Refinement Calculus of Reactive Systems*

Viorel Preoteasa
Aalto University, Finland.

Stavros Tripakis
Aalto University, Finland.
University of California, Berkeley, USA.

ABSTRACT

Refinement calculus is a powerful and expressive tool for reasoning about sequential programs in a compositional manner. In this paper we present an extension of refinement calculus for reactive systems. Refinement calculus is based on monotonic predicate transformers, which transform sets of post-states into sets of pre-states. To model reactive systems, we introduce monotonic property transformers, which transform sets of output infinite sequences into sets of input infinite sequences. We show how to model in this semantics refinement, sequential composition, demonic choice, and other semantic properties of reactive systems. We also show how such transformers can be defined by various formalisms such as linear temporal logic formulas (suitable for specifications) and symbolic transition systems (suitable for implementations). Finally, we show how this framework generalizes previous work on relational interfaces to systems with infinite behaviors and liveness properties.

1. INTRODUCTION

Refinement calculus [3, 5] is a powerful and expressive tool for reasoning about sequential programs. Refinement calculus is based on a *monotonic predicate transformer* semantics which allows to model total correctness (functional correctness and termination), unbounded nondeterminism, demonic and angelic nondeterminism, among other program features. The framework also allows to express compatibility during program composition (e.g., whether the postcondition of a statement is strong enough to guarantee the precondition of another) and also to reason about program evolution and substitution via refinement.

As an illustrative example, consider a simple assignment statement performing division: $z := x/y$. Semantically, this statement is modeled as a predicate transformer, denoted Div. Div is a function which takes as input a predicate q characterizing a set of program states and returns a new predicate p such that if the program is started in any state in p it is guaranteed to terminate and reach a state in q (that is, p is the *weakest precondition* of q). For our division example, we would also like to express the fact that division by zero is not allowed. To achieve this, we can define the predicate transformer as follows:

$\mathsf{Div}(q) = \{(x,\ y,\ z) \mid y \neq 0 \wedge (x,\ y,\ x/y) \in q\}$.

Having defined the semantics of the division statement, we can now compose it with another statement, say, a statement that reads the values of x and y from the console: $(x, y) := read()$. Making no assumptions on what *read* does, we model it as the so-called Havoc statement, which assigns arbitrary values to program variables. Formally, *read* is modeled as the predicate transformer: $\mathsf{Havoc}(q) = \begin{cases} \top & \text{if } q = \top \\ \bot & \text{otherwise} \end{cases}$ where $\top$ and $\bot$ denote the universal and empty sets, respectively. Now, what happens if we compose the two statements in sequence? That is, $(x, y) := read(); z := x/y$. Refinement calculus tells us that sequential composition of statements corresponds to function composition of their predicate transformers, so the semantics of the composition is $\mathsf{Havoc} \circ \mathsf{Div}$, which can be shown to be equivalent to the predicate transformer Fail, defined as $\mathsf{Fail}(q) = \bot$ for any q. This indicates incompatibility, i.e., the fact that the composition of the two statements is invalid. Indeed, without any assumptions on *read*, we cannot guarantee absence of division by zero.

We can go one step further and reason about program substitution via refinement. Assume we have another division statement, but this time it calculates only some approximation of the result: $z := z'$ such that $abs(x/y - z') \leq \epsilon$. We model this new division statement as a new predicate transformer Div' defined as follows: $\mathsf{Div}'(q) = \{(x,\ y,\ z) \mid y \neq 0 \wedge (\forall z' : abs(x/y - z') \leq \epsilon \Rightarrow (x,\ y,\ z) \in q)\}$. Refinement calculus allows us to state and prove that Div refines Div', and conclude that the Div statement can substitute the Div' statement without affecting the properties of the overall program.

Refinement calculus has been developed so far primarily for sequential programs. In this paper we present an extension of refinement calculus for *reactive systems* [11]. Denotationally, a reactive system can be seen as a system which accepts as input infinite sequences of values, and produces as output infinite sequences of values. Operationally, a reactive system can be seen as a machine with input, output, and state variables, which operates in steps, each step consisting of reading the inputs, writing the outputs, and updating the state. Our framework allows us to specify a very large class of reactive systems, including nondeterministic and non-receptive systems, with both safety and liveness properties, both denotationally and operationally. It also allows to define system composition and to talk about incompatibility, refinement, and so on. To illustrate these features, we provide an example analogous to the division example above.

*This work was supported in part by the Academy of Finland and by the NSF via projects *COSMOI: Compositional System Modeling with Interfaces* and *ExCAPE: Expeditions in Computer Augmented Program Engineering*

ESWEEK'14, October 12 - 17 2014, New Delhi, India

http://dx.doi.org/10.1145/2656045.2656068

Consider the two components shown in Figure 1 and specified using the syntax of linear temporal logic [15]. Component $A = \Box\,(x \geq 0)$ specifies that its output x is never less than zero, while component $B = \Box \Diamond\,(x = 1)$ requires that its input is infinitely often equal to one. The output of A is connected to the input of B. Using our framework, we can show that this composition is invalid, that is, that A and B are incompatible, because the output guarantee of A is not strong enough to satisfy the input requirement of B.

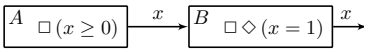

Figure 1: Two incompatible reactive systems

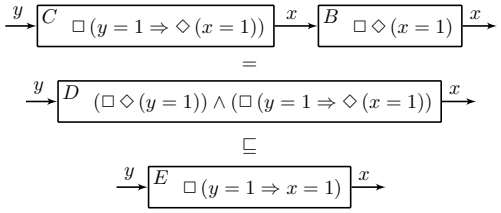

Figure 2: Two compatible systems (top), their composition (middle), and a refinement (bottom)

The above is akin to behavioral type checking. We can also use our framework to perform behavioral type inference. We can deduce, for instance, that component $C = \Box\,(y = 1 \Rightarrow \Diamond\,(x = 1))$ from Figure 2, which models a request-response property (always $y = 1$ implies eventually $x = 1$) is compatible with component B above, and infer automatically a new input requirement $\Box \Diamond\,(y = 1)$ for the composite system D.

Finally, we can reason about refinement, akin to behavioral subtyping. In the example of Figure 2, we can show that the executable component E which sets output $x = 1$ whenever input $y = 1$ refines the component D, and therefore conclude that E can substitute D in any context.

The key technical contribution of our paper, which allows us to develop a refinement calculus of reactive systems, is the notion of *monotonic property transformers*. A property transformer is a function which takes as input an *output property* q and returns an *input property* p. Properties are sets of traces, so that q is a set of output traces and p is a set of input traces. In other words, similarly to predicate transformers, which transform postconditions to preconditions, property transformers transform *out-conditions* to *in-conditions*.

Monotonic property transformers (MPTs) provide the semantical foundation for system specification and implementation in our framework. We generally use higher order logic to specify MPTs, but we also show how to MPTs can be defined using formalisms more amenable to automation, such as linear temporal logic and *symbolic transition systems* (similar to the formalism used by the popular model-checker NuSMV). We also provide the basic operations on MPTs: composition, compatibility, refinement, variable hiding, and so on. We study subclasses of MPTs specified by input-output relations, and derive a number of interesting closure and other properties on them. Finally, as an application of our framework, we show how it can be used to extend the relational interfaces framework of [17] from only safety (finite, prefix-closed) properties, to also infinite properties and liveness.

In the sequel we use higher order logic as implemented in Isabelle/HOL [14] to express our concepts. All results presented in this paper were formalized in Isabelle, and our presentation translates directly into Isabelle's formal language.

1.1 Related work

A number of compositional frameworks for the specification and verification of input-output reactive systems have been proposed in the literature. In the Focus framework [8] specifications are relations on input-output streams. Focus is able to express infinite streams and liveness properties, however, it focuses on *input-receptive* systems. Other compositional frameworks that assume input-receptiveness are Dill's *trace theory* [10], *IO automata* [13], and *reactive modules* [2]. Our framework allows to specify non-input-receptive systems, that is, systems which specify some of their inputs to be illegal. As argued in [17], the ability to specify illegal inputs enables lightweight verification, akin to type-checking. In particular, it allows to define a behavioral notion of compatibility during system composition, which goes beyond syntactic compatibility (correct port matching) as illustrated by the examples given above.

