The Refinement Calculus of Reactive Systems

Viorel Preoteasa
Huld Finland
viorel.preoteasa@gmail.com

Iulia Dragomir
GMV Madrid
giuliadragomir@gmail.com

Stavros Tripakis (corresponding author)
Northeastern University
stavros@northeastern.edu,stavros.tripakis@gmail.com

November 16, 2021

Abstract

The Refinement Calculus of Reactive Systems (RCRS) is a compositional formal framework for modeling and reasoning about reactive systems. RCRS provides a language which can describe atomic components as symbolic transition systems or QLTL formulas, and composite components formed using three primitive composition operators: serial, parallel, and feedback. The semantics of the language is given in terms of monotonic property transformers, an extension of monotonic predicate transformers to reactive systems. RCRS can specify both safety and liveness properties. It can also model input-output systems which are both non-deterministic and non-input-receptive (i.e., which may reject some inputs at some points in time), and can thus be seen as a behavioral type system. RCRS provides a set of techniques for symbolic computer-aided reasoning, including compositional static analysis and verification. RCRS comes with a publicly available implementation which includes a complete formalization of the RCRS theory in the Isabelle proof assistant.

Keywords: Compositionality; Refinement; Verification; Reactive systems

Contents

1 Introduction

This paper presents the *Refinement Calculus of Reactive Systems* (RCRS), a comprehensive framework for compositional modeling of and reasoning about reactive systems. RCRS originates from the precursor theory of *synchronous relational interfaces* [85, 86], and builds upon the classic *refinement calculus* [12]. A number of publications on RCRS exist [72, 34, 73, 69, 36, 37]. This paper collects some of these results and extends them in significant ways. The novel contributions of this paper and relation to our previous work are presented in §1.1.

The motivation for RCRS stems from the need for a compositional treatment of reactive systems. Generally speaking, compositionality is a divide-and-conquer principle. As systems grow in size, they grow in complexity. Therefore dealing with them in a monolithic manner becomes unmanageable. Compositionality comes to the rescue, and takes many forms [84]. Many industrial-strength systems have employed for many years mechanisms for compositional modeling. An example is the Simulink tool from the Mathworks. Simulink is based on the widespread notation of hierarchical block diagrams. Such diagrams are both intuitive, and naturally compositional: a block can be refined into sub-blocks, sub-sub-blocks, and so on, creating hierarchical models of arbitrary depth. This allows the user to build large models (many thousands of blocks) while at the same time managing their complexity (at any level of the hierarchy, only a few blocks may be visible).

But Simulink's compositionality has limitations, despite its hierarchical modeling approach. Even relatively simple problems, such as the problem of modular code generation (generating code for a block independently from context), require techniques not always available in standard code generators [57, 56]. Perhaps more serious, and more relevant in the context of this paper, is Simulink's lack of formal semantics, and consequent lack of rigorous analysis techniques that can leverage the advances in the fields of computer-aided verification and programming languages.

RCRS provides a compositional formal semantics for Simulink in particular, and hierarchical block diagram notations in general, by building on well-established principles from the formal methods and programming language domains. In particular, RCRS relies on the notion of *refinement* (and its counterpart, *abstraction*) which are both fundamental in system design. Refinement is a binary relation between components, and ideally characterizes *substitutability*: the conditions under which some component can replace another component, without compromising the behavior of the overall system. RCRS refinement is *compositional* in the sense that it is preserved by composition: if A' refines A and B' refines B, then the composition of A' and B' refines the composition of A and B.

RCRS can be viewed as a refinement theory. It can also be viewed as a *behavioral type system*, similar to type systems for programming languages, but targeted to reactive systems. By *behavioral* we mean a type system that can capture not just data types of input and output ports of components (bool, int, etc.), but also complete specifications of the behavior of those components. As discussed more extensively in [86], such a behavioral type system has advantages over a full-blown verification system, as it is more lightweight. For instance, a type system allows type checking, which does not require the user to provide a formal specification of the correctness properties that a model must satisfy. The model must simply *type-check*.

As also argued in [86], in order to have a type system it is essential for a framework to be able to express *non-input-receptive* (also called *non-input-enabled* or *non-input-complete*) components, i.e., components that reject some input values. RCRS allows this. To see why non-input-receptiveness is essential consider, for example, a square-root component which requires its input to be non-negative. Such a component can be described in RCRS alternatively as: either (1) a non-input-receptive component $C_\wedge$ with input-output *contract* $x \geq 0 \wedge y = \sqrt{x}$ (where x, y are the input and output variables, respectively); or (2) an input-receptive component $C_\to$ with contract $x \geq 0 \to y = \sqrt{x}$. Now, connecting the non-input-receptive square-root component $C_\wedge$ to a component which outputs $x = -1$ results in a type error (in RCRS this is called *incompatibility*). Connecting $C_\wedge$ to a non-deterministic component which outputs an arbitrary value for x (this can be specified by the formula/contract *true*) also results in a type error in RCRS. Yet in both these cases, replacing $C_\wedge$ by the input-receptive component $C_\rightarrow$ results in no type error (no incompatibility). Instead, $C_\rightarrow$ will simply output a non-deterministically chosen value, even though the requirement that the input to square root is non-negative is not satisfied. This simple example demonstrates the need for non-input-receptiveness, and also illustrates the concept of type-checking in the RCRS context. A more extensive argument for non-input-receptiveness can be found in [86] (see also [88]). Let us also remark that input-receptive components could be thought of as "programs that terminate on all inputs" whereas non-input-receptive components could be thought of as "programs that terminate on some inputs". However, this is a matter of interpretation. RCRS as a semantic framework is agnostic to whether components are internally programs implemented in a standard programming language, or something entirely different.

RCRS allows components which are both non-input-receptive and non-deterministic. This combination results in a *game-theoretic* interpretation of the composition operators, like in *interface theories* [25, 86]. Refinement also becomes game-theoretic, as in *alternating refinement* [7]. Game-theoretic composition can be used for an interesting form of type inference. For example, if we connect the non-input-receptive square-root component C_1 above to a non-deterministic component C_3 with input-output contract $x \geq u+1$ (where x is the output of C_3, and u its input), and apply the (game-theoretic) serial composition of RCRS, we obtain the condition $u \geq -1$ on the external input of the overall composition. The constraint $u \geq -1$ represents the weakest condition on u which ensures compatibility of the connected components.

In a nutshell, RCRS consists of the following elements:

1. A modeling language (syntax), which can describe *atomic* components, and *composite* components formed by a small number of primitive composition operators (serial, parallel, and feedback). The language is described in §3.

2. A formal semantics, presented in §4. Component semantics are defined in terms of *monotonic property transformers* (MPTs). MPTs are extensions of monotonic *predicate* transformers used in theories of programming languages, and in particular in refinement calculus [12]. Predicate transformers transform sets of *post-states* (states reached by the program after its computation) into sets of *pre-states* (states where the program begins). Property transformers transform sets of a component's *output traces* (infinite sequences of output values) into sets of *input traces* (infinite sequences of input values). Using this semantics we can express both safety and liveness properties.

   MPTs are very general objects. In practice the systems we deal with fall into restricted subclasses of MPTs which are both easier to represent syntactically and also to manipulate symbolically. Section 4 includes a detailed study of MPT subclasses and their corresponding closure properties with respect to the composition operators. In particular, we show that the restricted subclasses of *relational property transformers* (RPTs) and *guarded property transformers* (GPTs) are both closed under serial and parallel composition. We also show that the semantics of all atomic RCRS components are GPTs, and that components with feedback are also GPTs as long as they are deterministic and do not contain *algebraic loops* (i.e., instantaneous feedback).

3. A set of symbolic reasoning techniques, described in §5. In particular, RCRS offers techniques to

   - compute the symbolic representation of a composite component from the symbolic representations of its sub-components;
   - simplify composite components into atomic components;
   - reduce checking refinement between two components to checking satisfiability of certain logical formulas;
   - reduce input-receptiveness and compatibility checks to satisfiability;
   - compute the legal inputs of a component symbolically.

We note that these techniques are for the most part *logic-agnostic*, in the sense that they do not depend on the particular logic used to represent components. In addition, many of these techniques are purely syntactic, and therefore very efficient.

4. A toolset, described briefly in §6. The toolset consists mainly of:

   - a full implementation of the RCRS theory (more than 27k lines of Isabelle code) in the Isabelle proof assistant [65];
   - a translator of Simulink diagrams into RCRS code.

   Our implementation is open-source and publicly available from `http://rcrs.gitlab.io/`.

RCRS is inspired by and shares key principles (e.g., refinement) with existing formal compositional frameworks such as FOCUS [18], input-output automata [58], reactive modules [6], interface automata [25], and Dill's trace theory [32]. At the same time, RCRS differs and complements these frameworks in important ways. For instance, FOCUS, IO-automata, and reactive modules, are limited to input-receptive systems, while RCRS is explicitly designed to handle non-input-receptive specifications. An extensive discussion of how RCRS is related to these and other works is provided in Section 7.

