

Refinement Calculus of Reactive Systems

Viorel Preoteasa¹ and Stavros Tripakis^{1,2}

October 1, 2018

¹Aalto University, Finland.

²University 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 traces into sets of input traces. We show how to model in this semantics refinement, sequential composition, demonic choice, and other semantic operations on reactive systems. We use primarily higher order logic to express our results, but we also show how property transformers can be defined using other formalisms more amenable to automation, such as linear temporal logic (suitable for specifications) and symbolic transition systems (suitable for implementations). Finally, we show how this framework generalizes previous work on relational interfaces so as to be able to express systems with infinite behaviors and liveness properties.

1 Introduction

Refinement calculus [2, 4] 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: $\text{Div}(q) = \{(x, y, z) \mid y \neq 0 \wedge (x, y, x/y) \in q\}$.

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: $(x, y) := \text{read}()$. 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: $\text{Havoc}(q) = \begin{cases} \top & \text{if } q = \top \\ \perp & \text{otherwise} \end{cases}$ where \top and

\perp denote the universal and empty sets, respectively. Now, what happens if we compose the two statements in sequence? That is, $(x, y) := \text{read}(); z := x/y$. Refinement calculus tells us that sequential composition of statements corresponds to function composition of their predicate transformers, so the semantics of the

composition is $\text{Havoc} \circ \text{Div}$, which can be shown to be equivalent to the predicate transformer Fail , defined as $\text{Fail}(q) = \perp$ 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 $\text{abs}(x/y - z') \leq \epsilon$. We model this new division statement as a new predicate transformer Div' defined as follows: $\text{Div}'(q) = \{(x, y, z) \mid y \neq 0 \wedge (\forall z' : \text{abs}(x/y - z') \leq \epsilon \Rightarrow (x, y, z) \in q)\}$. 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* [10]. 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.

Example 1. Consider the two components shown in Figure 1 and specified using the syntax of linear temporal logic [14]. 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 fact that the output of B has the same label x as the input means that B sets its output to be equal to the input – provided the input requirement holds). 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 .

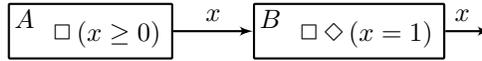


Figure 1: Two incompatible systems

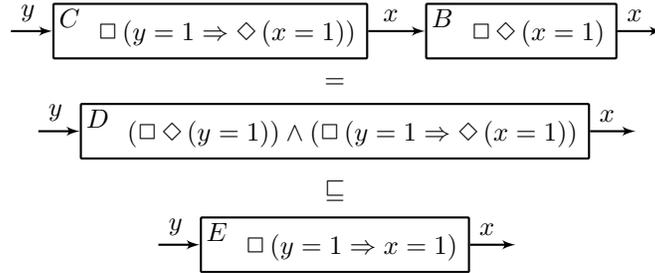


Figure 2: Two compatible systems (top), their composition (middle), and a refinement (bottom)

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 [18] 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 [13] to express our concepts. All results presented in this paper were formalized in Isabelle, and our presentation translates directly into Isabelle’s formal language. The Isabelle formalization is available from the Archive of Formal Proofs [15].

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 [7] specifications are relations on input-output streams. Focus is able to express infinite streams and liveness properties, however, it focuses on *input-receptive* systems, that is, systems where all input values are always legal. Other compositional frameworks that also assume input-receptiveness are Dill’s *trace theory* [9], *IO automata* [12], and *reactive modules* [1]. Our framework allows to specify non-input-receptive systems, where some inputs are sometimes illegal. For example, in the case of the division statement $[z := x/y]$, we can write $y \neq 0 \wedge \dots$ instead of $y \neq 0 \Rightarrow \dots$. The conjunction specifies a non-input-receptive system where $y = 0$ is an illegal input, whereas the implication specifies an input-receptive system. As argued in [18], the ability to specify illegal inputs is essential in order to obtain a lightweight verification framework, akin to type-checking. In particular, it allows to define a behavioral notion of component *compatibility*, 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* [8] and *relational interfaces* [18]. 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* [3, 5], which do not have acceptance conditions (say, of type Bï¿œchi) and therefore cannot express general liveness properties. *Fair* action systems [6], 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.

2 Preliminaries

We use capital letters X, Y, Σ, \dots to denote types, and small letters to denote elements of these types $x \in X, \dots$. 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 $X \rightarrow Y$ 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 $f : X \rightarrow Y \rightarrow Z$ 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 the result applied to y . According to this interpretation function application associates to the left ($f.x.y = (f.x).y$) and correspondingly the function type constructor (\rightarrow) associates to the right ($X \rightarrow Y \rightarrow Z = X \rightarrow (Y \rightarrow Z)$). We use also

lambda notation for constructing functions. For example if $x + y + 2 \in \mathbf{Nat}$ is a natural expression then $(\lambda x, y : x + y + 2) : \mathbf{Nat} \rightarrow \mathbf{Nat}$ 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 returning Boolean values (e.g., $p : X \rightarrow Y \rightarrow \mathbf{Bool}$), and relations are predicates with at least two arguments. For a relation $r : X \rightarrow Y \rightarrow \mathbf{Bool}$ we denote by $\text{in}.r : X \rightarrow \mathbf{Bool}$ the predicate given by

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

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

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

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

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

and we also have in this case:

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

We use \perp and \top as the smallest and greatest predicates

$$\perp.x = \mathbf{false} \text{ and } \top.x = \mathbf{true}$$

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 (e.g., $\lambda x, y : x \leq 10 \Rightarrow y > 10$), 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, \dots , a state $s \in \Sigma$ gives values to the program variables x, y, z, \dots . 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 $\sigma \in \Sigma^\omega$ where $\Sigma^\omega = (\mathbf{Nat} \rightarrow \Sigma)$. For $\sigma \in \Sigma^\omega$, $\sigma_i = \sigma.i$, and $\sigma^i \in \Sigma^\omega$ is given by $\sigma_j^i = \sigma^i.j = \sigma_{i+j}$. We consider a pair of traces (σ, σ') as being the same as a trace of pairs $(\lambda i : (\sigma_i, \sigma'_i))$.

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

2.1 Linear temporal logic

Linear temporal logic (LTL) [14] 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 $i \in \mathbf{Nat}$ $x.\sigma_i = 1$.