There are also compositional frameworks which allow to specify non-input-receptive systems. Among such frameworks, our work is inspired from refinement calculus, on one hand, and *interface theories* on the other, such as *interface automata* [9] and *relational interfaces* [17]. These interface theories, however, cannot express liveness properties. The same is true with existing extensions of refinement calculus to infinite behaviors such as *action systems* [4, 6], which do not have acceptance conditions (say, of type Büchi) and therefore cannot express general liveness properties. *Fair* action systems [7], augment action systems with fairness assumptions on the actions, but it is unclear whether they can handle general liveness properties, e.g., full LTL. Our approach is based on a natural generalization from predicate to property transformers, and as such can handle liveness (and more) as part of system specification.

The Temporal Logic of Actions [12] can be used to specify systems with liveness properties, but their semantics is based on sets of infinite traces, and it does not seem possible to express illegal inputs and compatibility. Event B [1] is a specification method based on refinement calculus and action systems, and it supports building of systems using stepwise refinement of events. Event B models can be shown to preserve (bounded) liveness properties [18], but it is also unclear whether they can be used to specify systems with liveness properties.

2. PRELIMINARIES

We use capital letters X, Y, Σ, ... to denote types, and small letters to denote elements of these types $x \in X$,.... We denote by Bool the type of the Boolean values true and false, and by Nat the type of natural numbers. We use in general the sans-serif font to denote constants (types and elements). We use $\wedge$, $\vee$, $\Rightarrow$, $\neg$ for the Boolean operations.

If X and Y are types, then $X \to Y$ denotes the type of functions from X to Y. We use a *dot notation* for function application, so we write $f.x$ instead of $f(x)$ from now on. If $f : X \to Y \to Z$ is a function which takes the first argument from X and the second argument from Y and the result is from Z, and if $x \in X$ and $y \in Y$ then $f.x.y$ denotes the function f applied to x and applied to y. According to this interpretation function application associates to left ($f.x.y = (f.x).y$) and correspondingly the function type constructor ($\to$) associates to right ($X \to Y \to Z = X \to (Y \to Z)$). We use also lambda notation for constructing functions. For example if $x + y + 2 \in \mathsf{Nat}$ is natural expression then $(\lambda x,\ y : x + y + 2) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$ is the function which maps x and y to $x + y + 2$. We use the notation $X \times Y$ for the Cartesian product of X and Y, and if $x \in X$ and $y \in Y$, then $(x,\ y)$ is a pair from $X \times Y$.

Predicates are functions with Boolean values with one or more

arguments ($p : X \to Y \to \mathsf{Bool}$), and relations are predicates with at least two arguments. For a relation $r : X \to Y \to \mathsf{Bool}$ we denote by $\mathsf{in}.r : X \to \mathsf{Bool}$ the predicate given by

$$\mathsf{in}.r = (\exists y : r.x.y)$$

If r is a relation with more than two arguments then we define $\mathsf{in}.r$ similarly by quantifying over the last argument of r:

$$\mathsf{in}.r.x.y.z = (\exists u : r.x.y.z.u)$$

We extend point-wise the operations on Bool to operations on predicates. For example, if p is a predicate only on x, i.e., $p : X \to \mathsf{Bool}$ and q is a predicate on x and y, i.e., $q : X \to Y \to \mathsf{Bool}$, then:

$$(p \wedge q).x.y = p.x \wedge q.x.y$$

and we also have in this case:

$$p \wedge (\mathsf{in}.q) = \mathsf{in}.(p \wedge q)$$

We use $\bot$ and $\top$ as the smallest and greatest predicates

$$\bot.x = \mathsf{false} \text{ and } \top.x = \mathsf{true}$$

The composition of relations $r,\ r'$ is denoted $r \circ r'$ and it is a relation given by:

$$(r \circ r').x.z = (\exists y : r.x.y \wedge r.y.z)$$

We treat subsets of a type, and predicates with one argument as being the same and we use both notations $x \in p$ and $p.x$ to express the fact that p is true in x. For constructing predicates we use lambda abstraction ($\lambda x,\ y : x \leq 10 \Rightarrow y > 10$), and for predicates with single arguments we use also set comprehension $\{x \mid x > 10\}$.

We assume that Σ is a type of program states. For example for imperative programs over some variables $x, y, z, \ldots$, a state $s \in \Sigma$ gives values to the program variables $x,\ y,\ z,\ \ldots$. In general, the systems that we consider may have different input and output variables, and we can also have different state sets. For a system with a variable x, Σ_x denotes the type of states which gives values to x. For a state $s \in \Sigma$, $x.s$ is the value of x in s and $s[x := a]$ is new state obtained from s by changing the value of x to a.

For reactive systems we model states as *infinite sequences* or *traces* from Σ. Formally such an infinite sequence is an element $\sigma \in \Sigma^\omega = (\mathsf{Nat} \to \Sigma)$. For $\sigma \in \Sigma^\omega$, $\sigma_i = \sigma.i$, and $\sigma^i \in \Sigma^\omega$ is given by $\sigma^i_j = \sigma^i.j = \sigma_{i+j}$. We consider pair of traces (σ, σ') as being the same as traces of pairs $(\lambda i : (\sigma_i, \sigma'_i))$.

In the next subsection we introduce the linear temporal logic which is the main logic that we use to specify reactive systems.

2.1 Linear temporal logic

Linear temporal logic (LTL) [15] is a logic used for specifying properties of reactive systems. In addition to the connectives of classical logic it contains modal operators referring to time. LTL formulas can express temporal properties like something is always true, or something is eventually true, and their truth values are given for infinite sequences of states. For example the formula $\Box\, x = 1$ (always x is equal to 1) is true for the infinite sequence σ if for all $i \in \mathsf{Nat}\ x.\sigma_i = 1$.

The semantics of an LTL formula is the set of all sequences for which the formula is true. In this paper we use a semantic (algebraic) version of LTL. For us 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 $p,\ q \in \Sigma^\omega \to \mathsf{Bool}$ are properties, then *always* p, *eventually* p, *next* p, and p *until* q are also properties and they are denoted by $\Box\, p$, $\Diamond\, p$, $\bigcirc\, p$, and $p\ \mathsf{U}\ q$ respectively. The property $\Box\, p$ is true in σ if p is true at all time points in σ, $\Diamond\, p$ is true in σ if p is true at some time point in σ, $\bigcirc\, p$ is true in σ if p is true at the next time point in σ, and $p\ \mathsf{U}\ q$ is true in σ if there is some time in σ when q is true, and until then p is true. Formally we have:

$$\begin{array}{lcl}(\Box\, p).\sigma & = & (\forall n : p.\sigma^n)\\ (\Diamond\, p).\sigma & = & (\exists n : p.\sigma^n)\\ (\bigcirc\, p).\sigma & = & p.\sigma^1\\ (p\ \mathsf{U}\ q).\sigma & = & (\exists n : (\forall i < n : p.\sigma^i) \wedge q.\sigma^n)\end{array}$$

Quantification for properties is defined in the following way

$$(\forall x : p).\sigma = (\forall a : p.(\sigma[x := a]))$$

where a ranges over infinite traces of x values, and $\sigma[x := a].i = \sigma_i[x := a_i]$. When p is a predicate on traces x and y, then quantification is defined as normally in predicate calculus, as in $\forall a : p.a.b$.

We lift normal arithmetic and logical operations to traces (x and y) in the following way

$$x + y = x_0 + y_0 \ \text{ and } \ x \wedge y = x_0 \wedge y_0$$

Lemma 1. *If p and q are properties, then we have:* $(\exists x : \Box\, p) = \Box\, (\exists x : p)$ *and* $\Box\, (\mathsf{in}.p) = \mathsf{in}.(\Box\, p)$.

Definition 2. 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.

Lemma 3. *If p and q are properties, then we have*

1. $(p\ \mathsf{L}\ q).\sigma = (\forall n : (\forall i < n : p.\sigma^i) \Rightarrow q.\sigma^n)$
2. $p\ \mathsf{L}\ p = \Box\, p$ and $\mathsf{true}\ \mathsf{L}\ p = \Box\, p$

Using LTL properties we can express *safety* properties, expressing that *something bad never happens* (e.g., $\Box\, t \leq 10^\circ$ – the temperature stays always below 10°), as well as *liveness* properties, expressing that *something good eventually happens* (e.g., $\Box \Diamond\, x = 0$ – infinitely often x becomes 0).