1.1 Novel contributions of this paper and relation to our prior work

Several of the ideas behind RCRS originated in the theory of synchronous relational interfaces [85, 86]. The main novel contributions of RCRS w.r.t. that theory are: (1) RCRS is based on the semantic foundation of monotonic property transformers, whereas relational interfaces are founded on relations; (2) RCRS can handle liveness properties, whereas relational interfaces can only handle safety; (3) RCRS has been completely formalized and most results reported in this and other RCRS papers have been proven in the Isabelle proof assistant; (4) RCRS comes with a publicly available toolset (`http://rcrs.gitlab.io/`) which includes the Isabelle formalization, a Translator of Simulink hierarchical block diagrams, and a Formal Analyzer which performs, among other functions, compatibility checking, refinement checking, and automatic simplification of RCRS contracts [34, 36, 37].

RCRS was introduced in [72], which focuses on monotonic property transformers as a means to extend relational interfaces with liveness properties. [72] covers serial composition, but not parallel nor feedback. It also does not cover symbolic reasoning nor the RCRS implementation. Feedback is considered in [73], with a particular aim of studying *instantaneous feedback* for non-deterministic and non-input-receptive systems. The study of instantaneous feedback is an interesting problem, but beyond the scope of the current paper. In this paper we consider non-instantaneous feedback, i.e., feedback for systems without same-step cyclic dependencies (no *algebraic loops*).

[34] presents part of the RCRS implementation, focusing on the translation of Simulink (and hierarchical block diagrams in general) into an algebra of components with three composition primitives, serial, parallel, and feedback, like RCRS. As it turns out, there is not a unique way to translate a graphical notation like Simulink into an algebraic formalism like RCRS. The problem of how exactly to do it and what are the trade-offs is an interesting one, but beyond the scope of the current paper. This problem is studied in depth in [34] which proposes three different translation strategies and evaluates their pros and cons. [34] leaves open the question whether the results obtained by the different translations are equivalent. This question is settled in [70], by proving that a class of translations, including the ones proposed in [34], are semantically equivalent for any input block diagram. [69] also concerns the RCRS implementation, discussing solutions to subtle typing problems that arise when translating Simulink diagrams into RCRS/Isabelle code.

In summary, the current paper does not cover the topics covered in [34, 73, 69, 70], only briefly covers the RCRS Toolset, and can be seen as a significantly revised and extended version of [72], focusing on the RCRS theory. The main novel contributions with respect to [72] are the following: (1) a language of components (§3); (2) a revised MPT semantics (§4), including in particular novel operators for feedback (§4.1.4), a classification of MPT subclasses (§4.2), and a complete semantics of both atomic and composite components in terms of MPTs (§4.3); (3) a new section on symbolic reasoning (§5).

2 Preliminaries

Sets, types. We use capital letters X, Y, Σ, ... to denote types or sets, and small letters to denote elements of these types $x \in X$, $y \in Y$, etc. We denote by $\mathbb{B}$ the type of Boolean values true and false. We use $\wedge$, $\vee$, $\Rightarrow$, and $\neg$ for the Boolean operations. The type of natural numbers is denoted by $\mathbb{N}$, while the type of real numbers is denoted by $\mathbb{R}$. The Unit type contains a single element denoted $()$.

Cartesian product. For types X and Y, $X \times Y$ is the Cartesian product of X and Y, and if $x \in X$ and $y \in Y$, then (x, y) is a tuple from $X \times Y$. The empty Cartesian product is Unit. We assume that we have only flat products $X_1 \times \ldots \times X_n$, and then we have

$$(X_1 \times \ldots \times X_n) \times (Y_1 \times \ldots \times Y_m) = X_1 \times \ldots \times X_n \times Y_1 \times \ldots \times Y_m$$

Functions. If X and Y are types, $X \to Y$ denotes the type of *functions* from X to Y. The function type constructor associates to the right (e.g., $X \to Y \to Z = X \to (Y \to Z)$) and the function interpretation associates to the left (e.g., $f(x)(y) = (f(x))(y)$). In order to construct functions we use lambda notation, e.g., $(\lambda x, y : x + y + 1) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. Similarly, we can have tuples in the definition of functions, e.g., $(\lambda(x, y) : x + y + 2) : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$. The composition of two functions $f : X \to Y$ and $g : Y \to Z$, is a function denoted $g \circ f : X \to Z$, where $(g \circ f)(x) = g(f(x))$.

Predicates. A *predicate* is a function returning Booleans, e.g., $p : X \to Y \to \mathbb{B}$ with $p(x)(y) = (x = y)$. We define the *smallest predicate* $\bot : X \to \mathbb{B}$ where $\bot(x) = \mathsf{false}$ for all $x \in X$. The *greatest predicate* is $\top : X \to \mathbb{B}$, with $\top(x) = \mathsf{true}$ for all $x \in X$. We will often interpret predicates as sets. A predicate $p : X \to \mathbb{B}$ can be viewed as the set of all $x \in X$ such that $p(x) = \mathsf{true}$. For example, viewing two predicates $p, q : X \to \mathbb{B}$ as sets, we can write $p \subseteq q$, meaning that for all x, $p(x) \Rightarrow q(x)$.

Relations. A *relation* is a predicate with at least two arguments, e.g., $r : X \to Y \to \mathbb{B}$. For such a relation r, we denote by $\mathsf{in}(r) : X \to \mathbb{B}$ the predicate $\mathsf{in}(r)(x) = (\exists y : r(x)(y))$. If the relation r has more than two arguments, then we define $\mathsf{in}(r)$ similarly by quantifying over the last argument.

We extend point-wise all operations on Booleans to operations on predicates and relations. For example, if $r, r' : X \to Y \to \mathbb{B}$ are two relations, then $r \wedge r'$ and $r \vee r'$ are the relations given by $(r \wedge r')(x)(y) = r(x)(y) \wedge r'(x)(y)$ and $(r \vee r')(x)(y) = r(x)(y) \vee r'(x)(y)$. We also introduce the order on relations $r \subseteq r' = (\forall x, y : r(x)(y) \Rightarrow r'(x)(y))$.

The composition of two relations $r : X \to Y \to \mathbb{B}$ and $r' : Y \to Z \to \mathbb{B}$ is a relation $(r \circ r') : X \to Z \to \mathbb{B}$, where $(r \circ r')(x)(z) = (\exists y : r(x)(y) \wedge r'(y)(z))$.

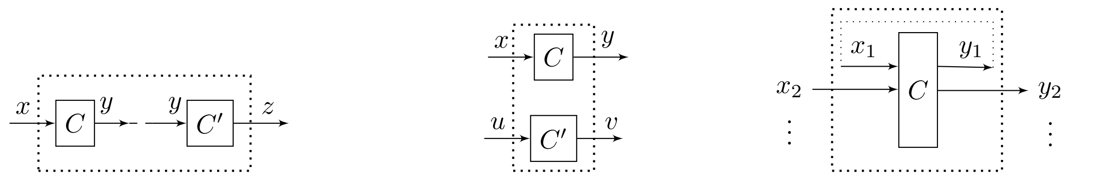
Infinite sequences. If Σ is a type, then $\Sigma^\omega = (\mathbb{N} \to \Sigma)$ is the set of all *infinite sequences* over Σ, also called *traces*. For a trace $\sigma \in \Sigma^\omega$, let $\sigma_i = \sigma(i)$ be the i-th element in the trace. Let $\sigma^i \in \Sigma^\omega$ denote the suffix of σ starting from the i-th step, i.e., $\sigma^i = \sigma_i \sigma_{i+1} \cdots$. We often view a pair of traces $(\sigma, \sigma') \in \Sigma^\omega \times \Sigma'^\omega$ as being also a trace of pairs $(\lambda i : (\sigma_i, \sigma'_i)) \in (\Sigma \times \Sigma')^\omega$.

Properties. A *property* is a predicate p over a set of infinite sequences. Formally, $p \in (\Sigma^\omega \to \mathbb{B})$. Just like any other predicate, a property can also be viewed as a set. In particular, a property can be viewed as a set of traces.

3 Language

3.1 An Algebra of Components

We model systems using a simple language of *components*. The grammar of the language is as follows:



(a) Serial composition: $C\,;C'$ (b) Parallel composition: $C \parallel C'$ (c) Feedback composition: $\texttt{fdbk}(C)$

Figure 1: The three composition operators of RCRS.

$$\begin{array}{rcl} \text{component} & ::= & \text{atomic\_component} \mid \text{composite\_component} \\ \text{atomic\_component} & ::= & \text{STS\_component} \mid \text{QLTL\_component} \\ \text{STS\_component} & ::= & \text{GEN\_STS\_component} \mid \text{STATELESS\_STS\_component} \\ & & \mid \text{DET\_STS\_component} \mid \text{DET\_STATELESS\_STS\_component} \\ \text{composite\_component} & ::= & \text{component}\ ;\ \text{component} \mid \text{component} \parallel \text{component} \mid \texttt{fdbk}(\text{component}) \end{array}$$

The elements of the above grammar are defined in the remainder of this section, where examples are also given to illustrate the language. In a nutshell, the language contains *atomic* components of two kinds: atomic components defined as *symbolic transition systems* (STS_component), and atomic components defined as quantified first-order LTL (QLTL) formulas over input and output variables (QLTL_component).