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 $p, q \in \Sigma^\omega \rightarrow \text{Bool}$ 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 $p \text{ U } q$ 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 $p \text{ U } q$ is true in σ if there is some time in σ when q is true, and until then p is true. Formally we have:

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

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

$$\begin{aligned} x + y &= x_0 + y_0 \\ x \wedge y &= x_0 \wedge y_0 \end{aligned}$$

Lemma 2. *If p and q are properties, then we have: $(\exists x : \Box p) = \Box(\exists x : p)$ and $\Box(\text{in}.p) = \text{in}.\Box p$.*

Definition 3. We define the operator $p \text{ L } q = \neg(p \text{ U } \neg q)$. Intuitively, $p \text{ L } q$ holds if, whenever p holds continuously up to step $n - 1$, then q must hold at step n .

Lemma 4. *If p and q are properties, then we have*

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

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 $(\Sigma_2 \rightarrow \text{Bool}) \rightarrow (\Sigma_1 \rightarrow \text{Bool})$, with a weakest precondition interpretation. If S is a program and $q \in \Sigma_2 \rightarrow \text{Bool}$ 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 $(\Sigma_i \rightarrow \text{Bool})$ 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 $(\Sigma_y^\omega \rightarrow \text{Bool})$ to the complete Boolean algebra of Σ_x properties $(\Sigma_x^\omega \rightarrow \text{Bool})$, 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 , $\sigma = x_0, x_1, \dots$, and produces a sequence of values for y , $\sigma' = y_0, y_1, \dots$. This execution may be nondeterministic, that is, for the same input sequence σ we can obtain different output sequences 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

$q \in \Sigma_y^\omega \rightarrow \text{Bool}$, 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 .

S must be monotonic in the following sense: interpreting properties as sets, S is *monotonic* if for any two properties q, q' , if $q \subseteq q'$ then $S.q \subseteq S.q'$.

3.1 Property transformers based on LTL

Monotonic property transformers are appropriate primarily as semantic descriptions of systems. In practice, we also need some *syntax* for describing systems in general, and property transformers in particular. In this paper, we use two types of syntax: LTL, and *symbolic transition systems*. Property transformers based on symbolic transition systems will be discussed in detail in Section 5. Property transformers based on LTL are a special case of *relational property transformers*, discussed in detail in Section 4. Here we provide an illustrative example.

Example 5. Consider again component B from Example 1, Figure 1. Suppose variable x is a Boolean, taking values in the set $\{0, 1\}$, i.e., $\Sigma_x = \{0, 1\}$. Then, B can be modeled as a property transformer which from the set of properties ($\{0, 1\}^\omega \rightarrow \text{Bool}$) to the same set (because B copies its input to its output, provided the requirements on the input are satisfied). Let $q \subseteq \{0, 1\}^\omega$. Then $B.q$ must contain exactly those infinite input sequences $\sigma \in \{0, 1\}^\omega$ such that: (1) σ satisfies the input requirement expressed by the LTL property $\Box \Diamond (x = 1)$, i.e., σ must contain infinitely many 1's; and (2) σ is in q , since B copies its input to its output. Written formally, $B.q = \{\sigma \in q \mid (\Box \Diamond x = 1).\sigma\}$. Clearly, B is a monotonic property transformer, as the larger the set q is, the larger $B.q$ is.

3.2 Using property transformers as implicit system specifications

Example 6. Example 5 provided the explicit definition of the property transformer for a certain system, thereby also essentially completely defining that system. Using property transformers, we can also specify systems *implicitly*, by imposing constraints that the property transformers of these systems must satisfy. In this way, we can specify the fact that a certain system must exhibit various properties that we are interested in. For example, the specification of a system S that guaranties 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 (interpreted as a property transformer) must satisfy. Below, in Section 3.3 we give a complete definition of a MPT which satisfies the requirement above.

Similarly, the specification of a system S' that guaranties 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

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

3.3 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 \sqsubseteq as the lattice *order*, \sqcap as the *greatest lower bound*, or *meet*, \sqcup as the *least upper bound*, or *join*, **Fail** as the *bottom* element, and **Magic** as the *top* element. If S and T are monotonic property transformers, and q is a property, then these elements are formally defined by

$$\begin{aligned} (S \sqsubseteq T) &= (\forall q : S.q \subseteq T.q) \\ (S \sqcap T).q &= S.q \cap T.q \\ (S \sqcup T).q &= S.q \cup T.q \\ \text{Fail}.q &= \perp \\ \text{Magic}.q &= \top \end{aligned}$$

Note that \sqcap and \sqcup preserve monotonicity. Also note that, for any S , $\text{Fail} \sqsubseteq S$ and $S \sqsubseteq \text{Magic}$, so indeed Fail and Magic are the bottom and top elements, respectively. The transformer Fail does not guarantee any property. For any property q , we have $\text{Fail}.q = \perp$, 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 (for any q , $\text{Magic}.q = \top$). The problem with Magic is that it cannot be implemented.

All these lattice operations are also meaningful as operations on reactive systems. The order of this lattice ($S \sqsubseteq T$) 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. Note that in some works (e.g., [8, 18]) the notation is inverted, with $S \sqsubseteq T$ denoting S refines T , instead of T refines S as we employ here. In this paper we follow the same convention as in [4], which is also consistent with the definition of refinement for property transformers: $S \sqsubseteq T$ iff $S.q$ is a subset of $T.q$.

The 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 in Example 6 are completely defined by

$$S.q = \begin{cases} \top & \text{if } \{y \mid \square \diamond y\} \subseteq q \\ \perp & \text{otherwise} \end{cases}$$

and

$$S'.q = \begin{cases} \{x \mid \square \diamond x = 1\} & \text{if } \{y \mid \square \diamond y\} \subseteq q \\ \perp & \text{otherwise} \end{cases}$$

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 guaranties outputs where y is true infinitely often, then S can replace S' because S guaranties 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 $\prod_{i \in I} S_i$ where I is infinite. For example, assume that we have two systems S and S' which compute 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

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

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 .

Sequential composition of two systems S and T is simply the functional composition of S and T viewed as property transformers ($S \circ T$). We denote this type of 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 ($\forall q : \text{Skip}.q = q$) is the neutral element for sequential composition:

$$\text{Skip} ; S = S ; \text{Skip} = S, \text{ for any } S.$$

It is easy to see that sequential composition preserves monotonicity, that is, if S and T are both monotonic property transformers, then so is $S ; T$.

Definition 7. Two systems S and T are *incompatible* (w.r.t. the sequential composition $S ; T$) if

$$S ; T = \text{Fail}.$$

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.