3. MONOTONIC PROPERTY TRANSFORMERS

Monotonic *predicate* transformers are a powerful formalism for modeling programs. A program S from state space Σ_1 to state space Σ_2 is formally modeled as a monotonic predicate transformer, that is, a monotonic function from $(\Sigma_2 \to \mathsf{Bool}) \to (\Sigma_1 \to \mathsf{Bool})$, with a weakest precondition interpretation. If S is a program and $q \in \Sigma_2 \to \mathsf{Bool}$ is a predicate on Σ_2 (set of final states), then $S.q$ is the set of all initial states from which the execution of S always terminates and it terminates in a state from q. *Monotonic Boolean transformers* (MBTs) [16] is a generalization of monotonic predicate transformers, where instead of predicates $(\Sigma_i \to \mathsf{Bool})$ arbitrary complete Boolean algebras are used. MBTs are monotonic functions from a complete Boolean algebra B_2 to a complete Boolean algebra B_1.

In this section we introduce *monotonic property transformers* (MPTs), and we use them to model input-output reactive systems. MPTs are MBTs from the complete Boolean algebra of Σ_y properties ($\Sigma_y^\omega \to \mathsf{Bool}$) to the complete Boolean algebra Σ_x properties ($\Sigma_x^\omega \to \mathsf{Bool}$), where x and y are the input and output variables, respectively. If S is a reactive system with input variable x and output variable y, then a *legal execution* of S takes as input a sequence of values for x, $\sigma = x_0,\ x_1,\ \ldots$, and produces a sequence of values for y, $\sigma' = y_0,\ y_1,\ \ldots$. This execution may be nondeterministic, that is, for the same input values σ we can obtain different

output values σ'. The execution of S from σ may also *fail* if σ does not satisfy some requirements on the input variables. As a property transformer, the system S is applied to a property $q \in \Sigma_y^\omega \to \mathsf{Bool}$, i.e., to a set of sequences over the output variable y. Then, S returns the set of all sequences over the input variable x from which all executions of S do not fail and produce sequences in q.

Informally we say that the system S has *local execution*, or it is *local* if S can be executed step by step. That is, starting with input x_0, S can choose output y_0 and possibly go to a new state, and then from the current state with input x_1 can choose the next output $y_1$and the next state, and so on. The system S is *executable* if it is local and deterministic, that is, the output and the next state is a function of the input and the current state. We will define later formally local and deterministic reactive systems. Using monotonic property transformers we can model both specifications of reactive systems as well as local and executable systems, and we can use refinement to construct executable systems that refine abstract specifications.

Using property transformers, we can specify systems that exhibit various properties that we are interested in. For example the specification of a system S that guarantees the *liveness* property that the output Boolean variable y is true infinitely often regardless of the input, is given by

$$S.\{y \mid \Box \Diamond y\} = \top$$

Note that the above equation does not define S completely, it only specifies a constraint that S must satisfy. This is useful when we want to specify some properties that the system must satisfy, without completely defining the system itself. Below, in Section 3.1 we give a complete definition of a MPT which satisfies the requirement above.

Similarly, the specification of a system S' that guarantees the liveness property that the output Boolean variable y is true infinitely often when the integer input variable x is equal to one infinitely often, is given by

$$\{x \mid \Box \Diamond x = 1\} \subseteq S'.\{y \mid \Box \Diamond y\}$$

3.1 Basic operations on monotonic property transformers

The point-wise extension of the Boolean operations to properties, and then to monotonic property transformers gives us a complete lattice with the operations $\sqsubseteq$, $\sqcap$, $\sqcup$, $\bot$, $\top$ (observe that $\sqcap$ and $\sqcup$ preserve monotonicity). All these simple lattice operations are also meaningful as operations on reactive systems. The order of this lattice ($S \sqsubseteq T \Leftrightarrow (\forall q : S.q \subseteq T.q)$) gives the *refinement relation* of reactive systems. If $S \sqsubseteq T$, then we say that T *refines* S (or S is *refined* by T). If T refines S then we can replace S with T in any context. This interpretation of the lattice order as refinement follows from the modeling of reactive systems as monotonic property transformers. For example if we assume that S and S' introduced above are completely defined by

$$S.q = \begin{cases} \top & \text{if } \{y \mid \Box \Diamond y\} \subseteq q \\ \bot & \text{otherwise} \end{cases}$$

and

$$S'.q = \begin{cases} \{x \mid \Box \Diamond x = 1\} & \text{if } \{y \mid \Box \Diamond y\} \subseteq q \\ \bot & \text{otherwise} \end{cases}$$

then S and S' are monotonic and S refines S' ($S' \sqsubseteq S$). In this example we see that if S' is used within some context where for certain inputs it guarantees outputs where y is true infinitely often, then S can replace S' because S guarantees the same property of the output regardless of its input.

The operations $\sqcap$ and $\sqcup$ model (*unbounded*) *demonic* and *angelic nondeterminism* or *choice*. The interpretation of the demonic choice is that the system $S \sqcap T$ is correct (i.e., satisfies its specification) if both S and T are correct. In this choice someone else (the demon) can choose to execute S or T, so they must both be correct. On the other hand the angelic choice $S \sqcup T$ is correct if one of the systems S and T are correct. In this choice we have the control over the choice, and we assume that we always choose the correct alternative. Unbounded nondeterminism means that we could have unbounded choices as for example in $\bigsqcap_{i \in I} S_i$ where I is infinite. For example if we have two systems S and S' which compute in x the factorial of n, but S computes the correct result only for $n \leq 20$ and S' computes the correct result only for $10 \leq n$. Formally we have

$$\begin{array}{lcl} S.\{x \mid x = n!\} & = & \{n \mid n \leq 20\} \\ S'.\{x \mid x = n!\} & = & \{n \mid 10 \leq n\} \end{array}$$

The demonic choice of S and S' is capable of computing the factorial only for numbers between 10 and 20, while the angelic choice will compute the factorial for all natural numbers n.

We denote the bottom and the top of the lattice of property transformers by Fail and Magic respectively. The transformer Fail does not guarantee any property. For any property q, we have $\mathsf{Fail}.q = \bot$, 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.

Sequential composition of two systems S and T is simply the functional composition of S and T ($S \circ T$). For reactive systems 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 composition

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

Definition 4. Two systems S and T are *incompatible* (w.r.t. the sequential composition $S \ ; \ T$) if $S \ ; \ T = \mathsf{Fail}$.

Intuitively, S and T are compatible if the outputs of S can be controlled so that they are legal inputs for T. Controlling the outputs of S might mean restricting its own legal inputs.

If for example we have S and T given by

$$S.q = \begin{cases} \top & \text{if } \{x \mid x > 5\} \subseteq q \\ \bot & \text{otherwise} \end{cases} \text{ and } T.q = \{x \mid x < 10\}$$

then $(S \ ; \ T).q = S.(T.q) = S.\{x \mid x < 10\} = \bot$, for any q. Therefore, S and T are in this case incompatible. This is because T requires its input to be smaller than 10, but S can only guarantee that its output will be greater than 5, and there is no way to restrict the input of S to make this guarantee stronger.

On the other hand, assuming that the input and output of S and T have the same type, T and S are compatible w.r.t. the reverse composition, i.e., $T \ ; \ S$ is not Fail. Indeed, we have $T.(S.q) = \{x \mid x < 10\}$, for any q.

Definition 5. For a property transformer S, the *guard* of S, denoted $\mathsf{grd}.S$, is the set of input sequences for which the system does not behave *miraculously*. Formally:

$$\mathsf{grd}.S = \neg S.\bot$$

For example the guard of Magic is $\bot$ and the guard of Fail is $\top$.

Definition 6. For a property transformer S, the *fail* of S, denoted $\mathsf{fail}.S$, is the set of *illegal* input sequences, i.e., the set of input sequences for which the system produces no output, or *fails* to establish any output property. Formally:

$$\mathsf{fail}.S = \neg S.\top$$

For example the fail of Magic is $\bot$ and the fail of Fail is $\top$.

3.2 Hiding local variables in property transformers

In this subsection we introduce a hiding operator on property transformers that we will use later to hide local variables. The theorem of this subsection will allow us to compose two property transformers, each having its own local variable, into a property transformer with a pair of local variables, one component for each system in the composition.

Definition 7. If $S.a$ is a property transformer depending on the parameter $a \in A$, then $\mathsf{hide}.A.S$ is the property transformer given by

$$\mathsf{hide}.A.S = \bigsqcap_{a \in A} S.a$$

We will use the hide operator to hide local variables in property transformers. If S is a property transformer expression containing free the variable a the we use the notation $(\mathsf{hide}\ a \in A : S)$ for $\mathsf{hide}.A.(\lambda a : S)$.