STS components are split in four categories: general STS components, stateless STS components, deterministic STS components, and deterministic stateless STS components. Semantically, the general STS components subsume all the other more specialized STS components, but we introduce the specialized syntax because symbolic compositions of less general components become simpler, as we shall explain in the sequel (see §5).

Also, as it turns out, atomic components of our framework form a lattice, shown in Fig. 8, from the more specialized ones, namely, deterministic stateless STS components, to the more general ones, namely QLTL components. The full definition of this lattice will become apparent once we provide a symbolic transformation of STS to QLTL components, in §5.1.

Apart from atomic components, the language also allows one to form *composite* components, by composing (atomic or other composite) components via three composition operators: serial ;, parallel $\parallel$, and feedback `fdbk`, as depicted in Fig. 1. The serial composition of two components C, C' is formed by connecting the output(s) of C to the input(s) of C'. Their parallel composition is formed by "stacking" the two components on top of each other without forming any new connections. The feedback of a component C is obtained by connecting the first output of C to its first input.

Our language is inspired by graphical notations such as Simulink, and hierarchical block diagrams in general. But our language is textual, not graphical. An interesting question is how to translate a graphical block diagram into a term in our algebra. We will not address this question here, as the issue is quite involved. We refer the reader to [34], which includes an extensive discussion on this topic. Suffice it to say here that there are generally many possible translations of a graphical diagram into a term in our algebra (or generally any algebra that contains primitive serial, parallel, and feedback composition operators). These translations achieve different tradeoffs in terms of size, readability, computational properties, and so on. See [34] for details.

Example 1. As an example, consider the Simulink diagram shown in Fig. 2. This diagram can be represented in our language as a composite component `Sum` defined as

$$\texttt{Sum} = \texttt{fdbk}(\texttt{Add}\,;\texttt{UnitDelay}\,;\texttt{Split})$$

where `Add`, `UnitDelay`, and `Split` are atomic components (for a definition of these atomic components see §3.2). Here `Split` models the "fan-out" element in the Simulink diagram (black bullet) where the output

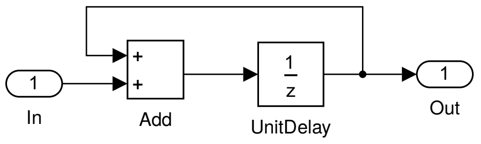


Figure 2: A Simulink diagram modeling the 1-step delayed sum of its input `In`. Each atomic block as well as the entire system can be formalized as STS components (see §3.2). The entire system can also be formalized as a composite component (see below).

wire of `UnitDelay` splits in two wires going to `Out` and back to `Add`.[1] A graphical representation of the composition `Add` ; `UnitDelay` ; `Split` is given in Figure 3. The operator `fdbk` connects the unnamed output from this figure to the unnamed input.

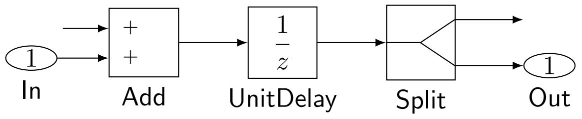


Figure 3: Graphical representation of the composition `Add` ; `UnitDelay` ; `Split`

3.2 Symbolic Transition System Components

We introduce all four categories of STS components and at the same time provide syntactic mappings from specialized STS components to general STS components.

3.2.1 General STS Components

A *general symbolic transition system component* (general STS component) is a transition system described symbolically, with Boolean expressions over input, output, state, and next state variables defining the initial states and the transition relation. When we say "Boolean expression" (here and in the definitions that follow) we mean an expression of type $\mathbb{B}$, in some arbitrary logic, not necessarily restricted to propositional logic. For example, if x is a variable of numerical type, then $x > 0$ is a Boolean expression. In the definition that follows, s' denotes the *primed*, or *next state* variable, corresponding to the *current state* variable s. Both can be vectors of variables. For example, if $s = (s_1, s_2)$ then $s' = (s'_1, s'_2)$. We assume that s' has the same type as s.

Definition 1 (STS component)**.** *A* (general) STS component *is a tuple*

$$\texttt{sts}(x : \Sigma_x, y : \Sigma_y, s : \Sigma_s, \mathit{init\_exp} : \mathbb{B}, \mathit{trs\_exp} : \mathbb{B})$$

where x, y, s are input, output and state variables (or tuples of variables) of types $\Sigma_x, \Sigma_y, \Sigma_s$, respectively, $\mathit{init\_exp}$ is a Boolean expression on s (in some logic), and $\mathit{trs\_exp}$ is a Boolean expression on x, y, s, s' (in some logic).

Intuitively, an STS component is a non-deterministic system which for an infinite input sequence $\sigma_x \in \Sigma_x^\omega$ produces as output an infinite sequence $\sigma_y \in \Sigma_y^\omega$. The system starts non-deterministically at some state $\sigma_s(0)$ satisfying *init_exp*. Given first input $\sigma_x(0)$, the system non-deterministically computes output $\sigma_y(0)$ and next state $\sigma_s(1)$ such that *trs_exp* holds (if no such values exist, then the input $\sigma_x(0)$ is *illegal*, as

[1]Note that the Simulink input and output ports `In` and `Out` are not explicitly represented in `Sum`. They are represented implicitly: `In` corresponds to the second input of `Add`, which carries over as the unique external input of `Sum` (thus, `Sum` is an "open system" in the sense that it has open, unconnected, inputs); `Out` corresponds to the second output of `Split`, which carries over as the unique external output of `Sum`.

discussed in more detail below). Next, it uses the following input $\sigma_x(1)$ and state $\sigma_s(1)$ to compute $\sigma_y(1)$ and $\sigma_s(2)$, and so on.

We will sometimes use the term *contract* to refer to the expression *trs_exp*. Indeed, *trs_exp* can be seen as specifying a contract between the component and its environment, in the following sense. At each step in the computation, the environment must provide input values that do not immediately violate the contract, i.e., for which we can find values for the next state and output variables to satisfy *trs_exp*. Then, it is the responsibility of the component to find such values, otherwise it is the component's "fault" if the contract is violated. This game-theoretic interpretation is similar in spirit with the classic refinement calculus for sequential programs [12].

We use Σ_x, Σ_y, Σ_s in the definition above to emphasize the types of the input, output and the state, and the fact that, when composing components, the types should match. However, in practice we often omit the types, unless they are required to unambiguously specify a component. Also note that the definition does not fix the logic used for the expressions *init_exp* and *trs_exp*. Indeed, our theory and results are independent from the choice of this logic. The choice of logic matters for algorithmic complexity and decidability. We will return to this point in §5. Finally, for simplicity, we often view the formulas *init_exp* and *trs_exp* as semantic objects, namely, as predicates. Adopting this view, *init_exp* becomes the predicate $init : \Sigma_s \to \mathbb{B}$, and *trs_exp* the predicate $trs\_exp : \Sigma_s \to \Sigma_x \to \Sigma_s \to \Sigma_y \to \mathbb{B}$. Equivalently, *trs_exp* can be interpreted as a relation $trs\_exp : (\Sigma_s \times \Sigma_x) \to (\Sigma_s \times \Sigma_y) \to \mathbb{B}$.

Throughout this paper we assume that *init_exp* is satisfiable, meaning that there is at least one valid initial state.

Examples. In the examples provided in this paper, we often specify systems that have tuples as input, state and output variables, in different equivalent ways. For example, we can introduce a general STS component with two inputs as $\texttt{sts}((n : \mathbb{N}, x : \mathbb{R}), s : \mathbb{R}, y : \mathbb{R}, s > 0, s' > s \wedge y + s = x^n)$, but also as $\texttt{sts}((n, x) : \mathbb{N} \times \mathbb{R}, s : \mathbb{R}, y : \mathbb{R}, s > 0, s' > s \wedge y + s = x^n)$, or $\texttt{sts}(z : \mathbb{N} \times \mathbb{R}, y : \mathbb{R}, s : \mathbb{R}, s > 0, s' > s \wedge y + s = \mathsf{snd}(z)^{\mathsf{fst}(z)})$, where fst and snd return the first and second elements of a pair.

Example 2. Let us model a system that at every step i outputs the input received at previous step $i - 1$ (assume that the initial output value is 0). This corresponds to Simulink's commonly used `UnitDelay` block, which is also modeled in the diagram of Fig. 2. This block can be represented by an STS component, where a state variable s is needed to store the input at moment i such that it can be used at the next step. We formally define this component as

$$\texttt{UnitDelay} = \texttt{sts}(x, y, s, s = 0, y = s \wedge s' = x).$$

We use first-order logic to define *init_exp* and *trs_exp*. Here *init_exp* is $s = 0$, which initializes the state variable s with the value 0. The *trs_exp* is $y = s \wedge s' = x$, that is at moment i the current state variable s which stores the input x received at moment $i - 1$ is output and its value is updated.