Example 8. If for example we have S and T given by

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

then $(S ; T).q = S.(T.q) = S.\{x \mid x < 10\} = \perp$, 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 9. For a property transformer S , the *fail* of S , denoted $\text{fail}.S$, 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:

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

For example, the fail of **Magic** is \perp and the fail of **Fail** is \top .

Definition 10. For a property transformer S , the *guard* of S , denoted $\text{grd}.S$, is the set of input sequences for which the system does not behave *miraculously*. Formally:

$$\text{grd}.S = \neg S.\perp.$$

For example, the guard of **Magic** is \perp and the guard of **Fail** is \top . To see the intuition behind the definition of guard, observe that $S.\perp$ is the set of input sequences for which S is guaranteed to establish \perp , that is, the empty property, and therefore by monotonicity of S also any other output property. In other words, $S.\perp$ is the set of inputs for which S behaves miraculously, since the empty property \perp cannot be established.

For instance, taking S and T to be as defined in Example 8, we have: $\text{fail}.S = \perp$, $\text{grd}.S = \top$, and $\text{fail}.T = \text{grd}.T = \{x \mid x \geq 10\}$.

3.4 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 $p, q \in \Sigma^\omega \rightarrow \text{Bool}$ and $r \in \Sigma_1^\omega \rightarrow \Sigma_2^\omega \rightarrow \text{Bool}$ we define the *assert property transformer* $\{p\} : (\Sigma^\omega \rightarrow \text{Bool}) \rightarrow (\Sigma^\omega \rightarrow \text{Bool})$, and the *demonic update property transformer* $[r] : (\Sigma_2^\omega \rightarrow \text{Bool}) \rightarrow (\Sigma_1^\omega \rightarrow \text{Bool})$ as follows:

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

The assert transformer $\{p\}$ models a system which, given input sequence σ , produces σ as output when $p.\sigma$ is true, and it fails otherwise. In other words, only inputs satisfying p are legal for the assert system. The demonic update transformer $[r]$ models a system which establishes a post condition q when given as input a sequence σ if all sequences σ' with $r.\sigma.\sigma'$ are in q . Note that the assert and demonic update property transformers are monotonic, for any p and r .

Example 11. The property transformer for component B of Figure 1, discussed already in Example 5, is an example of an assert property transformer $\{p\}$, where $p.\sigma$ holds iff σ satisfies the LTL formula $\Box \Diamond (x = 1)$. Component E of Figure 2 is an example of a demonic update property transformer $[r]$, where r is the input-output trace relation corresponding to the LTL formula $\Box (y = 1 \Rightarrow x = 1)$.

3.5 Notation for assert and demonic update

Let us now introduce some preliminary syntactic notations to describe the two kinds of property transformers introduced above. Let R be an expression in x and y , for example, the LTL formula $\Box(x = 1 \Rightarrow y = 1)$. Recall that $\lambda x, y : R$ denotes the function $r : \Sigma_x^\omega \rightarrow \Sigma_y^\omega \rightarrow \mathbf{Bool}$ that takes two sequences x and y and returns true iff these two sequences satisfy the LTL formula. Since r is also an input-output relation on sequences, it defines the demonic property transformer $[r]$. However, a notation such as $[\lambda x, y : \Box(x = 1 \Rightarrow y = 1)]$ may be heavier than necessary. Therefore, we introduce a lighter notation, namely, $[x \rightsquigarrow y \mid \Box(x = 1 \Rightarrow y = 1)]$. In general, for any expression R in x and y , we use notation $[x \rightsquigarrow y \mid R]$ as equivalent to $[\lambda x, y : R]$. This notation also extends to systems with more than one inputs or outputs. For example, if R is $z = x + y$, and x, y are the inputs while z is the output, then $[x, y \rightsquigarrow z \mid z = x + y] = [\lambda(x, y), z : z = x + y]$.

For assert property transformers we introduce similar lighter notation. If P is an expression in x then $\{x \mid P\} = \{\lambda x : P\}$. For example, if P is $x \leq y$, then $\{x, y \mid x \leq y\} = \{\lambda(x, y) : x \leq y\}$. Note that a notation such as $\{x \mid x < 1\}$ is ambiguous, as it could mean the set of all, say, real numbers smaller than 1, or the assert property transformer $\{\lambda x : x < 1\}$. We will still use such notation, however, and such ambiguity will be resolved from the context.

Note also that in notations such as $\{x \mid P\}$ and $[x \rightsquigarrow y \mid R]$, the variables x and y are bound. 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 input 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, as in, for example

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

In this case, we drop the condition $x = y$ from the relation of S and we simply 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 syntax like the following:

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

This notation stands for

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

which is equivalent to

$$S = [\lambda(x, y, u, z, x'), (z', y', x'', y'', v) : x = x' = x'' \wedge y = y' = y'' \wedge z = z']$$

If S starts on a tuple where the first component is the same as the last component ($x = x'$), then S returns 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 behaves miraculously.

3.6 Properties of assert and demonic update

Theorem 12. *If $p, q \in \Sigma^\omega \rightarrow \mathbf{Bool}$, $r \in \Sigma_1^\omega \rightarrow \Sigma_2^\omega \rightarrow \mathbf{Bool}$, and $r' \in \Sigma_2^\omega \rightarrow \Sigma_3^\omega \rightarrow \mathbf{Bool}$, then*

1. $\text{Skip} = [x \rightsquigarrow x] = \{x \mid \mathbf{true}\}$ (Skip is both a demonic update and an assert transformer)

2. $\text{Magic} = [x \rightsquigarrow y \mid \text{false}]$, and $\text{Fail} = \{x \mid \text{false}\}$ (Magic is a demonic update, and Fail is an assert transformer)
3. $\{p\}; \{p'\} = \{p \cap p'\}$ and $\{x \mid P\}; \{x \mid P'\} = \{x \mid P \wedge P'\}$ (Assert transformers are closed under sequential composition)
4. $[r]; [r'] = [r \circ r']$ and $[x \rightsquigarrow y \mid R]; [y \rightsquigarrow z \mid R'] = [x \rightsquigarrow z \mid \exists y : R \wedge R']$ (Demonic updates are closed under sequential composition)
5. $\text{grd.}\{p\} = \top$, and $\text{grd.}[r] = \text{in.}r$ (Calculating the gard of assert and domonic update transformers)
6. $\text{fail.}\{p\} = \neg p$ and $\text{fail.}[r] = \perp$ (Calculating the fail of assert and domonic update transformers)

4 Relational property transformers

Definition 13. 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:

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

Note that every assert property transformer $\{p\}$ is a relational property transformer, because $\{p\} = \{p\}; [x \rightsquigarrow y \mid x = y]$. Also, every demonic update property transformer is a relational property transformer, because $[r] = \{x \mid \text{true}\}; [r]$. Also note that every relational property transformer is by definition monotonic. This is because every assert transformer is monotonic, every demonic update transformer is monotonic, and monotonicity is preserved by sequential composition. Finally, note that, as a special case, property p and relation r can be described by LTL formulas. This allows us to describe RPTs syntactically, by means of LTL formulas. This is illustrated in the example that follows.