For example assume that we have the assignment $x := x + a$ where x is a program variable, and a some value. Then the property transformer associated to this assignment is

$$S.a.q.x = q.(x + a)$$

That is, x is in the set of initial values of $S.a$ for establishing q if $x + a$ is in q. We have

$$\mathsf{hide}.A.S.q.x = (\forall a \in A : q.(x + a))$$

If $q = \{x \mid x \leq 10\}$, then $\mathsf{hide}.A.S.q.x$ is true if $x + a \leq 10$ for all $a \in A$.

Definition 8. If S is a property transformer, then S is conjunctive if $S.(\bigcap Q) = \bigcap(S.Q)$ for all sets of properties Q, and S is strictly conjunctive if $S.(\bigcap Q) = \bigcap(S.Q)$ for all nonempty set of properties Q.

Lemma 9. *If S and T are (strictly) conjunctive, then S ; T is (strictly) conjunctive.*

Theorem 10. *If $S.a$ is conjunctive for all $a \in A$, then*

$$(\mathsf{hide}.A.S)\ ;\ (\mathsf{hide}.B.T) = \mathsf{hide}.(A \times B).(\lambda(u,\ v) : (S.u)\ ;\ (T.v))$$

Moreover if $S.a$ is strictly conjunctive for all $a \in A$, and B is nonempty then

$$(\mathsf{hide}.A.S)\ ;\ (\mathsf{hide}.B.T) = \mathsf{hide}.(A \times B).(\lambda(u,\ v) : (S.u)\ ;\ (T.v))$$

3.3 Assert and demonic update transformers

We now define two special types of property transformers which will be used to form more general property transformers by composition. For $p,\ q \in \Sigma^\omega \to \mathsf{Bool}$ and $r \in \Sigma_1^\omega \to \Sigma_2^\omega \to \mathsf{Bool}$ we define the *assert property transformer* $\{p\} : (\Sigma^\omega \to \mathsf{Bool}) \to (\Sigma^\omega \to \mathsf{Bool})$, and the *demonic update property transformer* $[r] : (\Sigma_2^\omega \to \mathsf{Bool}) \to (\Sigma_1^\omega \to \mathsf{Bool})$ as follows:

$$\begin{array}{ll} \{p\}.q & = p \cap q \\ [r].q.\sigma & = (\forall \sigma' : r.\sigma.\sigma' \Rightarrow q.\sigma') \end{array}$$

The assert system $\{p\}$ when executed from a sequence σ, behaves as skip when $p.\sigma$ is true, and it fails otherwise. The demonic update statement $[r]$ establishes a post condition q when starting in a sequence σ if all sequences σ' with $r.\sigma.\sigma'$ are in q.

If R is an expression in x and y, then $[x \rightsquigarrow y \mid R] = [\lambda x, y : R]$. For example if R is $z = x + y$, then $[x,\ y \rightsquigarrow z \mid z = x + y] = [\lambda(x,\ y),\ z : z = x + y]$ is the system which produces in the output z the value $x + y$ where x and y are the inputs. For assert statements we introduce similar notation. If P is an expression in x then $\{x \mid P\} = \{\lambda x : P\}$. If P is $x \leq y$, then $\{x,\ y \mid x \leq y\} = \{\lambda(x, y) : x \leq y\}$. The variables x and y are bounded in $[x \rightsquigarrow y \mid R]$ and $\{x \mid P\}$, however when we compose some of these property transformers we will try whenever possible to use the same name for the output variables of a transformer which are inputs to another transformer. For example we will use the notation:

$$\{x,\ y \mid x \leq y\}\ ;\ [x,\ y \rightsquigarrow z \mid z = x + y]\ ;\ [z \rightsquigarrow u \mid u = z^2]$$

instead of the equivalent one:

$$\{x,\ y \mid x \leq y\}\ ;\ [u,\ v \rightsquigarrow x \mid x = u + v]\ ;\ [u \rightsquigarrow x \mid x = u^2]$$

Sometimes we also need demonic transformers that copy some of the input variables into some of the output variables:

$$S = [u,\ x \rightsquigarrow y,\ v \mid (x = y) \wedge r.u.x.y.v]$$

In this case we drop the condition $x = y$ from the relation of S and we use the same name for x and y:

$$S = [u,\ x \rightsquigarrow x,\ v \mid r.u.x.x.v]$$

If we want to rearrange the input variables into the output variables and if we want to drop some input variables and introduce some new arbitrary variables, then we use the syntax:

$$S = [x,\ y,\ u,\ z,\ x \rightsquigarrow z,\ y,\ x,\ y,\ v]$$

This notation stands for

$$\begin{array}{c} S = [\lambda(x,\ y,\ u,\ z,\ x'),\ (z',\ y',\ x'',\ y'',\ v) : \\ x = x' = x'' \wedge y = y' = y'' \wedge z = z'] \end{array}$$

If S starts on a tuple where the first component is the same as the last component ($x = x'$), then S terminates normally and it chooses z', y', x'', y'', v such that $x = x' = x'' \wedge y = y' = y'' \wedge z = z'$. On the other hand if S starts on a tuple where the first component is different from the last component, then S terminates miraculously.

4. RELATIONAL PROPERTY TRANSFORMERS

Definition 11. A *relational property transformer* (RPT) is a property transformer of the form $\{p\}\ ;\ [r]$. The assert transformer $\{p\}$ imposes the restriction p on the input sequences, and the demonic update $[r]$ nondeterministically chooses output sequences according to the relation r. For a RPT $S = \{p\}\ ;\ [r]$ we call p the *precondition* of S and r the *input-output relation* of S. For a RPT $\{p\}\ ;\ [r]$ we use the notation $\{p \mid r\}$.

For RPTs we introduce also syntactic notation similar to the one introduced for assert and demonic transformers:

$$\{x \rightsquigarrow y \mid P \mid R\} \quad = \quad \{x \mid P\}\ ;\ [x \rightsquigarrow y \mid R]$$

Next lemma introduces some results about assert and demonic update transformers and about relational property transformers.

Lemma 12. *If $p, q \in \Sigma^\omega \to \mathsf{Bool}$ and $r, r' \in \mathsf{Rel}$, then*

1. $\mathsf{Skip} = [x \rightsquigarrow x] = \{\mathsf{true}\}$
2. $\mathsf{Magic} = [\mathsf{false}]$, and $\mathsf{Fail} = \{\mathsf{false}\}$
3. $\{p\}\ ;\ \{p'\} = \{p \wedge p'\}$, and $[r]\ ;\ [r'] = [r \circ r']$
4. $\{p \mid r]\ ;\ \{p' \mid r'] = \{x \rightsquigarrow z \mid p.x \wedge (\forall y : r.x.y \rightarrow p'.y) \mid (r \circ r').x.z]$
5. $\{p \mid r] \sqcap \{p' \mid r'] = \{p \wedge p' \mid r \vee r']$
6. $\{p \mid r] = \{p \mid p \wedge r]$
7. $\{p \mid r] \sqsubseteq \{p' \mid r'] \Leftrightarrow (\forall x : p.x \Rightarrow p'.x) \wedge (\forall x, y : p.x \wedge r'.x.y \Rightarrow r.x.y)$
8. $\mathsf{grd}.\{p\} = \top$, and $\mathsf{grd}.[r] = \mathsf{in}.r$
9. $\mathsf{grd}.(\{p \mid r]) = \neg p \vee \mathsf{in}.r$
10. $\{p\}$ is strictly conjunctive, and $[r]$ is conjunctive
11. $[x \rightsquigarrow u,\ x \mid init.u]\ ;\ \{u,\ x \rightsquigarrow y \mid p.u.x \mid r.u.x.y] = (\mathsf{hide}\ u \in init : \{x \rightsquigarrow y \mid p.u.x \mid r.u.x.y])$

The facts 1. – 5. from this lemma show that Skip, Magic, Fail, $\{p\}$, and $[r]$ are relational property transformers, and that the relational property transformers are closed under sequential composition and demonic choice. The fact 6. allows using an assertion in a demonic choice. In $\{p \mid p \wedge r]$, p can be used to simplify the demonic choice ($p \wedge r$). The fact 7. shows how to refine two RPTs, and the facts 8. and 9. show how to compute the guard of RPTs. The fact 10. will be used together with Theorem 10 to combine the local variables of the composition of two RPTs. Finally, fact 11. shows that adding a local variable u ($[x \rightsquigarrow u,\ x \mid init.u]$) is equivalent to hiding parameters of property transformers. This last result will be used later to compose property transformers with state.