Example 3. As another example, consider again the composite component `Sum` modeling the diagram of Fig. 2. `Sum` could also be defined as an atomic STS component:

$$\texttt{Sum} = \texttt{sts}(x, y, s, s = 0, y = s \wedge s' = s + x).$$

In §5 we will show how we can automatically and symbolically simplify composite component terms such as `fdbk(Add ; UnitDelay ; Split)`, to obtain syntactic representations of atomic components such as the one above.

These examples illustrate systems coming from Simulink models. However, our language is more general, and able to accommodate the description of other systems, such as state machines à la nuXmv [19], or input/output automata [58]. In fact, both `UnitDelay` and `Sum` are deterministic, so they could also be defined as deterministic STS components, as we will see below. Our language can capture non-deterministic systems easily.

Example 4. An example of a non-deterministic STS component is the following:

$$C = \mathtt{sts}(x, y, s, s = 0, x + s \le y).$$

For an input sequence $\sigma_x \in \mathbb{N}^\omega$, C outputs a non-deterministically chosen sequence σ_y such that the transition expression $x+s \le y$ is satisfied. Since there is no formula in the transition expression tackling the next state variable, this is updated also non-deterministically with values from $\mathbb{N}$.

Our language can also capture *non-input-receptive* systems, that is, systems which disallow some input values as *illegal*.

Example 5. For instance, a component performing division, but disallowing division by zero, can be specified as follows:

$$\mathtt{Div} = \mathtt{sts}((x, y), z, (), \mathtt{true}, y \ne 0 \wedge z = \frac{x}{y}).$$

Note that $\mathtt{Div}$ has an empty tuple of state variables, $s = ()$. Such components are called *stateless*, and are introduced in the sequel.

Example 6. Even though RCRS is primarily a discrete-time framework, we have used it to model and verify continuous-time systems such as those modeled in Simulink (see §6). We do this by discretizing time using a time step parameter $\Delta t > 0$ and applying Euler numerical integration. We can model Simulink's *Integrator* block in RCRS as an STS component parameterized by Δt:

$$\mathtt{Integrator}_{\Delta t} \quad = \quad \mathtt{sts}\big(x, y, s, s = 0, y = s \wedge s' = s + x \cdot \Delta t\big)$$

More complex dynamical system blocks can be modeled in a similar fashion. For instance, Simulink's *Transfer Fcn* block, with transfer function

$$\frac{s^2+2}{0.5s^2+2s+1}$$

can be modeled in RCRS as the following STS component parameterized by Δt:

$$\begin{array}{rcl} \mathtt{TransferFcn}_{\Delta t} & = & \mathtt{sts}\big(x, y, (s_1, s_2), s_1 = 0 \wedge s_2 = 0, trs\big) \\ \text{where} \qquad trs & = & (y = -8 \cdot s_1 + 2 \cdot x) \ \wedge \\ & & \quad (s_1' = s_1 + (-4 \cdot s_1 - 2 \cdot s_2 + x) \cdot \Delta t) \ \wedge \\ & & \quad (s_2' = s_2 + s_1 \cdot \Delta t) \end{array}$$

3.2.2 Variable Name Scope

We remark that variable names in the definition of atomic components are *local*. This holds for all atomic components in the language of RCRS (including STS and QLTL components, defined in the sequel). This means that if we replace a variable with another one in an atomic component, then we obtain a semantically equivalent component. For example, the two STS components below are equivalent (the semantical equivalence symbol $\equiv$ will be defined formally in Def. 21, once we define the semantics):

$$\mathtt{sts}((x, y), z, s, s > 0, z > s + x + y) \equiv \mathtt{sts}((u, v), w, t, t > 0, w > t + u + v)$$

3.2.3 Stateless STS Components

A special STS component is one that has no state variables:

Definition 2 (Stateless STS component)**.** *A* stateless STS component *is a tuple*

$$C = \texttt{stateless}(x : \Sigma_x, y : \Sigma_y, \mathit{io\_exp} : \mathbb{B})$$

where x, y are the input and output variables, and $\mathit{io\_exp}$ is a Boolean expression on x and y. Stateless STS components are special cases of general STS components, as defined by the mapping `stateless2sts`*:*

$$\texttt{stateless2sts}(C) = \texttt{sts}(x, y, (), \texttt{true}, \mathit{io\_exp}).$$

Note that the transformation `stateless2sts` is purely *syntactic*. This is also the case for the transformations of other special cases of components described in the sequel.

Example 7. A trivial stateless STS component is the one that simply transfers its input to its output (i.e., a "wire"). We denote such a component by `Id`, and we formalize it as

$$\texttt{Id} = \texttt{stateless}(x, y, y = x).$$

Another simple example is a component with no inputs and a single output, which always outputs a constant value c (of some type). This can be formalized as the following component parameterized by c:

$$\texttt{Const}_c = \texttt{stateless}((), y, y = c).$$

Component `Add` from Fig. 2, which outputs the sum of its two inputs, can be modeled as a stateless STS component:

$$\texttt{Add} = \texttt{stateless}((x, y), z, z = x + y).$$

Component `Split` from Fig. 2 can also be modeled as a stateless STS component:

$$\texttt{Split} = \texttt{stateless}(x, (y, z), y = x \wedge z = x).$$

The `Div` component introduced above is stateless, and therefore can be also specified as follows:

$$\texttt{Div} = \texttt{stateless}\big((x, y), z, y \neq 0 \wedge z = \frac{x}{y}\big).$$

The above examples are not only stateless, but also deterministic. We introduce deterministic STS components next.

3.2.4 Deterministic STS Components

Deterministic STS components are those which, for given current state and input, have at most one output and next state. Syntactically, they are introduced as follows:

Definition 3 (Deterministic STS component)**.** *A* deterministic STS component *is a tuple*

$$\texttt{det}(x : \Sigma_x, s : \Sigma_s, a : \Sigma_s, \mathit{inpt\_exp} : \mathbb{B}, \mathit{next\_exp} : \Sigma_s, \mathit{out\_exp} : \Sigma_y)$$

where x, s are the input and state variables, $a \in \Sigma_s$ is the initial value of the state variable, $\mathit{inpt\_exp}$ is a Boolean expression on s and x defining the legal inputs, $\mathit{next\_exp}$ is an expression of type Σ_s on x and s defining the next state, and $\mathit{out\_exp}$ is an expression of type Σ_y on x and s defining the output. Deterministic STS components are special cases of general STS components, as defined by the mapping `det2sts`*:*

$$\texttt{det2sts}(C) = (x, y, s, s = a, \mathit{inpt\_exp} \wedge s' = \mathit{next\_exp} \wedge y = \mathit{out\_exp})$$

where y is a new variable name (or tuple of new variable names) of type Σ_y.

Note that a deterministic STS component has a separate expression *inpt_exp* to define legal inputs. A separate such expression is not needed for general STS components, where the conditions for legal inputs are part of the expression *trs_exp*. For example, compare the definition of `Div` as a general STS above, and as a stateless deterministic STS below (see §3.2.5).

Example 8. As mentioned above, all three components, `UnitDelay`, `Add`, and `Split` from Fig. 2, as well as `Div` and `Const`, are deterministic. They could therefore be specified in our language as deterministic STS components, instead of general STS components:

$$\begin{aligned}
\texttt{UnitDelay} &= \texttt{det}\big(x, s, 0, \texttt{true}, x, s\big)\\
\texttt{Const}_c &= \texttt{det}\big((), (), (), \texttt{true}, (), c\big)\\
\texttt{Add} &= \texttt{det}\big((x, y), (), (), \texttt{true}, (), x + y\big)\\
\texttt{Split} &= \texttt{det}\big(x, (), (), \texttt{true}, (), (x, x)\big)\\
\texttt{Div} &= \texttt{det}\big((x, y), (), (), y \neq 0, (), \frac{x}{y}\big)
\end{aligned}$$

The component `Sum` modeling the entire system is also deterministic, and could be defined as a deterministic STS component:

$$\texttt{Sum} = \texttt{det}(x, s, 0, \texttt{true}, s + x, s).$$

Note that these alternative specifications for each of those components, although syntactically distinct, will turn out to be semantically equivalent by definition, when we introduce the semantics of our language, in §4.

3.2.5 Stateless Deterministic STS Components

STS components which are both deterministic and stateless can be specified as follows:

Definition 4 (Stateless deterministic STS component)**.** *A* stateless deterministic STS component *is a tuple*

$$C = \texttt{stateless\_det}(x : \Sigma_x, \mathit{inpt\_exp} : \mathbb{B}, \mathit{out\_exp} : \Sigma_y)$$

where x is the input variable, inpt_exp is a Boolean expression on x defining the legal inputs, and out_exp is an expression of type Σ_y on x defining the output. Stateless deterministic STS components are special cases of both deterministic STS components, and of stateless STS components, as defined by the mappings

$$\begin{aligned}
\texttt{stateless\_det2det}(C) &= \texttt{det}(x, (), (), \mathit{inpt\_exp}, (), \mathit{out\_exp}) && (1)\\
\texttt{stateless\_det2stateless}(C) &= \texttt{stateless}(x, y, \mathit{inpt\_exp} \wedge y = \mathit{out\_exp}) && (2)
\end{aligned}$$

where y is a new variable name or a tuple of new variable names.