Example 14. Consider again the division statement $z := x/y$ discussed in the introduction. Using LTL and the syntax introduced above, we can define several variants of property transformers which perform division on sequences of input pairs x and y , as follows:

$$\begin{aligned} S_1 &= [x, y \rightsquigarrow z \mid \square(y \neq 0 \wedge z = x/y)] \\ S_2 &= \{x, y \rightsquigarrow z \mid \square y \neq 0 \mid \square(y \neq 0 \wedge z = x/y)\} = \{x, y \mid \square y \neq 0\}; S_1 \end{aligned}$$

S_1 and S_2 are different property transformers. Both are relational, but they have different guards and fails. Specifically, $\text{fail.}S_1 = \perp$, whereas $\text{fail.}S_2 = (\diamond y = 0)$. This means that any input trace is legal for S_1 whereas only traces where y is never zero are legal for S_2 . On the other hand, $\text{grd.}S_1 = (\square y \neq 0)$, whereas $\text{grd.}S_2 = \top$. This means that S_2 never behaves miraculously, whereas S_1 behaves miraculously when the input assumption $\square y \neq 0$ is violated.

The next theorem states some important results, in particular regarding the compositionality (i.e., closure w.r.t. composition and other operations) of relational property transformers.

Theorem 15. *Let p, q be properties and r, r' be relations on sequences of appropriate types. Then:*

1. $\{p \mid r\}; \{p' \mid r'\} = \{x \rightsquigarrow z \mid p.x \wedge (\forall y : r.x.y \Rightarrow p'.y) \mid (r \circ r').x.z\}$ (relational property transformers are closed under sequential composition)
2. $\{p \mid r\} \sqcap \{p' \mid r'\} = \{p \wedge p' \mid r \vee r'\}$ (relational property transformers are closed under demonic choice)

3. $\{p \mid r\} = \{p \mid p \wedge r\}$ (precondition can be used in the input-output relation, e.g., for simplification)
4. $\{p \mid r\} \sqsubseteq \{p' \mid r'\} \Leftrightarrow (\forall x : p.x \Rightarrow p'.x) \wedge (\forall x, y : (p.x \wedge r'.x.y) \Rightarrow r.x.y)$ (necessary and sufficient condition for refinement)
5. $\text{grd.}(\{p \mid r\}) = \neg p \vee \text{in}.r$ (symbolic expression for the guard)
6. $\text{fail.}(\{p \mid r\}) = \neg p$ (symbolic expression for the fail predicate)

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*. An example of a transformer that may behave miraculously is transformer S_1 defined in Example 14. Often we are interested in systems that are guaranteed to never behave miraculously, i.e., in systems defined by transformers S such that $\text{grd}.S = \top$. In these cases we use relational property transformers of the form $\{\text{in}.r \mid r\}$. We call these RPTs *guarded*:

Definition 16. The *guarded system* of a relation r is the relational property transformer $\{r\} = \{\text{in}.r \mid r\}$.

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

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

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

$$\begin{aligned} & \{p \wedge \text{in}.r \mid r\} \\ = & \{\text{Theorem 15}\} \\ & \{p \wedge \text{in}.r \mid p \wedge \text{in}.r \wedge r\} \\ = & \{\text{Theorem 15}\} \\ & \{\text{in}.(p \wedge r) \mid p \wedge r\} \end{aligned}$$

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

Theorem 17. *If p is a property and r, r' are relations of appropriate types, then*

1. $\text{grd.}\{r\} = \top$ (guarded systems never behave miraculously)
2. $\text{Fail} = \{\perp\}$ and $\text{Skip} = \{x \rightsquigarrow x \mid \top\}$ (Fail and Skip are guarded)
3. $\{p\} = \{x \rightsquigarrow x \mid p.x\}$ and $\{p\} ; \{r\} = \{p \wedge r\}$ (assert transformers are guarded and assert can be moved inside a guarded transformer)
4. $\{r\} ; \{r'\} = \{x \rightsquigarrow z \mid \text{in}.r.x \wedge (\forall y : r.x.y \Rightarrow \text{in}.r'.y) \wedge (r \circ r').x.z\}$ (guarded systems are closed under sequential composition)
5. $\{r\} \sqcap \{r'\} = \{\text{in}.r \wedge \text{in}.r' \wedge (r \vee r')\}$ (guarded systems are closed under demonic choice)

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 $\text{grd.}[\perp] = \perp$. A less pathological example is the demonic update transformer S_1 from Example 14, which is also not a guarded system, because it behaves miraculously when y becomes 0. As the following lemma states, demonic updates are guarded systems if and only if they impose no requirements on the inputs.

Lemma 18. *The demonic update transformer $[r]$ is a guarded system if and only if $\text{in}.r = \text{true}$ and in this case we have $[r] = \{r\}$.*

Example 19. Here are some examples of guarded systems:

- **Havoc** = $[x \rightsquigarrow y \mid \text{true}]$: this demonic update transformer corresponds to a system which accepts any input sequence, and may generate an arbitrary output sequence. **Havoc** is a guarded system because it imposes no requirements on its input.
- **AssertLive** = $\{x \mid \Box(\Diamond x)\}$: this assert transformer corresponds to a system which requires its Boolean input to be infinitely often true.
- **LiveHavoc** = **AssertLive** ; **Havoc**: this system corresponds to the sequential composition of the previous two; it requires the input to be infinitely often true, and it makes no guarantees on the output (i.e., it can generate any output sequence).
- **ReqResp** = $[x \rightsquigarrow y \mid \Box(x \Rightarrow \Diamond y)]$: this demonic update transformer corresponds to a system which accepts any input sequence, and may generate an arbitrary output sequence, provided the request-response property *for every input there is eventually an output* is satisfied.