4.1 Guarded systems

Relational property transformers are a strict subclass of monotonic property transformers, but they still allow to describe systems that may behave *miraculously*. Often we are interested in systems that are guaranteed to never behave miraculously, i.e., in systems defined by transformers S such that $\mathsf{grd}.S = \top$. In these cases we use relational property transformers of the form $\{\mathsf{in}.r \mid r]$. We call these RPTs *guarded*:

Definition 13. The *guarded system* of a relation r is the relational property transformer $\{r] = \{\mathsf{in}.r \mid r]$.

For guarded systems we also introduce a notation similar to the notation introduced for relational property transformers:

$$\{x \rightsquigarrow y \mid R] = \{x \rightsquigarrow y \mid \mathsf{in}.R \mid R].$$

It is worth pointing out that the property transformer $\{\mathsf{in}.r \mid r]$ is as general as $\{p \wedge \mathsf{in}.r \mid r]$ because we have $\{p \wedge \mathsf{in}.r \mid r] = \{\mathsf{in}.(p \wedge r) \mid p \wedge r]$:

$\{p \wedge \mathsf{in}.r \mid r]$

$=$ {Theorem 12}

$\{p \wedge \mathsf{in}.r \mid p \wedge \mathsf{in}.r \wedge r]$

$=$ {Theorem 12}

$\{\mathsf{in}.(p \wedge r) \mid p \wedge r]$

The lemma that follows states several important closure properties for guarded systems.

Lemma 14. *If p is a property and r, r' are relations of appropriate types, then*

1. $\mathsf{grd}.\{r] = \top$
2. $\mathsf{Fail} = \{\bot]$ and $\mathsf{Skip} = \{x \rightsquigarrow x \mid \top]$
3. $\{p\} = \{x \rightsquigarrow x \mid p.x]$ and $\{p\}\ ;\ \{r] = \{p \wedge r]$
4. $\{r]\ ;\ \{r'] = \{x \rightsquigarrow z \mid \mathsf{in}.r.x \wedge (\forall y : r.x.y \Rightarrow \mathsf{in}.r'.y) \wedge (r \circ r').x.z]$
5. $\{r] \sqcap \{r'] = \{\mathsf{in}.r \wedge \mathsf{in}.r' \wedge (r \vee r')]$

Note that part 3 of the above lemma implies that assert transformers are special cases of guarded systems. However, a demonic update is generally not a guarded system. For instance, we have $\mathsf{grd}.[\bot] = \bot$. The following lemma states, demonic updates are guarded systems if and only if they impose no requirements on the inputs.

Lemma 15. *The demonic update transformer $[r]$ is a guarded system if and only if* $\mathsf{in}.r = \mathsf{true}$ *and in this case we have* $[r] = \{r]$.

Next we introduce some examples of guarded systems:

- $\mathsf{Havoc} = [x \rightsquigarrow y \mid \mathsf{true}]$
- $\mathsf{AssertLive} = \{x \mid \Box(\Diamond x)\}$
- $\mathsf{LiveHavoc} = \mathsf{AssertLive}\ ;\ \mathsf{Havoc}$
- $\mathsf{ReqResp} = [x \rightsquigarrow y \mid \Box(x \rightarrow \Diamond y)]$

The fact that these systems are guarded follows from Lemmas 14 and 15. Havoc is a reactive system that assigns an arbitrary value to y regardless of the value of the input sequence x. ReqResp is the request-response system *C* introduced in the introduction. Note that this system illustrates the ability of our framework to express unbounded nondeterminism since, for a given input sequence x, there is an infinite set of y sequences that satisfy the request-response LTL formula. (We can also express unbounded nondeterminism for systems with infinite data types.)

For the above systems we have the following properties

- $\mathsf{Havoc}\ ;\ \mathsf{AssertLive} = \mathsf{Fail}$
- $\mathsf{Havoc}\ ;\ \mathsf{LiveHavoc} = \mathsf{Fail}$
- $\mathsf{ReqResp}\ ;\ \mathsf{LiveHavoc} = \mathsf{LiveHavoc}$

The first two statements show that Havoc and AssertLive are incompatible and Havoc and LiveHavoc are also incompatible.

5. PROPERTY TRANSFORMERS BASED ON SYMBOLIC TRANSITION SYSTEMS

So far, we have introduced (relational and guarded) monotonic property transformers and showed how these can be defined using LTL. As a language, LTL is often more appropriate for system specification, and less appropriate for system implementation. For the latter purpose, it is often convenient to have a language which explicitly refers to state variables and allows to manipulate them, e.g., by defining the next state based on the current state and input. In this section we introduce a *symbolic transition system* notation which allows to do this, and show how this notation can be given semantics in terms of property transformers.

For example, suppose we want a counter which accepts as input infinite sequences of Boolean values, and returns infinite sequences of natural numbers where every output is the number of

true values seen so far in the input. Moreover, we want this counter to accept inputs where the number of true values is bounded by a given natural number n. If $\mathsf{count}.x.i$ is the number of trues in $x_0,\ x_1,\ \ldots,\ x_i$, then this system can be defined in the following way:

$$\mathsf{bcounter}.n = \{x \mid \forall i : \mathsf{count}.x.i \leq n\}\ ;\ [x \rightsquigarrow y \mid y = \mathsf{count}.x]$$

Although this system is defined globally, when computing y_i we only need to know x_i, and we need to know how many true values we have seen so far in the input. We can store the number of true values seen so far in a *state variable* u. Then, it would be natural to define the counter *locally*, that is, define *one step* of the counter, as follows:

$$\{u_i \leq n\}\ ;\ [u_i, x_i \rightsquigarrow u_{i+1}, y_i \mid \\ u_{i+1} = (\mathsf{if}\ x_i\ \mathsf{then}\ u_i + 1\ \mathsf{else}\ u_i) \wedge y_i = u_{i+1}]$$

where i is the index of the step, and we assume that initially $u_0 = 0$. In the above definition, u_i refers to the *current* state (i.e., the state at current step i) and u_{i+1} refers to the *next* state (i.e., the state at next step $i+1$), while x_i refers to the current input and y_i refers to the current output (both at current step i). At every step the assert statement $\{u_i \leq n\}$ tests if u_i is less or equal to n. If this is false then the system fails because the input requirement that the number of true values never exceeds n is violated. If $u_i \leq n$ then we calculate the next state value u_{i+1} and the output value y_i.

Generalizing from this example, a *symbolic transition system* is a tuple $(init, p, r)$, formed by a predicate $init.u$, a predicate $p.u.x$, and a relation $r.u.u'.x.y$, where x is the input, u is the current state, u' is the next state, and y is the output. The predicate $init$ is called the *local initialization predicate* of the system, the predicate p is called the *local precondition* of the system, and r is called the *local input-output relation* of the system. The intuitive interpretation of such a system is that we start with some initial state $u_0 \in init$ and we are given some input sequence $x_0,\ x_1,\ \ldots$, and if $p.u_0.x_0$ is true, then we compute the next state u_1 and the output y_0 such that $r.u_0.u_1.x_0.y_0$ is true. Next, if $p.x_1.u_1$is true, then we compute u_2 and y_1 such that $r.u_1.u_2.x_1.y_1$ is true, and so on. If at any step $p.u_i.x_i$ is false, then the computation fails, and the input $x_0,\ x_1,\ \ldots$ is not accepted.

Note that the computation defined by the relation r can be nondeterministic in both next state u' and output y. That is, for given values x and u for the input and current state, there could be multiple values for the next state u' and output y such that $r.u.u'.x.y$ is true. We must carefully account for this non-determinism when defining the property transformer based on such a symbolic transition system. To see the complications that may arise, consider another example:

$$init.u = u \ \text{ and } \ p.u.x = u \ \text{ and } \ p.u.u'.x.y = (x = y)$$

In this system, if the current state is true then we choose arbitrarily a new state u' and we copy the input x into the output y. If the system chooses $u' = \mathsf{false}$ then in the next step the system will fail, regardless of the input. This example shows that in a nondeterministic system, for the same input there could be different choices of internal states such that in one case the system succeeds while in another it fails. In the example above the choice of state sequence $(\forall i : u_i = \mathsf{true})$ results in a successful computation, but all other choices of state sequences fail. In our definition of property transformers, we accept an input only if *all* choices of internal states lead to no failures.

