \ [u, x \rightsquigarrow u', y \mid u' = u + x \wedge y = u]$. R_3 is similar to $StepSum$, but imposes the condition that at each step the state u is bounded by some constant a. Applying feedback we get:

$$\mathsf{DelayFeedback}.\mathsf{Zero}.R_3 = \{\mathbf{x} | \forall i : sum.\mathbf{x}.i \leq a\} \ ; \ [\mathbf{x} \rightsquigarrow sum.\mathbf{x}]$$

Note that the local (i.e., at each step) condition $u \leq a$ becomes (after feedback) the global condition that the sum of all elements $\mathbf{x}_i$ of every prefix of the input $\mathbf{x}$ must be less or equal to a.

Example 13 (A non-deterministic and non-input-receptive system)**.** We can weaken also the relation $u' = u + x \wedge y = u$ and make the previous system non-deterministic. For example at each step we can choose to compute $u' = u + x$ or $u' = x$. Let $R_4 = \{u, x \mid u \leq a\} \ ; \ [u, x \rightsquigarrow u', y | (u' = u + x \vee u' = x) \wedge y = u]$. Then we get:

$$\begin{aligned}\mathsf{DelayFeedback}.\mathsf{Zero}.R_4 = \{\mathbf{x} \mid \forall i : sum.\mathbf{x}.i \leq a\} \ ; \\ [\mathbf{x} \rightsquigarrow \mathbf{y} \mid \mathbf{y}_0 = 0 \wedge (\forall i : \mathbf{y}_{i+1} = \mathbf{y}_i + \mathbf{x}_i \vee \mathbf{y}_{i+1} = \mathbf{x}_i)]\end{aligned}$$

Observe that the global condition on $\mathbf{x}$ remains the same. This is because in the worst case the non-deterministic choice could always pick the alternative $u' = u + x$, which leads to higher numbers stored in u compared to the alternative $u' = x$.

8. Symbolic Computation

For a relation $R : ((A_{1\bot} \times \ldots \times A_{k\bot}) \times X)^\bullet \to ((A_{1\bot} \times \ldots \times A_{k\bot}) \times Y)^\bullet$, in the $\exists n$ quantifier from the definition of $\mathsf{InstFeedback}.R$, n is bound by k. This is so because a strictly increasing sequence $u_0 < u_1 < \ldots$ on $A_{1\bot} \times \ldots \times A_{k\bot}$ can have at most $k + 1$ elements. Because of this property we have:

$$(\mathsf{fb\_a}.R)^* = \bigvee_{i \leq k} (\mathsf{fb\_a}.R)^i$$

so we can calculate the feedback symbolically using

$$\mathsf{InstFeedback}.R = \mathsf{fb\_begin} \circ (\bigvee_{i \leq k} (\mathsf{fb\_a}.R)^i) \circ \mathsf{fb\_b}.R$$

If relation R is defined using a first order formula, then $\mathsf{InstFeedback}.R$ is also a first order formula.

Checking refinement of two relations with $R \sqsubseteq R'$ defined by first order formulas is also reduced to checking validity of a first order formula $(\forall x : R.x. \bullet \vee(\forall y : R'.x.y \Rightarrow R.x.y)$.

Refinement of predicate transformers of the form $\{p\}$; $[r]$ are also reduced to checking validity of a first order formula (Theorem 1). The predicate transformers of the form $\{p\}$; $[r]$ are closed to serial and demonic compositions, fusion, and product, so if p and r are defined using first order formulas, then so are the results of these operations (Theorem 1 and 2).

The delay feedback of a predicate transformer of the form $\{p\}$; $[r]$ is equivalent by Theorem 6 to a property transformer of the form $\{P\}$; $[R]$ where P and R are quantified LTL formulas. Checking the refinement $\mathsf{DelayFeedback}.S \sqsubseteq \mathsf{DelayFeedback}.S'$, is reduced by Theorem 5 to checking the refinement $S \sqsubseteq S'$ of predicate transformers.

9. Implementation in Isabelle

We have implemented all results presented in this paper, as well as in the extended version [24], in the Isabelle/HOL theorem prover. All results are stated and proven in Isabelle (about 5k lines of code). Our theory is built on top of theories for refinement calculus and a semantic formalization of LTL. Proofs omitted from this paper can be found in the Isabelle code, available from `http://users.ics.aalto.fi/viorel/FeedbackIsabelle.zip`. The code has been tested with versions 2015 and 2016-RC1 of Isabelle.