The fact that all these systems are guarded follows from Theorem 17 and Lemma 18. Note that **ReqResp** 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:

- **Havoc** ; **AssertLive** = **Fail**: this means that **Havoc** and **AssertLive** are incompatible. Indeed, since **Havoc** guarantees nothing about its output, it cannot meet the input requirements of **AssertLive**.
- **Havoc** ; **LiveHavoc** = **Fail**: for the same reason as above, **Havoc** and **LiveHavoc** are also incompatible.
- **ReqResp** ; **LiveHavoc** = **LiveHavoc**: this says that **ReqResp** and **LiveHavoc** are compatible, and in fact that their sequential composition is equivalent to **LiveHavoc**. This is indeed the case, because, in order to meet the input requirements of **LiveHavoc**, the **ReqResp** component must ensure that its output is infinitely often true. The only way for **ReqResp** to achieve that is to impose a requirement on its own input, namely, that its own input is infinitely often true as well. Since the names of input and output variables do not matter for the property transformer semantics, the result is identical to the property transformer **LiveHavoc**.

Example 20. Having introduced guarded systems, we can now give formal semantics to the components and diagrams introduced in Figures 1 and 2, from Example 1. The semantics of the components (boxes) in these figures are guarded monotonic property transformers, defined by LTL formulas. In particular, a component labeled with some formula ϕ corresponds to the guarded transformer $\{\phi\}$. For instance, component C introduced in Figure 2 corresponds to the guarded transformer $\{y \rightsquigarrow x \mid \Box(y = 1 \Rightarrow \Diamond x = 1)\}$, which is equivalent to the transformer **ReqResp** from Example 19. (Note that **ReqResp** uses x as the input and y as the output, whereas in C it is the other way around. This difference does not matter, as input and output variables are bound; semantically, the two systems define identical property transformers.)

We can also use some of the established results to reason about such components formally. For example, let us apply the results of Theorem 15 to see how checking refinement of systems specified in LTL can be reduced to checking satisfiability of quantified LTL formulas. Consider again Example 1, and in particular components D and E from Figure 2. Let ϕ_1 be the LTL formula of D and ϕ_2 be the LTL formula of E . Then, checking that E refines D amounts to checking $\{\phi_1\} \sqsubseteq \{\phi_2\}$. By Theorem 15, Part 4, checking $\{\phi_1\} \sqsubseteq \{\phi_2\}$ is equivalent to checking validity of the formula $\Phi = (\psi_1 \Rightarrow \psi_2 \wedge (\psi_1 \wedge \phi_2 \Rightarrow \phi_1))$, where $\psi_i = \text{in}.\phi_i$, for $i = 1, 2$. The formulas ψ_1 and ψ_2 can be obtained from ϕ_1 and ϕ_2 by existential quantification of the output variables. For example, $\psi_2 = \text{in}.\phi_2 = (\exists x : \Box(y = 1 \Rightarrow x = 1))$. In this specific example, quantifiers can be eliminated

in both cases of ψ_1 and ψ_2 , and this results in two pure LTL formulas: $\psi_1 = \Box \Diamond (y = 1)$ and $\psi_2 = \text{true}$. In general, however, LTL is not closed under quantifier elimination [19]. Therefore, Φ is generally an LTL formula with quantifiers. Checking validity of Φ amounts to checking (un)satisfiability of $\neg\Phi$. Satisfiability of quantified LTL is decidable, and methods such as those presented in [17, 11] can be used for that purpose.

We can also use some of the established results to reduce checking compatibility of components to checking satisfiability of formulas of the appropriate logic, as the following theorem states:

Theorem 21. *For property transformers S and T , and properties and relations p, p', r , and r' , of appropriate types, we have:*

1. If S is monotonic, then $S = \text{Fail}$ iff $\text{fail}.S = \top$.
2. If S and T are monotonic, then S and T are incompatible with respect to $S ; T$ iff $\text{fail}.(S ; T) = \top$.
3. $\{p \mid r\}$ and $\{p' \mid r'\}$ are incompatible with respect to $\{p \mid r\} ; \{p' \mid r'\}$ iff $\neg\exists x : p.x \wedge (\forall y : r.x.y \Rightarrow p'.y)$.
4. $\{r\}$ and $\{r'\}$ are incompatible with respect to $\{r\} ; \{r'\}$ iff $\neg\exists x : \text{in}.r.x \wedge (\forall y : r.x.y \Rightarrow \text{in}.r'.y)$.

For instance, Part 3 of Theorem 21 states that the composition of two relational property transformers $\{x \rightsquigarrow y \mid P \mid R\} ; \{y \rightsquigarrow z \mid P' \mid R'\}$ is valid (i.e., the two are compatible) iff the formula $P \wedge (\forall y : R \Rightarrow P')$ is satisfiable.

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 $\text{count}.x.i$ is the number of trues in x_0, x_1, \dots, x_i , then this system can be defined in the following way:

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

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:

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

where i is the index of the step, and we can 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 $\text{init}.u$, 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 $u_0 \in \text{init}$ and we are given some input sequence x_0, x_1, \dots , 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, \dots 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 nondeterminism when defining the property transformer based on such a symbolic transition system. To see the complications that may arise, consider another example:

$$\begin{aligned} \text{init}.u &= u \\ p.u.x &= u \\ r.u.u'.x.y &= (x = y) \end{aligned}$$

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 $u' = \text{false}$ 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 $(\forall i : u_i = \text{true})$ 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, \dots is *illegal* for a symbolic transition system if there is some $k \in \text{Nat}$ and some choice u_0, u_1, \dots of states and y_0, y_1, \dots of outputs such that $\text{init}.u_0$ 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 25 and discussion that follows). With this generalization, we define the illegal predicate on symbolic transition systems and input sequences, as follows:

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

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

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

where, recall, u^1 denotes the sequence u_1, u_2, \dots , i.e., the sequence of states starting from the second state u_1 instead of the initial state u_0 . If the predicate $\text{run}.r.u.x.y$ 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 22. 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 $\{\{ \text{init} \mid p \mid r \}\}$, as follows:

$$\{\{ \text{init} \mid p \mid r \}\}.q.x = \neg \text{illegal}.init.p.r.x \wedge (\forall u, y : (\text{init}.u_0 \wedge \text{run}.r.u.x.y) \Rightarrow q.y)$$

What the above definition states is that an input sequence x is in the set of input sequences of $\{\{ \text{init} \mid p \mid r \}\}$ 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 .