Formally, we say that an input sequence $x_0,\ x_1, \ldots$ is *illegal* for a symbolic transition system if there is some $k \in \mathsf{Nat}$ and some choice $u_0,\ u_1, \ldots$ of states and $y_0,\ y_1,\ \ldots$ of outputs such that $init.u_0$ and $(\forall i < k : r.u_i.u_{i+1}.x_i.y_i)$ and $\neg p.u_k.x_k$. For technical reasons, we need to generalize p to be a predicate not only on the current state and input, but also on the next state (the need for this will become clear in the sequel, see Theorem 18 and discussion that follows). With this generalization, we define the $\mathsf{illegal}$ predicate on symbolic transition systems and input sequences, as follows:

$$\mathsf{illegal}.init.p.r.x = (\exists u, y, k : init.u_0 \\ \wedge (\forall i < k : r.u_i.u_{i+1}.x_i.y_y) \wedge \neg p.u_k.u_{k+1}.x_k)$$

We can also formalize a *run* of a symbolic transition system, using the predicate run. For sequences x, u, and y, the predicate $\mathsf{run}.r.u.x.y$ is defined by:

$$\mathsf{run}.r.u.x.y = (\forall i : r.u_i.u_{i+1}.x_i.y_i) = \Box\, r.u.u^1.x.y$$

where, recall, u^1 denotes the sequence $u_1, u_2, \cdots$, i.e., the sequence of states starting from the second state u_1 instead of the initial state u_0. If the predicate $\mathsf{run}.r.u.x.y$ is true we say that there is a *run* of the system with the inputs x, the outputs y and the states u. We can now define monotonic property transformers based on symbolic transition systems as follows:

Definition 16. Consider a symbolic transition system described by $(init, p, r)$. Such a system defines a monotonic property transformer called a *local property transformer,* and denoted $\{\!|\ init \mid p \mid r\ |\!\}$, as follows:

$$\{\!|\ init \mid p \mid r\ |\!\}.q.x = \neg\mathsf{illegal}.init.p.r.x \\ \wedge (\forall u, y : init.u \wedge \mathsf{run}.r.u.x.y \Rightarrow q.y).$$

What the above definition states is that an input sequence x is in the set of input sequences of $\{\!|\ init \mid p \mid r\ |\!\}$ that are guaranteed to establish q iff: (1) x is legal; and (2) for all choices of state traces u and output traces y, if u_0 satisfies $init$, and if there is a run of the system with the inputs x, the outputs y and the states u, then y must be in q.

The *local transition* from state u to u' of a local system is

$$\mathsf{localtran}.p.r.u.u' = \{x \mid p.u.u'.x\}\ ;\ [x \rightsquigarrow y \mid r.u.u'.x.y]$$

If $Init$, and P, and R are Boolean expression possibly containing free the variables u and u, u', x and u, u', x, y, respectively then

$$\{\!|\ x \rightsquigarrow u \rightsquigarrow y \mid Init \mid P \mid R\ |\!\} \\ = \{\!|\ \lambda u : Init \mid \lambda u, u', x : P \mid \lambda u, u', x, y : R\ |\!\}$$

For the counter example we have:

$$\mathsf{bcounter}.n = \{\!|x \rightsquigarrow u \rightsquigarrow y \mid u = 0 \mid \\ u \leq n \mid u' = (\mathsf{if}\ x\ \mathsf{then}\ u + 1\ \mathsf{else}\ u) \wedge y = u'\!|\!\}$$

and for the wrong choice example we have

$$\mathsf{Fail} = \{\!|\ x \rightsquigarrow u \rightsquigarrow y \mid u \mid u \mid y = x\ |\!\}$$

that is, for all input sequences this system fails.

Lemma 17. *The set of input sequences from which the system* $\{\!|\ init \mid p \mid r\ |\!\}$ *fails is equal to the set of its illegal input sequences:*

$$\mathsf{fail}.\{\!|\ init \mid p \mid r\ |\!\} = \mathsf{illegal}.init.p.r$$

The definition of a local property transformer is close to our intuition of what this system should be, however it is difficult to use this definition if we want to prove refinement of (local) property transformers, or if we want to compose two (local) property transformers. For example if we want to prove a refinement like

$$\{p \mid r\} \sqsubseteq \{\!|\ init \mid p' \mid r'\ |\!\}$$

then we need to expand the definition of $\{\!|\, init \mid p' \mid r' \,|\!\}$ and reason about individual values of traces (x_i, y_i, u_i). This reasoning is at a lower level than, for example, the reasoning about the refinement

$$\{p\} \,;\, [r] \sqsubseteq \{p''\} \,;\, [r'']$$

which, by Lemma 12, is equivalent to

$$(\forall x : p.x \Rightarrow p''.x) \wedge (\forall x, y : p.x \wedge r''.x.y \Rightarrow r.x.y)$$

In this property x, y may also stand for traces, but this formula does not contain references to specific values (x_i or y_i) of these traces.

Next theorem shows that a local property transformer $\{\!|\, init \mid p \mid r \,|\!\}$ belong to the class of relational property transformers.

Theorem 18. *For $init$, p, and r as in the definition of a local property transformer we have:*

$$\begin{array}{l}\{\!|\, init \mid p \mid r \,|\!\} = [x \rightsquigarrow u,\ x \mid init.u] \,; \\ \quad \{u,\ x \mid (\mathsf{in}.r \mathbin{\mathsf{L}} p).u.u^1.x\} \,;\, [u,\ x \rightsquigarrow y \mid \Box\, r.u.u^1.x.y] \\ = \{x \mid \forall u : init.u_0 \Rightarrow (\mathsf{in}.r \mathbin{\mathsf{L}} p).u.u^1.x\} \,; \\ \quad [x \rightsquigarrow y \mid \exists u : init.u_0 \wedge \Box\, r.u.u^1.x.y]\end{array}$$

Theorem 18 gives a representation of local property transformers where we do not have anymore quantification over the domain of the traces. This theorem also justifies our earlier generalization of the local precondition to be a function not only on the current state and the input, but also on the next state. This is so because the precondition of the representation of a local property transformer depends anyway on the next state $(\mathsf{in}.r.u.u^1.x.y)$.

We call the precondition $(\mathsf{in}.r \mathbin{\mathsf{L}} p).u.u^1.x$ from the representation of the local property transformer $\{\!|\, init \mid p \mid r \,|\!\}$ the *global precondition* of $\{\!|\, init \mid p \mid r \,|\!\}$. Similarly $\Box\, r.u.u^1.x.y$ is the *global input-output relation* of $\{\!|\, init \mid p \mid r \,|\!\}$.

The refinement of a general relational property transformer into a local property transformer is given by the next lemma.

Lemma 19. *For $init$, p, p', r, r' of appropriate types we have*

$$\begin{array}{l}\{x \rightsquigarrow y \mid p \mid r\} \sqsubseteq \{\!|\, init \mid p' \mid r' \,|\!\} \Leftrightarrow \\ \quad (\forall u \in init : (\forall x : p.x \Rightarrow (\mathsf{in}.r' \mathbin{\mathsf{L}} p').u.u^1.x) \\ \quad\ \wedge (\forall x,\ y : p.x \wedge \Box\, r'.u.u^1.x.y \Rightarrow r.x.y))\end{array}$$

5.1 Sequential composition of local transformers

Lemma 12 and Theorem 10 allow us to calculate the result of the sequential composition of two local systems. This composition is given by the next theorem.

Theorem 20.

$$\begin{array}{l}\{\!|\, init \mid p \mid r \,|\!\} \,;\, \{\!|\, init \mid p' \mid r' \,|\!\} = \\ [x \rightsquigarrow u,\ v,\ x \mid init.u \wedge init'.v] \,; \\ \quad \{u,\ v,\ x \mid (\mathsf{in}.r \mathbin{\mathsf{L}} p).u.u^1.x \\ \quad\ \wedge (\forall y : \Box\, r.u.u^1.x.y \Rightarrow (\mathsf{in}.r' \mathbin{\mathsf{L}} p').v.v^1.y)\} \,; \\ \quad [u,\ v,\ x \rightsquigarrow z \mid (\Box\, (r \circ\!\circ r').(u,v).(u^1,v^1).x.z]\end{array}$$

where $(r \circ\!\circ r').(u,v).(u',v') = (r.u.u' \circ r'.v.v')$

Ideally the composition of two local systems $S = \{\!|\, init \mid p \mid r \,|\!\}$ and $S' = \{\!|\, init' \mid p' \mid r' \,|\!\}$ would be a local system corresponding to the composition of the local transitions of S and S'. Unfortunately this is not the case. We show what would be the local system for the composition of the local transitions of S and S'. We have

$$\begin{array}{rcl}\mathsf{localtran}.p.r.u.u' & = & \{x \mid p.u.u'.x\} \,;\, [x \rightsquigarrow y \mid r.u.u'.x.y] \\ \mathsf{localtran}.p'.r'.v.v' & = & \{x \mid p'.v.v'.x\} \,;\, [x \rightsquigarrow y \mid r'.v.v'.x.y]\end{array}$$

and

$$\begin{array}{rl} & \mathsf{localtran}.p.r.u.u' \,;\, \mathsf{localtran}.p'.r'.v.v' \\ = & \{\text{Lemma 12}\} \\ & \{x \mid p.u.u'.x \wedge (\forall y : r.u.u'.x.y \Rightarrow p'.v.v'.y)\} \,; \\ & \quad [(r.u.u') \circ (r'.v.v')]\end{array}$$