10. Conclusions

Feedback composition is challenging to define for non-deterministic and non-input-receptive systems, so that refinement is preserved. In this paper we make progress toward this goal. First, we propose an instantaneous feedback composition operator for partial relations with fail and unknown (Section 5). This operator generalizes constructive semantics which are limited to total functions [4, 10, 17]. Second, we propose a feedback-with-unit-delay operator for systems with state (Section 7). Both operators preserve refinement (Theorems 3 and 5).

Our framework can be applied to arbitrary Simulink diagrams by first translating them into predicate transformers that handle current and next state variables as inputs and outputs, respectively, and then applying feedback with unit delay on these variables. Contrary to Simulink, our construction allows instantaneous feedback as well as non-deterministic and non-input-receptive components.

Open problems remain. Seeking a generalization of the feedback operators directly to property transformers, at the semantic level, is a worthwhile albeit difficult goal. With this generalization, we will be able to treat Simulink basic blocks as property transformers, and apply serial and feedback compositions directly.

References

[1] R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[2] R.-J. Back and M. Butler. Exploring summation and product operators in the refinement calculus. In *Mathematics of Program Construction*, volume 947 of *LNCS*. Springer, 1995.

[3] R.-J. Back and J. von Wright. *Refinement Calculus. A systematic Introduction*. Springer, 1998.

[4] G. Berry. The Constructive Semantics of Pure Esterel, 1999.

[5] M. Broy and K. Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer, 2001.

[6] B. Davey and H. Priestley. *Introduction to lattices and order*. Cambridge University Press, New York, second edition, 2002. ISBN 0-521-78451-4.

[7] L. de Alfaro. Game models for open systems. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 192–213. Springer, 2004.

[8] L. de Alfaro and T. Henzinger. Interface automata. In *Foundations of Software Engineering (FSE)*. ACM Press, 2001.

[9] L. Doyen, T. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *EMSOFT*, pages 79–88, 2008.

[10] S. Edwards and E. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comp. Progr.*, 48:21–42(22), July 2003.

[11] T. Freeman and F. Pfenning. Refinement Types for ML. *SIGPLAN Not.*, 26(6):268–277, May 1991.

[12] X. Jin, J. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Benchmarks for model transformations and conformance checking. In *1st Intl. Workshop on Applied Verification for Continuous and Hybrid Systems (ARCH)*, 2014. Benchmark Simulink models available from `http://cps-vo.org/group/ARCH/benchmarks`.

[13] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Powertrain control verification benchmark. In *17th Intl. Conf. on Hybrid Systems: Computation and Control*, HSCC '14, pages 253–262, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2732-9. doi: 10.1145/2562059.2562140.

[14] B. Jonsson. A fully abstract trace model for dataflow and asynchronous networks. *Distrib. Comput.*, 7(4):197–212, 1994. ISSN 0178-2770.

[15] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, Proceedings of IFIP Congress 74*. North-Holland, 1974.

[16] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[17] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. Computer-Aided Design*, 13(7):950–956, 1994.

[18] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-262-63132-6.

[19] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[20] G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

[21] A. Pnueli. The temporal logic of programs. In *FOCS*, 1977. doi: 10.1109/SFCS.1977.32.

[22] V. Preoteasa. Formalization of refinement calculus for reactive systems. *Archive of Formal Proofs*, Oct. 2014. ISSN 2150-914x. `http://afp.sf.net/entries/RefinementReactive.shtml`.

[23] V. Preoteasa and S. Tripakis. Refinement calculus of reactive systems. In *EMSOFT*. ACM, 2014.

[24] V. Preoteasa and S. Tripakis. Towards Compositional Feedback in Non-Deterministic and Non-Input-Receptive Systems. *ArXiv e-prints*, Oct. 2015. `arXiv:1510.06379`.

[25] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, June 2008. ISSN 0362-1340.

[26] S. Tripakis and C. Shaver. Feedback in Synchronous Relational Interfaces. In *From Programs to Systems*, volume 8415 of *LNCS*, pages 249–266. Springer, 2014.

[27] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. A Theory of Synchronous Relational Interfaces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(4), July 2011.

[28] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227. ACM, 1999.