Example 9. Many of the examples introduced above are both deterministic and stateless. They could be specified as follows:

$$\begin{aligned}
\texttt{Id} &= \texttt{stateless\_det}\big(x, \texttt{true}, x\big)\\
\texttt{Const}_c &= \texttt{stateless\_det}\big((), \texttt{true}, c\big)\\
\texttt{Add} &= \texttt{stateless\_det}\big((x, y), \texttt{true}, x + y\big)\\
\texttt{Split} &= \texttt{stateless\_det}\big(x, \texttt{true}, (x, x)\big)\\
\texttt{Div} &= \texttt{stateless\_det}\big((x, y), y \neq 0, \frac{x}{y}\big)
\end{aligned}$$

3.3 Quantified Linear Temporal Logic Components

Although powerful, STS components have limitations. In particular, they cannot express *liveness* properties [4]. To remedy this, we introduce another type of components, based on Linear Temporal Logic (LTL) [68] and quantified propositional LTL (QPTL) [82, 52], which extends LTL with $\exists$ and $\forall$ quantifiers over propositional variables. In this paper we use *quantified first-order LTL* (which we abbreviate as QLTL). QLTL further extends QPTL with functional and relational symbols over arbitrary domains, quantification of variables over these domains, and a next operator applied to variables.[2] We need this expressive power in order to be able to handle general models (e.g., Simulink) which often use complex arithmetic formulas, and also to be able to translate STS components into semantically equivalent QLTL components (see §5.1).

3.3.1 QLTL

QLTL formulas are generated by the following grammar. We assume a set of constants and functional symbols ($0, 1, \ldots,$ `true`, `false`, $+, \ldots$), a set of predicate symbols ($=, \leq, <, \ldots$), and a set of variable names ($x, y, z, \ldots$).

Definition 5 (Syntax of QLTL)**.** *A QLTL formula φ is defined by the following grammar:*

$$\begin{array}{lcll}
\textit{term} & ::= & x \mid y \mid \ldots \mid & \textit{(variable names)} \\
 & & 0 \mid 1 \mid \ldots \mid \texttt{true} \mid \ldots \mid & \textit{(constants)} \\
 & & \textit{term} + \textit{term} \mid \ldots \mid & \textit{(functional symbol application)} \\
 & & \bigcirc\, \textit{term} & \textit{(next applied to a term)} \\
\varphi & ::= & (\textit{term} = \textit{term}) \mid (\textit{term} \leq \textit{term}) \mid \ldots \mid & \textit{(atomic QLTL formulas)} \\
 & & \neg\varphi \mid & \textit{(negation)} \\
 & & \varphi \vee \psi \mid & \textit{(disjunction)} \\
 & & \varphi \ \mathbf{U}\ \psi \mid & \textit{(until)} \\
 & & \forall x : \varphi & \textit{(forall)}
\end{array}$$

As in standard first order logic, the *bounded variables* of a formula φ are the variables in scope of the universal quantifier $\forall$, and the *free variables* of φ are those that are not bounded. The logic connectives $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ can be expressed with $\neg$ and $\vee$. Quantification is over atomic variables. The existential quantifier $\exists$ can be defined via the universal quantifier usually as $\neg\forall\neg$. The primitive temporal operators are *next for terms* ($\bigcirc$) and *until* ($\mathbf{U}$). As is standard, QLTL formulas are evaluated over infinite traces, and $\varphi\ \mathbf{U}\ \psi$ intuitively means that φ continuously holds until some point in the trace where ψ holds.

Formally, we will define the relation $\sigma \models \varphi$ (σ *satisfies* φ) for a QLTL formula φ over free variables $x, y, \ldots$, and an infinite sequence $\sigma \in \Sigma^\omega$, where $\Sigma = \Sigma_x \times \Sigma_y \times \ldots$, and $\Sigma_x, \Sigma_y, \ldots$ are the types (or *domains*) of variables $x, y, \ldots$. As before we assume that σ can be written as a tuple of sequences $(\sigma_x, \sigma_y, \ldots)$ where $\sigma_x \in \Sigma_x^\omega, \sigma_y \in \Sigma_y^\omega, \ldots$. The semantics of a term t on variables $x, y, \ldots$ is a function from infinite sequences to infinite sequences $\langle\!\langle t \rangle\!\rangle : \Sigma^\omega \to \Sigma_t^\omega$, where $\Sigma = \Sigma_x \times \Sigma_y \times \ldots$, and Σ_t is the type of t. When giving the semantics of terms and formulas we assume that constants, functional symbols, and predicate symbols have the standard semantics. For example, we assume that $+, \leq, \ldots$ on numeric values have the semantics of standard arithmetic.

Definition 6 (Semantics of QLTL)**.** *Let x be a variable, t, t' be terms, φ, ψ be QLTL formulas, P be a*

[2]A logic similar to the one that we use here is presented in [51], however in [51] the next operator can be applied only once to variables, and the logic from [51] uses also past temporal operators.

predicate symbol, f be a functional symbol, c be a constant, and $\sigma \in \Sigma^\omega$ be an infinite sequence. Then:

$$\begin{array}{lcl}
\langle\!\langle x \rangle\!\rangle(\sigma) & := & \sigma_x \\
\langle\!\langle c \rangle\!\rangle(\sigma) & := & (\lambda i : c) \\
\langle\!\langle f(t,t') \rangle\!\rangle(\sigma) & := & (\lambda i : f(\langle\!\langle t \rangle\!\rangle(\sigma)(i), \langle\!\langle t' \rangle\!\rangle(\sigma)(i))) \\
\langle\!\langle \bigcirc t \rangle\!\rangle(\sigma) & := & \langle\!\langle t \rangle\!\rangle(\sigma^1) \\
\sigma \models P(t,t') & := & P(\langle\!\langle t \rangle\!\rangle(\sigma)(0), \langle\!\langle t' \rangle\!\rangle(\sigma)(0)) \\
\sigma \models \neg\varphi & := & \neg\,(\sigma \models \varphi) \\
\sigma \models \varphi \vee \psi & := & \sigma \models \varphi \vee \sigma \models \psi \\
\sigma \models \varphi\ \mathbf{U}\ \psi & := & (\exists n \geq 0 : (\forall\ 0 \leq i < n : \sigma^i \models \varphi) \wedge \sigma^n \models \psi) \\
\sigma \models (\forall x : \varphi) & := & (\forall \sigma'_x \in \Sigma_x^\omega : (\sigma[x := \sigma'_x]) \models \varphi)
\end{array}$$

where $\sigma[x := \sigma'_x]$ denotes the trace $\sigma' \in \Sigma^\omega$ obtained by replacing σ_x in σ by σ'_x.

Intuitively, the semantics of variable x is a function which returns the sequence σ_x corresponding to x, given a sequence σ. The semantics of constant c is the constant function which returns the sequence which has c at every step. The semantics of a function f applies f at every step of a sequence σ. Given sequence σ, the semantics of $\bigcirc t$ applies the semantics of t, $\langle\!\langle t \rangle\!\rangle$, to the sequence σ^1, that is, to σ starting from position 1 instead of position 0. The satisfaction relation between sequences and formulas is denoted $\models$. A sequence σ satisfies a predicate P if it satisfies P at position 0. σ satisfies $\neg\varphi$ if it does not satisfy φ. σ satisfies $\varphi \vee \psi$ if it satisfies either φ or ψ. σ satisfies $\varphi\ \mathbf{U}\ \psi$ if ψ is satisfied at some point, and until that point φ is continuously satisfied. Finally, σ satisfies $\forall x : \varphi$ if $\sigma[x := \sigma'_x]$ satisfies φ, for any σ'_x.

Other temporal operators can be defined as follows. *Eventually* ($\mathbf{F}\,\varphi = \texttt{true}\ \mathbf{U}\ \varphi$) states that φ must hold in some future step. *Always* ($\mathbf{G}\,\varphi = \neg\mathbf{F}\,\neg\varphi$) states that φ must hold at all steps. The next operator for formulas $\mathbf{X}$ can be defined using the next operator for terms $\bigcirc$. The formula $\mathbf{X}\,\varphi$ is obtained by replacing all occurrences of the free variables in φ by their next versions (i.e., x is replaced by $\bigcirc x$, y by $\bigcirc y$, etc.). For example the propositional LTL formula $\mathbf{X}\,(x \wedge \mathbf{X}\,y \Rightarrow \mathbf{G}\,z)$ can be expressed as

$$(\bigcirc x = \texttt{true} \wedge (\bigcirc\ \bigcirc\ y = \texttt{true}) \Rightarrow \mathbf{G}\,(\bigcirc z = \texttt{true})).$$

We additionally introduce the operator: $\varphi\,\mathbf{L}\,\psi := \neg(\varphi\ \mathbf{U}\ \neg\psi)$. Intuitively, $\varphi\,\mathbf{L}\,\psi$ holds if whenever φ holds continuously up to some step $n-1$, ψ must hold at step n. Later we will use the operator $\mathbf{L}$ to transform a STS component into a QLTL component.