The sequential composition of the two systems has as state pairs (u, v) and the initialization predicate, the local precondition, and the local relation of the composition should be given by

$$\begin{array}{rcl}init''.(u,v) & = & init.u \wedge init'.v \\ p''.(u,v).(u',v').x & = & p.u.u'.x \wedge (\forall y : r.u.u'.x.y \Rightarrow p'.v.v'.y) \\ r'' & = & r \circ\!\circ r'\end{array}$$

The local system of the composition of the local transitions of S and S' is

$$\begin{array}{l}\{\!|\, init'' \mid p'' \mid r'' \,|\!\} = \\ \quad [x \rightsquigarrow u,\ v,\ x \mid init''.(u,v)] \,; \\ \quad\ \{u,\ v,\ x \mid (\mathsf{in}.r'' \mathbin{\mathsf{L}} p'').(u,v).(u^1,v^1).x\} \,; \\ \quad\ [u,\ v,\ x \rightsquigarrow z \mid \Box\, r''.(u,v).(u^1,v^1).x.z]\end{array}$$

Unfortunately the equality $S \,;\, S' = \{\!|\, init'' \mid p'' \mid r'' \,|\!\}$ is not true in general for arbitrary local systems S and S'. We have

$$init''.(u,v) = init.u \wedge init'.v \text{ and } r'' = r \circ\!\circ r'$$

but there exists $p,\ r,\ p',\ r',\ u,\ v$, and x such that

$$\begin{array}{l}(\mathsf{in}.r'' \mathbin{\mathsf{L}} p'').(u,v).(u^1,v^1).x \neq (\mathsf{in}.r \mathbin{\mathsf{L}} p).u.u^1.x \\ \quad \wedge (\forall y : \Box\, r.u.u^1.x.y \Rightarrow (\mathsf{in}.r' \mathbin{\mathsf{L}} p').v.v^1.y)\end{array} \qquad (1)$$

If we take

$$\begin{array}{rcl}p.u.u'.x & = & \mathsf{true} \\ r.u.u'.x.y & = & (u = 0 \wedge u' = 1) \\ p'.v.v'.y & = & \mathsf{false} \\ u.i & = & i\end{array}$$

then (1) becomes true.

Intuitively $S \,;\, S' \neq \{\!|\, init'' \mid p'' \mid r'' \,|\!\}$ because when executing the system $S \,;\, S'$, the precondition p' of S' is tested after a complete execution of S, however in our example above, the execution of S proceeds normally with the first step when started in the state $u = 0$, but then next step is miraculous because $r.1.u'.x.y$ is false. Therefore the assertion of S' containing p' is not reached. On the other hand the execution of $\{\!|\, init'' \mid p'' \mid r'' \,|\!\}$ starting from the same initial state $u = 0$ proceeds normally with the first step of S ($\{x \mid p.u.u'.x\} \,;\, [x \rightsquigarrow y \mid r.u.u'.x.y]$), and then tests p', and it fails because p' is false.

5.2 Guarded local systems

As we have seen from the previous section, the composition of two local transformers is not necessarily a local transformers. This is because of the possible miraculous behavior of the systems. In this section we restrict the local precondition of a local system such that we do not have miraculous behavior anymore. We achieve this by considering systems where the local precondition p is $\mathsf{in}.r$.

Definition 21. The *guarded local system* of $init$ and r is denoted by $\{\!|\, init \mid r \,|\!\}$ and it is given by

$$\{\!|\, init \mid r \,|\!\} = \{\!|\, init \mid \mathsf{in}.r \mid r \,|\!\}$$

and the *local precondition* of a local guarded reactive systems with state is $\mathsf{in}.r$.

Theorem 22. *For $init$, and r as in the definition of a local guarded system we have:*

$$\{\!|\, init \mid r \,|\!\} = [x \rightsquigarrow u,\ x \mid init.u] \,;\, \{u,\ x \rightsquigarrow y \mid \Box\, r.u.u^1.x.y]$$

The next theorem shows that the sequential composition of two guarded local systems is also a guarded local system.

Theorem 23. *For $init$, $init'$, r, and r' we have*

$$\{\!|\, init \mid r \,|\!\} \,;\, \{\!|\, init' \mid r' \,|\!\} = \{\!|\, init'' \mid rel\_comp.r.r' \,|\!\}$$

where $init''.(u, v) = init.u \wedge init'.v$ *and*

$$\begin{array}{l} rel\_comp.r.r'.(u,v).(u',v').x.z = \\ \quad (\mathsf{in}.r.u.u'.x \wedge (\forall y : r.u.u'.x.y \;\Rightarrow \\ \quad\quad \mathsf{in}.r'.v.v'.y) \wedge ((r.u.u') \circ (r'.v.v')).x.z) \end{array}$$

5.3 Stateless systems

We define *stateless systems* as a special case of the local systems, where the state u ranges over a singleton set $\{\bullet\}$ and where $init.u = \mathsf{true}$. In this case we have

Theorem 24. *For p, and r as in the definition of a stateless local reactive system we have:*

$$\begin{array}{rcl} \{\!|\, init \mid p \mid r \,|\!\} & = & \{x \rightsquigarrow y \mid (\mathsf{in}.r \mathsf{\ L\ } p).x \mid (\Box\, r).x.y\} \\ & = & \{\mathsf{in}.r \mathsf{\ L\ } p \mid \Box\, r\} \end{array}$$

Based on this theorem we use the notation $\{\mathsf{in}.r \mathsf{\ L\ } p \mid \Box\, r\}$ for a stateless local reactive system.

The next theorem gives a procedure to calculate the sequential composition of two stateless systems.

Theorem 25.

$$\begin{array}{l} \{\mathsf{in}.r \mathsf{\ L\ } p \mid \Box\, r\} \,;\, \{\mathsf{in}.r' \mathsf{\ L\ } p' \mid \Box\, r'\} = \\ \quad \{x \mid (\mathsf{in}.r \mathsf{\ L\ } p).x \wedge (\forall y : \Box\, r.x.y \Rightarrow (\mathsf{in}.r' \mathsf{\ L\ } p').y)\} \,; \\ \quad\quad [\Box\, (r \circ r')] \end{array}$$

As in the case of general local systems with state, the composition of two stateless systems is not a stateless system. This motivates us to introduce guarded stateless systems, similarly to guarded local systems.

Definition 26. A *guarded stateless system* is a stateless system where $p = \mathsf{in}.r$.

Theorem 27. *For any r:* $\{\mathsf{in}.r \mathsf{\ L\ } \mathsf{in}.r \mid \Box\, r\} = \{\Box\, r\}$.

This is because we have

$$\{\mathsf{in}.r \mathsf{\ L\ } \mathsf{in}.r \mid \Box\, r\} = \{\Box\, \mathsf{in}.r \mid \Box\, r\} = \{\mathsf{in}.(\Box\, r) \mid \Box\, r\} = \{\Box\, r\}$$

We use the notation $\{\Box\, r\}$ for a stateless local guarded reactive system.

Sequential composition of guarded stateless systems is also a guarded stateless system:

Theorem 28. *For r, and r' we have*

$$\{\Box\, r\} \,;\, \{\Box\, r'\} = \{\Box\, (rel\_comp.r.r')\}$$

where rel_comp is as defined before, but without the state parameters $u,\ u',\ v$, and v'.

Next we introduce some special properties and we show that stateless guarded local systems behave consistently with respect to these properties.

Definition 29. For a property q, the i-th *projection* of q is a predicate on states given by $\mathsf{proj}.q.i.s = (\exists \sigma : q.\sigma \wedge \sigma_i = s)$ and a property q is a *piecewise local property* if it satisfies the condition$(\forall \sigma : (\forall i : \mathsf{proj}.q.i.\sigma_i) \Rightarrow q.\sigma)$.