Before proceeding, let us make a remark on why we use similar, but different, notation for relational property transformers and for local property transformers. In the case of relational property transformers

we use notation such as $\{p \mid r\}$. Here, p and r are predicates over *sequences* (traces). For instance, p might be the LTL formula $\Box x = 1$. In the case of local property transformers we use notation such as $\{\{init \mid p \mid r\}\}$. Here, $init, p$, and r are local predicates over (input, output, and state) *variables*. For example, p in this case might be the predicate $u = 0 \Rightarrow x = 1$.

Note that so far our definition of symbolic transition systems is essentially semantic, since $init, p, r$ are semantic objects. In practice, we may use a syntax such as Boolean expressions for these elements. This is essentially the language used in symbolic model-checking tools like, say, NuSMV. If $Init, P$, and R are Boolean expressions possibly containing free the variables u and u', x and u, u', x, y , respectively then we define a syntax to describe local property transformers similar to the syntax we used for relational and guarded property transformers:

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

Example 23. For instance, we can define the property transformer for the counter system discussed in the beginning of this section as follows:

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

We can also prove that the non-deterministic example discussed above is equivalent to the Fail transformer:

$$\{\{x \rightsquigarrow u \rightsquigarrow y \mid u \mid u \mid y = x\}\} = \text{Fail}$$

that is, for all input sequences this system fails.

Lemma 24. *For a symbolic transition system $(init, p, r)$, the set of input sequences for which its local property transformer $\{\{init \mid p \mid r\}\}$ fails is equal to the set of its illegal input sequences:*

$$\text{fail}.\{\{init \mid p \mid r\}\} = \text{illegal}.init.p.r$$

5.1 Local property transformers are relational

The definition of a local property transformer is close to our intuition of how a system with state should operate, step by step, however, it is difficult to see immediately from this definition whether local transformers belong to the class of relational property transformers. The following theorem shows that this is indeed the case:

Theorem 25. *For any symbolic transition system $(init, p, r)$, we have:*

$$\begin{aligned} & \{\{init \mid p \mid r\}\} \\ = & [x \rightsquigarrow u, x \mid init.u_0] ; \{u, x \mid (\text{in}.r \text{ L } p).u.u^1.x\} ; [u, x \rightsquigarrow y \mid \Box r.u.u^1.x.y] \\ = & \{x \mid \forall u : init.u_0 \Rightarrow (\text{in}.r \text{ L } p).u.u^1.x\} ; [x \rightsquigarrow y \mid \exists u : init.u_0 \wedge \Box r.u.u^1.x.y] \end{aligned}$$

Theorem 25 shows that a local property transformer $\{\{init \mid p \mid r\}\}$ can be expressed as a sequential composition of assert and update transformers. Since the latter are special cases of relational transformers, and relational transformers are closed by sequential composition, this shows that local property transformers are relational. Moreover, the assert and update transformers used in the right-hand side of theorem above are constructed by applying some temporal operators to the *local* precondition p and the *local* input-output relation r . Here, p and r are *local* in the sense that they refer only to one step, i.e., they are predicates on state, input and output variables, and not on infinite sequences.

Theorem 25 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 $\text{in}.r$ depends anyway on the next state $(\text{in}.r.u.u'.x)$.

We call the precondition $(\text{in}.r \text{ L } p).u.u^1.x$ from the representation of the local system $\{\{init \mid p \mid r\}\}$ the *global precondition* of $\{\{init \mid p \mid r\}\}$. Similarly $\Box r.u.u^1.x.y$ is the *global input-output relation* of $\{\{init \mid p \mid r\}\}$.

5.2 Checking that symbolic transition systems refine their specification

An additional benefit of Theorem 25 is that it makes checking refinement of local systems against their specification (or against another system) easier. For example, suppose that we want to prove a refinement like

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

If we use the original definition of local system (Definition 22), then we need to expand the definition of $\{\!\!| \text{init} \mid p' \mid r' \!\!\}$ 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 Theorem 15, 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. Therefore, checking validity of this formula can be often reduced to simpler problems, e.g., satisfiability of LTL formulas, as explained in Example 20.

We can exploit Theorem 25 to obtain an analogous result for local transformers:

Theorem 26. *For $\text{init}, p, p', r, r'$ of appropriate types we have:*

$$\begin{aligned} & \{x \rightsquigarrow y \mid p \mid r\} \sqsubseteq \{\!\!| \text{init} \mid p' \mid r' \!\!\} \\ \Leftrightarrow & (\forall u, x : \text{init}.u_0 \wedge p.x \Rightarrow (\text{in}.r' \text{ L } p').u.u^1.x) \wedge (\forall u, x, y : \text{init}.u_0 \wedge p.x \wedge \Box r'.u.u^1.x.y \Rightarrow r.x.y) \end{aligned}$$

5.3 Sequential composition of local transformers

Theorems 15 and 25 allow us to calculate the sequential composition of two local systems, as follows:

Theorem 27.

$$\begin{aligned} & \{\!\!| \text{init} \mid p \mid r \!\!\}; \{\!\!| \text{init}' \mid p' \mid r' \!\!\} \\ = & \{x \mid \forall u, v : \text{init}.u_0 \wedge \text{init}'.v_0 \Rightarrow (\text{in}.r \text{ L } p).u.u^1.x \wedge (\forall y : \Box r.u.u^1.x.y \Rightarrow (\text{in}.r' \text{ L } p').v.v^1.y)\}; \\ & [x \rightsquigarrow z \mid \exists u, v : \text{init}.u_0 \wedge \text{init}'.v_0 \wedge \Box (r \circ \circ r').(u, v).(u^1, v^1).x.z] \end{aligned}$$

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

Ideally the composition of two local systems $S = \{\!\!| \text{init} \mid p \mid r \!\!\}$ and $S' = \{\!\!| \text{init}' \mid p' \mid r' \!\!\}$ would be a local system corresponding to the composition of the local transitions of S and S' . Unfortunately this is not the case. In the rest of this subsection we explain why this is the case. This will motivate the definition of a restricted class of local transformers, called *guarded* local transformers, which are analogous to guarded property transformers, and enjoy good closure properties.