Lemma 1. *The semantics of $\varphi\,\mathbf{L}\,\psi$ is given by*

$$\sigma \models \varphi\,\mathbf{L}\,\psi \ = \ (\forall n \geq 0 : (\forall 0 \leq i < n : \sigma^i \models \varphi) \Longrightarrow \sigma^n \models \psi)$$

Two QLTL formulas φ and ψ are semantically equivalent, denoted $\varphi \iff \psi$, if

$$\forall \sigma : (\sigma \models \varphi) \iff (\sigma \models \psi).$$

Lemma 2. *Let φ be a QLTL formula. Then:*

1. $(\exists x : \mathbf{G}\,\varphi) \iff \mathbf{G}\,(\exists x : \varphi)$ *when φ does not contain temporal operators.*
2. $\varphi\,\mathbf{L}\,\varphi \iff \mathbf{G}\,\varphi$
3. $\texttt{true}\,\mathbf{L}\,\varphi \iff \mathbf{G}\,\varphi$
4. $\varphi\,\mathbf{L}\,\texttt{true} \iff \texttt{true}$
5. $\varphi\,\mathbf{L}\,\texttt{false} \iff \texttt{false}$
6. $\forall y : (\varphi\,\mathbf{L}\,\psi) \iff (\exists y : \varphi)\,\mathbf{L}\,\psi$, *when φ does not contain temporal operators and y is not free in ψ.*

7. $(\mathbf{G}\ (\varphi \wedge \psi)) = (\mathbf{G}\ \varphi) \wedge (\mathbf{G}\ \psi)$

The proof of the above result, as well as of most results that follow, is omitted. All omitted proofs have been formalized and proved in the Isabelle proof assistant, and are available as part of the public distribution of RCRS from http://rcrs.gitlab.io/. In particular, the results contained in this paper can be accessed from the theory `RCRS_Overview.thy` – either directly in that file or via references to the other RCRS files.

Example 10. Using QLTL we can express *safety*, as well as *liveness requirements*. Informally, a safety requirement expresses that something bad never happens. An example is the formula

$$\texttt{thermostat} = \mathbf{G}\,(180° \leq t \wedge t \leq 220°),$$

which states that the thermostat-controlled temperature t stays always between 180° and 220°.

A liveness requirement informally says that something good eventually happens. An example is the formula $\mathbf{F}\,(t > 200°)$ stating that the temperature t is eventually over 200°.

A more complex example is a formula modeling an oven that starts increasing the temperature from an initial value of 20° until it reaches 180°, and then keeps it between 180° and 220°.

$$\texttt{oven} = (t = 20° \wedge ((t < \bigcirc\, t \wedge t < 180°)\ \mathbf{U}\ \texttt{thermostat})).$$

In this example the formula $t < \bigcirc\, t$ specifies that the temperature increases from some point to the next.

3.3.2 QLTL Components

A QLTL component is an atomic component where the input-output behavior is specified by a QLTL formula:

Definition 7 (QLTL component)**.** *A* QLTL component *is a tuple* $\texttt{qltl}(x : \Sigma_x, y : \Sigma_y, \varphi)$*, where* x, y *are input and output variables (or tuples of variables) of types* Σ_x, Σ_y*, and* φ *is a QLTL formula over* x *and* y*.*

Intuitively a QLTL component $C = \texttt{qltl}(x, y, \varphi)$ represents a system that takes as input an infinite sequence $\sigma_x \in \Sigma_x^\omega$ and produces as output an infinite sequence $\sigma_y \in \Sigma_y^\omega$ such that $(\sigma_x, \sigma_y) \models \varphi$. If there is no σ_y such that $(\sigma_x, \sigma_y) \models \varphi$ is true, then input σ_x is illegal for C, i.e., C is not input-receptive. There could be many possible σ_y for a single σ_x, in which case the system is non-deterministic.

Example 11. As a simple example, we can model the oven as a QLTL component with no input variables and the temperature as the only output variable:

$$\texttt{qltl}((), t, \texttt{oven})$$

3.4 Well Formed Composite Components

Not all composite components generated by the grammar introduced in §3.1 are *well formed*. Two components C and C' can be composed in series only if the number of outputs of C matches the number of inputs of C', and in addition the input types of C' are the same as the corresponding output types of C. Also, `fdbk` can be applied to a component C if the type of the first output of C is the same as the type of its first input. Formally, for every component C we define below $\Sigma_{in}(C)$ - the *input type* of C, $\Sigma_{out}(C)$ - the *output type* of C, and $\mathbf{wf}(C)$ - the *well-formedness* of C, by induction on the structure of C. In the definitions below, both n and m are natural numbers. Recall that the empty Cartesian product is Unit, so that if $n = 0$ then $X_1 \times \cdots \times X_n$ denotes the Unit type.

$$
\begin{array}{lcl}
\Sigma_{in}(\texttt{sts}(x:\Sigma_x, y:\Sigma_y, s:\Sigma_s, init, trs)) & = & \Sigma_x \\
\Sigma_{in}(\texttt{stateless}(x:\Sigma_x, y:\Sigma_y, trs)) & = & \Sigma_x \\
\Sigma_{in}(\texttt{det}(x:\Sigma_x, s:\Sigma_s, a, inpt, next, out:\Sigma_y)) & = & \Sigma_x \\
\Sigma_{in}(\texttt{stateless\_det}(x:\Sigma_x, inpt, out:\Sigma_y)) & = & \Sigma_x \\
\Sigma_{in}(\texttt{qltl}(x:\Sigma_x, y:\Sigma_y, \varphi)) & = & \Sigma_x \\
\Sigma_{in}(C\ ;\ C') & = & \Sigma_{in}(C) \\
\Sigma_{in}(C \parallel C') & = & \Sigma_{in}(C) \times \Sigma_{in}(C') \\
\Sigma_{in}(\texttt{fdbk}(C)) & = & X_2 \times \cdots \times X_n \text{ provided } \Sigma_{in}(C) = X_1 \times \cdots \times X_n \\
 & & \text{for some } \ n \geq 1
\end{array}
$$

That is, the input type of an atomic component is the type Σ_x of input variable x. The input type of serial composite component $C\,;C'$ is the input type of the "upstream" component C. The input type of parallel composite component $C \parallel C'$ is the Cartesian product of the input types of C and C'. Assuming that C has $n \geq 1$ inputs with types $X_1, ..., X_n$, the input type of feedback composite component $\texttt{fdbk}(C)$ is the Cartesian product $X_2 \times \cdots \times X_n$, i.e., the first input is omitted. If $n = 1$ then the input type of $\texttt{fdbk}(C)$ is Unit.

$$
\begin{array}{lcl}
\Sigma_{out}(\texttt{sts}(x:\Sigma_x, y:\Sigma_y, s:\Sigma_s, init, trs)) & = & \Sigma_y \\
\Sigma_{out}(\texttt{stateless}(x:\Sigma_x, y:\Sigma_y, trs)) & = & \Sigma_y \\
\Sigma_{out}(\texttt{det}(x:\Sigma_x, s:\Sigma_s, a, inpt, next, out:\Sigma_y)) & = & \Sigma_y \\
\Sigma_{out}(\texttt{stateless\_det}(x:\Sigma_x, inpt, out:\Sigma_y)) & = & \Sigma_y \\
\Sigma_{out}(\texttt{qltl}(x:\Sigma_x, y:\Sigma_y, \varphi)) & = & \Sigma_y \\
\Sigma_{out}(C\ ;\ C') & = & \Sigma_{out}(C') \\
\Sigma_{out}(C \parallel C') & = & \Sigma_{out}(C) \times \Sigma_{out}(C') \\
\Sigma_{out}(\texttt{fdbk}(C)) & = & Y_2 \times \cdots \times Y_n \text{ provided } \Sigma_{out}(C) = Y_1 \times \cdots \times Y_n \\
 & & \text{for some } n \geq 1
\end{array}
$$

That is, the output type of an atomic component is the type Σ_y of output variable y. The output type of serial composite component $C\,;C'$ is the output type of the "downstream" component C'. The output type of parallel composite component $C \parallel C'$ is the Cartesian product of the output types of C and C'. Assuming that C has $n \geq 1$ outputs with types $Y_1, ..., Y_n$, the output type of feedback composite component $\texttt{fdbk}(C)$ is the Cartesian product $Y_2 \times \cdots \times Y_n$, i.e., the first output is omitted. If $n = 1$ then the output type of $\texttt{fdbk}(C)$ is Unit.