Equivalently, a property q is piecewise local if there exist some predicates $p_0, p_1, \ldots$ such that $(\forall \sigma : \sigma \in q \Leftrightarrow (\forall i : p_i.\sigma_i))$.

There are properties which are not local. For example the liveness property $q = (\Box\, (\Diamond\, x))$ is not local because $\sigma = (\lambda i : \mathsf{false})$ satisfies the condition $(\forall i : \mathsf{proj}.q.i.(\sigma.i))$, but $\sigma \notin q$.

Lemma 30. *If r is a state relation, then*

1. $\{\Box\, r\}.q.x \Rightarrow (\forall i : \{r\}.(\mathsf{proj}.q.i).x_i)$
2. If q is piecewise local then
   $(\forall i : \{r\}.(\mathsf{proj}.q.i).x_i) \Rightarrow \{\Box\, r\}.q.x$

This lemma asserts that for the piecewise local property q and input sequence x, all possible outputs of $\{\Box\, r\}$ starting from x are in q if and only if for all steps i all possible outputs of $\{r\}$ from x_i are in $\mathsf{proj}.q.i$. So the global execution of $\{\Box\, r\}$ is equivalent to the execution of $\{r\}$ on all steps.

If we have a local system it does not necessarily mean that we cannot study its behavior with respect to non piecewise local properties. For example let us consider a stateless local guarded system that at each step computes $y = x$ or $y = x + 1$, assuming that $x > 0$:

$$S = \{x \rightsquigarrow y \mid \Box\, (x > 0 \wedge (y = x \vee y = x + 1))\}$$

If we want to see under what conditions on the input x the output of S satisfies the property $q = \Box \Diamond y < 10$, then we should calculate $S.q$:

$$S.q = \Box\, (x > 0 \wedge \Diamond\, x < 9)$$

If we want to see under what conditions on the input the output of S satisfies $q' = \Box \Diamond y = 10$, then we should calculate $S.q'$. In this case we have $S.q' = \bot$. This is so because of the demonic choice $y = x$ or $y = x + 1$. For all values of x it is always possible to choose $y \neq 10$.

6. APPLICATION: EXTENDING RELATIONAL INTERFACES WITH LIVENESS

To illustrate the power of our framework, we show how it can handle as a special case the extension of the relational interface theory presented in [17] to infinite behaviors and liveness. We note that the theory proposed in [17] allows to describe only safety properties, in fact, finite and prefix-closed behaviors. Extending to infinite behaviors and liveness properties is mentioned as an open problem in [17].

A number of examples showcasing this extension have already been provided in the introduction. Here we provide an additional example:

$$\begin{array}{l} init.u = (u = 0) \\ p.u.u'.x = (-1 \leq u \leq 3) \\ r.u.u'.x.y = ((x \wedge u' = u + 1) \vee (\neg x \wedge u' = u - 1) \\ \quad \vee u' = 0) \wedge (y = (u' = 0)) \end{array}$$

This symbolic system accepts a Boolean input x and a Boolean output y. If the input is true then state counter u is incremented. If the input is false then u is decremented. Regardless of the input, the system may also choose nondeterministically to set the counter to zero. The output of the system is true whenever the counter reaches zero. The system also restricts the value of the state to be between -1 and 3. If the state goes out of this range the system will fail. The system is supposed to start from state $u = 0$. The local system for this relation is

$$\begin{array}{l} \{\!|\, init \mid p \mid r \,|\!\} = \\ \quad [x \rightsquigarrow u\ x \mid u = 0] \,;\, \{u,\ x; \mid (\mathsf{in}.r \mathsf{\ L\ } p).u.u^1.x\} \,; \\ \quad\quad [u,\ x \rightsquigarrow y \mid \Box\, r.u.u^1.x.y] \end{array} \qquad (2)$$

However we are interested in a system which is also capable of ensuring the liveness property that y is true infinitely often. We achieve this by adding the constraint $\Box \Diamond y$ to the input-output relation of 2. So the full example is

$$\begin{array}{l}\mathsf{EXAMPLE} = \\ \quad [x \rightsquigarrow u\ x \mid u = 0]\ ;\ \{u,\ x \mid (\mathsf{in}.r \mathsf{L}\ p).u.u^1.x\}\ ; \\ \quad\quad [u,\ x \rightsquigarrow y \mid (\Box\, r.u.u^1.x.y) \wedge (\Box \Diamond y)]\end{array} \tag{3}$$

In this example the condition $-1 \leq u \leq 3$ is a safety property, and we designed the example such that this property is enforced on the input. That is, some input is accepted by this system only if this property is not violated. For example the input $x_0 = \mathsf{true}$, $x_1 = \mathsf{false}$, ... maintains this property. On the other hand the property $\Box \Diamond y$ is a liveness property which is guaranteed by the system, regardless of the input. If some input violates this property, then that input is rejected. If we need we can move this property to the precondition (adapted to the state variable) and then the system will fail if the input is such that this property is false. We can prove that our example system establishes the liveness property $\Box \Diamond y$ for all inputs that do not fail, i.e. they satisfy

$$prec\_g.x = (\forall u : u_0 = 0 \Rightarrow (\mathsf{in}.r \mathsf{L}\ p).u.u^1.x)$$

We have

$$\forall x : \mathsf{EXAMPLE}.(\{y \mid \Box \Diamond y\}).x = prec\_g.x \tag{4}$$

We can now use the system EXAMPLE as specification and we can, for instance, refine it to the system which always assigns true to the output variable:

$$\mathsf{EXAMPLE} \sqsubseteq [x \rightsquigarrow y \mid \Box\, y].$$

We can also assume that the input satisfies some additional property. For example we can assume that x is alternating between true and false:

$$\{x \mid \Box\, (x = \neg \bigcirc x)\}\ ;\ \mathsf{EXAMPLE}$$

Then we can show that this new system is refined by the original symbolic transition system:

$$\begin{array}{l}\{x \mid \Box\, (x = \neg \bigcirc x)\}\ ;\ \mathsf{EXAMPLE} \\ \quad \sqsubseteq \{x \mid \Box\, (x \Leftrightarrow \neg \bigcirc x)\}\ ;\ \{\!|\, init \mid p \mid r \,|\!\} \\ \quad \sqsubseteq \{\!|\, init \mid p \mid r \,|\!\}\end{array}$$

because the additional property used as precondition ensures the liveness property.

Using this formalism we can construct liveness specifications as the example system, and we can refine them in appropriate contexts to systems which do not have any liveness property, but they preserve the liveness property of the input.

From 4 we also obtain

$$\mathsf{EXAMPLE} = \mathsf{EXAMPLE}\ ;\ \{y \mid \Box \Diamond y\}$$

We can use this property when constructing another system that uses the output from EXAMPLE as input. Then we know that this input satisfies the liveness property $\Box \Diamond y$ and we can design this second system accordingly.

7. CONCLUSIONS

In this paper we introduced a property transformer semantics for reactive systems which supports refinement, composition, compatibility, demonic choice, unbounded nondeterminism, among other interesting system properties. The semantics also supports angelic choice: we haven't specifically exploited this here and leave it for future work. The semantics is compositional, and can be used to specify and reason about both safety and liveness properties. Our work generalizes previous work on relational interfaces to systems with infinite behavior and liveness properties. Future work includes extending the framework to continuous-time and hybrid systems.

8. REFERENCES

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

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

[3] R.-J. Back. *On the correctness of refinement in program development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978.

[4] R.-J. Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, pages 67–93. Springer, 1990.

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

[6] R.-J. Back and J. Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *CONCUR '94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 367–384. Springer Berlin Heidelberg, 1994.

[7] R.-J. Back and Q. Xu. Refinement of fair action systems. *Acta Informatica*, 35(2):131–165, 1998.

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

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

[10] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. MIT Press, Cambridge, MA, USA, 1989.

[11] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. 1985.

[12] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

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

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

[15] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Oct 1977.

[16] V. Preoteasa. Refinement algebra with dual operator. *Science of Computer Programming*, 92, Part B(0):179 – 210, 2014.

[17] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. A theory of synchronous relational interfaces. *ACM Trans. Program. Lang. Syst.*, 33(4):14:1–14:41, July 2011.

[18] D. Yadav and M. Butler. Verification of liveness properties in distributed systems. In *Contemporary Computing*, volume 40 of *Communications in Computer and Information Science*, pages 625–636. Springer Berlin Heidelberg, 2009.