We begin by defining the *local transition* of a local system to be the predicate transformer:

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

Note that $\text{localtran}.p.r$ is a *predicate* transformer, not a property transformer. Also note that here it suffices to consider p as a predicate on the current state and input only. The execution of $\text{localtran}.p.r$ starts from the input value x and the state u and if $p.u.x$ is true, then it computes the output value y and the new state u' such that $r.u.u'.x.y$ is true. The local transitions of S and S' have the local states u and v , respectively, and their composition will have the local state pairs (u, v) . In order to be able to compose the local transitions of S and S' , we add the state v to the local transition of S and the state u to local transition of S' :

$$\begin{aligned}\text{localtran}^*.p.r &= \{x, u, v \mid p.u.x\}; [x, u, v \rightsquigarrow y, u', v \mid r.u.u'.x.y] \\ \text{localtran}^*.p'.r' &= \{y, u, v \mid p'.v.y\}; [y, u, v \rightsquigarrow z, u, v' \mid r'.v.v'.y.z]\end{aligned}$$

We show what would be the local system for the composition of the local transitions of S and S' . We have

$$\begin{aligned}& \text{localtran}^*.p.r; \text{localtran}^*.p'.r' \\ = & \{\text{Theorem 15}\} \\ & \{x, u, v \mid p.u.x \wedge (\forall u', y : r.u.u'.x.y \Rightarrow p'.v.y)\}; [x, u, v \rightsquigarrow z, u, v' \mid (r.u.u') \circ (r'.v.v').x.z]\end{aligned}$$

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

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

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

$$\begin{aligned}& \llbracket \text{init}'' \mid p'' \mid r'' \rrbracket \\ = & \{x \mid \forall u, v : \text{init}''.(u_0, v_0) \Rightarrow (\text{in}.r'' \text{ L } p'').(u, v).(u^1, v^1).x\}; \\ & [x \rightsquigarrow z \mid \exists u, v : \text{init}''.(u_0, v_0) \wedge \square r''.(u, v).(u^1, v^1).x.z]\end{aligned}$$

Now, one might expect the equality $S; S' = \llbracket \text{init}'' \mid p'' \mid r'' \rrbracket$. Unfortunately this does not generally hold for arbitrary local systems S and S' . We do have, by definition

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

but there exist p, r, p', r', u, v , and x such that

$$\begin{aligned}& (\text{in}.r'' \text{ L } p'').(u, v).(u^1, v^1).x \\ \neq & (\text{in}.r \text{ L } p).u.u^1.x \wedge (\forall y : \square r.u.u^1.x.y \Rightarrow (\text{in}.r' \text{ L } p').v.v^1.y)\end{aligned} \tag{1}$$

For example, if we take

$$\begin{aligned}\text{init}.u &= (u = 0) \\ p.u.x &= \text{true} \\ r.u.u'.x.y &= (u = 0 \wedge u' = 1) \\ \text{init}'.v &= \text{true} \\ p'.v.y &= \text{false} \\ r'.v.v'.y.z &= \text{true}\end{aligned}$$

then (1) becomes true.

What happens in this case is that $S = \text{Magic}$ so $S; S' = \text{Magic}$, whereas $\llbracket \text{init}'' \mid p'' \mid r'' \rrbracket = \text{Fail}$, and therefore clearly $S; S' \neq \llbracket \text{init}'' \mid p'' \mid r'' \rrbracket$. Intuitively, 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 $\llbracket \text{init}'' \mid p'' \mid r'' \rrbracket$ starting from the same initial state $u = 0$ proceeds normally with the first step of S ($\{x, u \mid p.u.x\}; [x, u \rightsquigarrow y, u' \mid r.u.u'.x.y]$), and then tests p' , and it fails because p' is false.

5.4 Guarded local systems

As we have seen from the previous section, the composition of two local transformers is not necessarily a local transformer. This is because of the possible miraculous behavior of such 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 $\text{in}.r$. This is similar to what we have done in order to restrict general relational transformers to guarded systems, in Section 4.1.

Definition 28. The *guarded local system* of init and r is denoted by $\{\!\{ \text{init} \mid r \}\!\}$ and it is given by

$$\{\!\{ \text{init} \mid r \}\!\} = \{\!\{ \text{init} \mid \text{in}.r \mid r \}\!\}$$

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

Theorem 29. For init and r as in the definition of a local guarded system we have:

$$\{\!\{ \text{init} \mid r \}\!\} = [x \rightsquigarrow u, x \mid \text{init}.u_0] ; \{u, x \rightsquigarrow y \mid \square r.u.u^1.x.y\}$$

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

Theorem 30. For init , init' , r , and r' we have

$$\{\!\{ \text{init} \mid r \}\!\} ; \{\!\{ \text{init}' \mid r' \}\!\} = \{\!\{ \text{init}'' \mid \text{rel_comp}.r.r' \}\!\}$$

where

$$\text{init}''.(u, v) = \text{init}.u \wedge \text{init}'.v$$

and

$$\begin{aligned} & \text{rel_comp}.r.r'.(u, v).(u', v').x.z \\ = & (\text{in}.r.u.u'.x \wedge (\forall y : r.u.u'.x.y \Rightarrow \text{in}.r'.v.v'.y) \wedge ((r.u.u') \circ (r'.v.v')).x.z) \end{aligned}$$

5.5 Stateless systems

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

Theorem 31. For p , and r as in the definition of a stateless system, we have:

$$\begin{aligned} \{\!\{ \text{init} \mid p \mid r \}\!\} &= \{x \rightsquigarrow y \mid (\text{in}.r \text{ L } p).x \mid (\square r).x.y\} \\ &= \{\!\{ \text{in}.r \text{ L } p \mid \square r \}\!\} \end{aligned}$$

Based on this theorem we use the notation $\{\!\{ \text{in}.r \text{ L } p \mid \square r \}\!\}$ for a stateless system.

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

Theorem 32.

$$\begin{aligned} & \{\!\{ \text{in}.r \text{ L } p \mid \square r \}\!\} ; \{\!\{ \text{in}.r' \text{ L } p' \mid \square r' \}\!\} \\ = & \{x \mid (\text{in}.r \text{ L } p).x \wedge (\forall y : \square r.x.y \Rightarrow (\text{in}.r' \text{ L } p').y)\} ; [\square (r \circ r')] \end{aligned}$$

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

5.6 Guarded stateless systems

Definition 33. A *guarded stateless system* is a stateless system where $p = \text{in}.r$.

Theorem 34. For any r : $\{\text{in}.r \text{ L } \text{in}.r \mid \square r\} = \{\square r\}$.