$$
\begin{array}{lcl}
\mathbf{wf}(\texttt{sts}(x, y, s, init, trs)) & = & true \\
\mathbf{wf}(\texttt{stateless}(x, y, trs)) & = & true \\
\mathbf{wf}(\texttt{det}(x, s, a, inpt, next, out)) & = & true \\
\mathbf{wf}(\texttt{stateless\_det}(x, inpt, out)) & = & true \\
\mathbf{wf}(\texttt{qltl}(x, y, \varphi)) & = & true \\
\mathbf{wf}(C\ ;\ C') & = & \mathbf{wf}(C) \wedge \mathbf{wf}(C') \wedge \Sigma_{out}(C) = \Sigma_{in}(C') \\
\mathbf{wf}(C \parallel C') & = & \mathbf{wf}(C) \wedge \mathbf{wf}(C') \\
\mathbf{wf}(\texttt{fdbk}(C)) & = & \mathbf{wf}(C) \wedge \Sigma_{in}(C) = X \times X_1 \cdots \times X_n \\
 & & \wedge\, \Sigma_{out}(C) = X \times Y_1 \cdots \times Y_m, \text{ for some } n, m \geq 0.
\end{array}
$$

That is, all atomic components are by definition well-formed. A serial composite component $C\,;C'$ is well-formed iff both its subcomponents C and C' are well-formed, and the output type of C is equal to the input type of C'. A parallel composite component $C \parallel C'$ is well-formed iff both its subcomponents C and C' are well-formed. A feedback composite component $\texttt{fdbk}(C)$ is well-formed iff C is well-formed and the type of the first output of C is equal to the type of its first input.

Atomic components are by definition well-formed. The composite components considered in the sequel are required to be well-formed too.

We note that the above well-formedness conditions are not restrictive. Components that do not have matching inputs and outputs can still be composed by adding appropriate *switching* components which reorder inputs, duplicate inputs, and so on. An example of such a component is the component `Split`, introduced earlier.

Example 12. As another example, consider the diagram in Fig. 4.

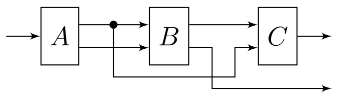


Figure 4: Another block diagram.

This diagram can be expressed in our language as the composite component:

$$A\,;\texttt{Switch1}\,;(B \parallel \texttt{Id})\,;\texttt{Switch2}\,;(C \parallel \texttt{Id})$$

where

$$\begin{array}{rcl}\texttt{Switch1} & = & \texttt{stateless\_det}((x,y),\texttt{true},(x,y,x))\\ \texttt{Switch2} & = & \texttt{stateless\_det}((u,v,x),\texttt{true},(u,x,v))\end{array}$$

Component `Switch1` takes as input the two outputs x and y of A and outputs three outputs by replicating x as its third output. This triplet of outputs (x, y, x) is fed as input to the parallel composite component $B \parallel$ `Id`. The first two outputs, x and y, are fed as inputs to the corresponding two inputs of B. The third output, which is also equal to x, if fed as input to the identity component `Id`, which simply acts as a "wire" transferring its input to its output. Therefore, if we let u and v be the two outputs of B, the output of $B \parallel$ `Id` is the triplet (u, v, x). Now, we want to feed u and x as the two inputs to C. To do that, we use `Switch2`, which swaps the order of v and x, so that its output triplet becomes (u, x, v). Now we can feed the latter triplet into $C \parallel$ `Id`.

This completes the presentation of the syntax of RCRS. In the section that follows we define the formal semantics of RCRS.

4 Semantics

In RCRS, the semantics of components is defined in terms of *monotonic property transformers* (MPTs). This is inspired by classical refinement calculus [12], where the semantics of sequential programs is defined in terms of monotonic *predicate* transformers [31]. Predicate transformers are functions that transform sets of *post*-states (states reached after the program executes) into sets of *pre*-states (states from which the program begins). Property transformers map sets of *output traces* (that a component produces) into sets of *input traces* (that a component consumes).

In this section we define MPTs formally, and introduce some basic operations on them, which are necessary for giving the semantics of components. We also introduce subclasses of MPTs which are easier to work with in practice, especially in terms of symbolic computation, and we also study their closure properties with respect to the operators. The definitions of some of these operations (e.g., product and fusion) are simple extensions of the corresponding operations on predicate transformers [12, 11]. Several of the subclasses and closure properties that we examine here are also extensions to the MPT context of known subclasses of predicate transformers and their closure properties in the refinement calculus theory [12]. Other operations, in particular those related to feedback, are new (§4.1.4). The definition of component semantics is also new (§4.3).

4.1 Monotonic Property Transformers

A property transformer is a function $S : (\Sigma_y^\omega \to \mathbb{B}) \to (\Sigma_x^\omega \to \mathbb{B})$, where Σ_x, Σ_y are input and output types of the component in question. Note that x is the input and y is the output. A property transformer has a weakest precondition interpretation: it is applied to a set of output traces $Q \subseteq \Sigma_y^\omega$, and returns a set of input traces $P \subseteq \Sigma_x^\omega$, such that all traces in P are legal and, when fed to the component, are guaranteed to produce only traces in Q as output.

Interpreting properties as sets, monotonicity of property transformers simply means that these functions are monotonic with respect to set inclusion. That is, S is *monotonic* if for any two properties q, q', if $q \subseteq q'$ then $S(q) \subseteq S(q')$.

Similar to the domain or precondition of a relation, for an MPT S we define its set of *legal input traces* as $\mathsf{legal}(S) = S(\top)$, where $\top$ is the greatest predicate on traces. Note that, because of monotonicity, and the fact that $q \subseteq \top$ holds for any property q, we have that $S(q) \subseteq \mathsf{legal}(S)$ for all q. This justifies the definition of $\mathsf{legal}(S)$ as a "maximal" set of input traces for which a system does not fail, assuming no restrictions on the post-condition. An MPT S is said to be *input-receptive* if $\mathsf{legal}(S) = \top$.

4.1.1 Some Commonly Used MPTs

Definition 8 (Skip)**.** Skip *is defined to be the MPT such that for all* q, $\mathsf{Skip}(q) = q$.

Skip models the identity function, i.e., the system that accepts all input traces and simply transfers them unchanged to the output (this will become more clear when we express Skip in terms of assert or update transformers, below). Note that Skip is different from `Id`, defined above, although the two are strongly related: `Id` is a component, i.e., a syntactic object, while Skip is an MPT, i.e., a semantic object. As we shall see in §4.3, the semantics of `Id` is defined as Skip.

Definition 9 (Fail)**.** Fail *is defined to be the MPT such that for all* q, $\mathsf{Fail}(q) = \bot$.

Recall that $\bot$ is the predicate that returns false for any input. Thus, viewed as a set, $\bot$ is the empty set. Consequently, Fail can be seen to model a system which rejects all inputs, i.e., a system such that for any output property q, there are no input traces that can produce an output trace in q.

Definition 10 (Assert)**.** *Let* $p \in \Sigma^\omega \to \mathbb{B}$ *be a property. The* assert property transformer $\{p\} : (\Sigma^\omega \to \mathbb{B}) \to (\Sigma^\omega \to \mathbb{B})$ *is defined by*

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

The assert transformer $\{p\}$ can be seen as modeling a system which accepts all input traces that satisfy p, and rejects all others. For all the traces that it accepts, the system simply transfers them, i.e., it behaves as the identity function.

To express MPTs such as assert transformers syntactically, let us introduce some notation. First, we can use lambda notation for predicates, as in $\lambda(\sigma, \sigma') : (\sigma = \sigma')$ for some predicate $p : \Sigma^\omega \to \Sigma^\omega \to \mathbb{B}$ which returns true whenever it receives two equal traces. Then, instead of writing $\{\lambda(\sigma, \sigma') : (\sigma = \sigma')\}$ for the corresponding assert transformer $\{p\}$, we will use the slightly lighter notation $\{\sigma, \sigma' \mid \sigma = \sigma'\}$.

Definition 11 (Demonic update)**.** *Let* $r : \Sigma_x^\omega \to \Sigma_y^\omega \to \mathbb{B}$ *be a relation. The* demonic update property transformer $[r] : (\Sigma_y^\omega \to \mathbb{B}) \to (\Sigma_x^\omega \to \mathbb{B})$ *is defined by*

$$[r](q) = \{\sigma \mid \forall \sigma' : r(\sigma)(\sigma') \Rightarrow \sigma' \in q\}.$$

That is, $[r](q)$ contains all input traces σ which are guaranteed to result into an output trace in q when fed into the (generally non-deterministic) input-output relation r. The term "demonic update" comes from the refinement calculus literature [12].

For a demonic update $[r]$, if for some input trace σ there is no trace σ' such that $r(\sigma)(\sigma')$, then for all properties q (including $q = \emptyset$) we have $\sigma \in [r](q)$, meaning that the property transformer $[r]$ establishes any post-property when fed with input trace σ. This behavior cannot be implemented and we call it *magical.* A more detailed discussion about this case is given in Section 4.2.2.