This is because we have

$$\{\text{in}.r \text{ L } \text{in}.r \mid \square r\} = \{\square \text{in}.r \mid \square r\} = \{\text{in}.(\square r) \mid \square r\} = \{\square r\}$$

We use the notation $\{\square r\}$ for a guarded stateless system.

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

Theorem 35. For r , and r' we have

$$\{\square r\}; \{\square r'\} = \{\square (\text{rel_comp}.r.r')\}$$

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

5.6.1 Local properties

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

Definition 36. For a property q , the i -th *projection* of q is a predicate on states given by

$$\text{proj}.q.i.s = (\exists \sigma : q.\sigma \wedge \sigma_i = s)$$

and a property q is a *piecewise local property* if it satisfies the condition

$$(\forall \sigma : (\forall i : \text{proj}.q.i.\sigma_i) \Rightarrow q.\sigma)$$

Equivalently, a property q is piecewise local if there exist some predicates p_0, p_1, \dots 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 $q = (\square (\diamond x))$ is not local because $\sigma = (\lambda i : \text{false})$ satisfies the condition $(\forall i : \text{proj}.q.i.(\sigma.i))$, but $\sigma \notin q$.

Lemma 37. If r is a state relation, then

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

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

Example 38. 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 $x > 0$:

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

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

$$S.q = \square (x > 0 \wedge \diamond x < 9)$$

If we want to see under what conditions on the input the output of S satisfies $q' = \square \diamond y = 10$, then we should calculate $S.q'$. In this case we have $S.q' = \perp$. 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 [18] to infinite behaviors and liveness. We note that the theory proposed in [18] 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 [18].

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

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

This symbolic transition system has 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 reset 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

$$\begin{aligned} & \{ \mathit{init} \mid p \mid r \} \\ = & \{ x \mid \forall u : \mathit{init}.u_0 \Rightarrow (\mathit{in}.r \text{ L } p).u.u^1.x \} ; [x \rightsquigarrow y \mid \exists u : \mathit{init}.u_0 \wedge \square r.u.u^1.x.y] \end{aligned} \quad (2)$$

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 $\square \diamond y$ to the input-output relation of (2). So the full example is

$$\begin{aligned} & \text{EXAMPLE} \\ = & \{ x \mid \forall u : \mathit{init}.u_0 \Rightarrow (\mathit{in}.r \text{ L } p).u.u^1.x \} ; [x \rightsquigarrow y \mid \exists u : \mathit{init}.u_0 \wedge \square r.u.u^1.x.y \wedge \square \diamond y] \end{aligned} \quad (3)$$

In this example the state 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 trace is accepted by this system only if this property is not violated. For example the input sequence $x_0 = \text{true}$, $x_1 = \text{false}$, \dots maintains this property. On the other hand the property $\square \diamond y$ is a liveness property which is guaranteed by the system, regardless of the input. 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 $\square \diamond y$ for all inputs that do not fail, i.e., for all input traces which satisfy

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

We have

$$\forall x : \text{EXAMPLE}.(\{y \mid \square \diamond y\}).x = \mathit{prec}_g.x \quad (4)$$

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

$$\text{EXAMPLE} \sqsubseteq [x \rightsquigarrow y \mid \square y].$$

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

$$\{x \mid \square (x \Leftrightarrow \neg \bigcirc x)\} ; \text{EXAMPLE}$$

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

$$\begin{aligned} & \{x \mid \Box(x \Leftrightarrow \neg \bigcirc x)\}; \text{EXAMPLE} \\ \sqsubseteq & \\ & \{x \mid \Box(x \Leftrightarrow \neg \bigcirc x)\}; \{\!\!| \text{init} \mid p \mid r \!\!\} \\ \sqsubseteq & \\ & \{\!\!| \text{init} \mid p \mid r \!\!\} \end{aligned}$$

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

$$\text{EXAMPLE} = \text{EXAMPLE} ; \{y \mid \Box \diamond y\}$$

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 monotonic property transformer semantics for reactive systems. The semantics supports refinement, composition, compatibility, demonic choice, unbounded nondeterminism, and other interesting system properties. The semantics also supports angelic choice: we have not specifically exploited this feature here and we leave it for future work. The semantics can be used to specify and reason about both safety and liveness properties. Our framework allows to describe systems using a variety of formalisms, from higher order logic, to temporal logic, to symbolic transition systems. The framework is compositional, in particular if we restrict ourselves to the most realistic case of guarded systems, which cannot behave miraculously, and enjoy good closure properties. Our work generalizes previous work on relational interfaces to systems with infinite behavior and liveness properties. Future work includes studying more operators (e.g., angelic choice), and extending the framework to continuous-time and hybrid systems.

References

- [1] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
- [2] Ralph-Johan Back. *On the correctness of refinement in program development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978.
- [3] Ralph-Johan Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, pages 67–93. Springer, 1990.
- [4] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus. A systematic Introduction*. Springer, 1998.
- [5] Ralph-Johan Back and Joakim Wright. Trace refinement of action systems. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR '94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 367–384. Springer Berlin Heidelberg, 1994.
- [6] Ralph-Johan Back and Qiwen Xu. Refinement of fair action systems. *Acta Informatica*, 35(2):131–165, 1998.

- [7] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer, 2001.
- [8] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Foundations of Software Engineering (FSE)*. ACM Press, 2001.
- [9] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. MIT Press, Cambridge, MA, USA, 1989.
- [10] David Harel and Amir Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. 1985.
- [11] Yonit Kesten and Amir Pnueli. A complete proof system for QPTL. In *LICS*, June 1995.
- [12] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.
- [13] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [14] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Oct 1977.
- [15] Viorel Preteasa. Formalization of refinement calculus for reactive systems. *Archive of Formal Proofs*, June 2014. <http://afp.sf.net/entries/RefinementReactive.shtml>, Formal proof development. *Review pending*.
- [16] Viorel Preteasa. Refinement algebra with dual operator. *Science of Computer Programming*, 92, Part B(0):179 – 210, 2014. Selected papers from the Brazilian Symposium on Formal Methods (SBMF 2011).
- [17] Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.
- [18] Stavros Tripakis, Ben Lickly, Thomas A. Henzinger, and Edward A. Lee. A theory of synchronous relational interfaces. *ACM Trans. Program. Lang. Syst.*, 33(4):14:1–14:41, July 2011.
- [19] Pierre Wolper. Temporal logic can be more expressive. In *Foundations of Computer Science*, pages 340–348, 1981.