Similarly to assert, we introduce a lightweight notation for the demonic update. If r is an expression in σ and σ', then $[\sigma \rightsquigarrow \sigma' \mid r] = [\lambda(\sigma, \sigma') : r]$. For example, $[\sigma_x, \sigma_y \rightsquigarrow \sigma_z \mid \forall i : \sigma_z(i) = \sigma_x(i) + \sigma_y(i)]$ is the system which produces as output the sequence $\sigma_z = (\lambda i : \sigma_x(i) + \sigma_y(i))$, where σ_x and σ_y are the input sequences. If e is an expression in σ, then $[\sigma \rightsquigarrow e]$ is defined to be $[\sigma \rightsquigarrow \sigma' \mid \sigma' = e]$, where σ' is a new variable different from σ and which does not occur free in e. For example, $[\sigma \rightsquigarrow (\lambda i : \sigma(i)+1)] = [\sigma \rightsquigarrow \sigma' \mid \sigma' = (\lambda i : \sigma(i)+1)]$.

The following lemma states that Skip can be defined as an assert transformer, or as a demonic update transformer.

Lemma 3. $\mathsf{Skip} = [\sigma \rightsquigarrow \sigma' \mid \sigma = \sigma'] = \{\top\} = \{\sigma \mid \mathsf{true}\}$.

In general Skip, Fail, and other property transformers are polymorphic with respect to their input and output types. In Skip the input and output types must be the same. Fail, on the other hand, may have an input type and a different output type.

Definition 12 (Angelic update)**.** *Let $r : \Sigma_x^\omega \to \Sigma_y^\omega \to \mathbb{B}$ be a relation. The* angelic update property transformer *$\{r\} : (\Sigma_y^\omega \to \mathbb{B}) \to (\Sigma_x^\omega \to \mathbb{B})$ is defined by*

$$\{r\}(q) = \{\sigma \mid \exists \sigma' : r(\sigma)(\sigma') \wedge \sigma' \in q\}.$$

An input sequence σ is in $\{r\}(q)$ if there exists an output sequence σ' such that $r(\sigma)(\sigma')$ and $\sigma' \in q$. Notice the duality between the angelic and demonic update transformers. Consider, for example, a relation $r = \{(\sigma, \sigma'), (\sigma, \sigma'')\}$. If $q = \{\sigma', \sigma''\}$, then $\{r\}(q) = [r](q) = \{\sigma\}$. If $q = \{\sigma'\}$ then $\{r\}(q) = \{\sigma\}$, while $[r](q) = \emptyset$.

We use a lightweight notation for the angelic update transformer, similar to the one for demonic update. If r is an expression in σ and σ', then $\{\sigma \rightsquigarrow \sigma' \mid r\} = \{\lambda(\sigma, \sigma') : r\}$.

Note that although the notations for the assert property transformer $\{p\}$ and the angelic update property transformer $\{r\}$ are similar, the two types of transformers differ because r is a relation whereas p is a property. The following lemma states that assert is a special case of angelic update.

Lemma 4. *Assert is a particular case of angelic update: $\{p\} = \{\sigma \rightsquigarrow \sigma' \mid p(\sigma) \wedge \sigma' = \sigma\}$.*

4.1.2 Relational MPTs

Monotonic property transformers are a very rich and powerful class of semantic objects. In practice, the systems that we deal with often fall into restricted subclasses of MPTs, which are easier to represent syntactically and manipulate symbolically. We introduce one of these subclasses here. MPT subclasses are further discussed in Section 4.2.

Definition 13 (Relational property transformers)**.** *A* relational property transformer *(RPT) S is an MPT of the form $\{p\} \circ [r]$. We call p the* precondition *of S and r the* input-output relation *of S.*

Intuitively, the assert part $\{p\}$ of the RPT imposes restrictions on the legal inputs, whereas the update $[r]$ specifies the generally non-deterministic set of possible outputs for each input. Relational property transformers correspond to *conjunctive* transformers [12]. A transformer S is conjunctive if it satisfies $S(\bigcap_{i\in I} q_i) = \bigcap_{i\in I} S(q_i)$ for all $(q_i)_{i\in I}$ and $I \neq \emptyset$.

Fail, Skip, any assert transformer $\{p\}$, and any demonic update transformer $[r]$, are RPTs. Indeed, Fail can be written as $\{\sigma \mid \mathtt{false}\} \circ [\sigma \rightsquigarrow \sigma' \mid \mathtt{true}]$. Skip can be written as $\{\sigma \mid \mathtt{true}\} \circ [\sigma \rightsquigarrow \sigma]$. The assert transformer $\{p\}$ can be written as the RPT $\{p\} \circ [\sigma \rightsquigarrow \sigma]$. Finally, the demonic update transformer $[r]$ can be written as the RPT $\{\sigma \mid \mathtt{true}\} \circ [r]$. Angelic update transformers are generally not RPTs: the angelic update transformer $\{\sigma \rightsquigarrow \sigma' \mid \mathtt{true}\}$ is not an RPT, as it is not conjunctive.

Example 13. Suppose we wish to specify a system that performs division. Here are three possible ways to represent this system with RPTs:

$$S_1 = \{\top\} \circ [\sigma_x, \sigma_y \rightsquigarrow \sigma_z \mid \forall i : \sigma_y(i) \neq 0 \wedge \sigma_z(i) = \frac{\sigma_x(i)}{\sigma_y(i)}]$$

$$S_2 = \{\top\} \circ [\sigma_x, \sigma_y \rightsquigarrow \sigma_z \mid (\forall i : \sigma_y(i) \neq 0 \Rightarrow \sigma_z(i) = \frac{\sigma_x(i)}{\sigma_y(i)})]$$

$$S_3 = \{\sigma_x, \sigma_y \mid \forall i : \sigma_y(i) \neq 0\} \circ [\sigma_x, \sigma_y \rightsquigarrow \sigma_z \mid \sigma_z(i) = \frac{\sigma_x(i)}{\sigma_y(i)}]$$

Although S_1, S_2, and S_3 are all relational, they are not equivalent transformers. S_1 and S_2 are input-receptive: they accept all input traces. S_1 behaves miraculously if the input $\sigma_y(i)$ is 0 at some step i. If at some step i the input $\sigma_y(i)$ is 0, then the output $\sigma_z(i)$ of transformer S_2 is arbitrary (non-deterministic). In contrast, S_3 is non-input-receptive as it accepts only those traces σ_y that are guaranteed to be non-zero at every step, i.e., those that satisfy the condition $\forall i : \sigma_y(i) \neq 0$.

4.1.3 Operators on MPTs: Function Composition, Product, and Fusion

As we shall see in §4.3, the semantics of composition operators in the language of components will be defined by the corresponding composition operators on MPTs. We now introduce the latter operators on MPTs. First, we begin by the operators that have been known in the literature, and are recalled here. In §4.1.4 we introduce some novel operators explicitly designed in order to handle feedback composition.

Serial composition of MPTs (and property transformers in general) is simply function composition:

Definition 14. *Let $S : (\Sigma_y^\omega \to \mathbb{B}) \to (\Sigma_x^\omega \to \mathbb{B})$ and $T : (\Sigma_z^\omega \to \mathbb{B}) \to (\Sigma_y^\omega \to \mathbb{B})$ be two property transformers. Then $S \circ T : (\Sigma_z^\omega \to \mathbb{B}) \to (\Sigma_x^\omega \to \mathbb{B})$, is the function composition of S and T, i.e., $\forall q : (S \circ T)(q) = S(T(q))$.*

Note that serial composition preserves monotonicity, so that if S and T are MPTs, then $S \circ T$ is also an MPT. Also note that Skip is the neutral element for serial composition, i.e., $S \circ \mathsf{Skip} = \mathsf{Skip} \circ S = S$.

The following lemma shows how the serial composition of two demonic updates is also a demonic update.

Lemma 5. $[r] \circ [r'] = [r \circ r']$

To express parallel composition of components, we need a product operation on property transformers. We define such an operation below. Similar operations for predicate transformers have been proposed in [11].

Definition 15 (Product). *Let $S : (\Sigma_y^\omega \to \mathbb{B}) \to (\Sigma_x^\omega \to \mathbb{B})$ and $T : (\Sigma_v^\omega \to \mathbb{B}) \to (\Sigma_u^\omega \to \mathbb{B})$. The* product *of S and T, denoted $S \otimes T : (\Sigma_y^\omega \times \Sigma_v^\omega \to \mathbb{B}) \to (\Sigma_x^\omega \times \Sigma_u^\omega \to \mathbb{B})$, is given by*

$$(S \otimes T)(q) = \{(\sigma, \sigma') \mid \exists p : \Sigma_y^\omega \to \mathbb{B}, p' : \Sigma_v^\omega \to \mathbb{B} : p \times p' \subseteq q \wedge \sigma \in S(p) \wedge \sigma' \in T(p')\}$$

where $(p \times p')(\sigma_y, \sigma_v) = p(\sigma_y) \wedge p'(\sigma_v)$.

The product $S \otimes T$ models the simultaneous execution of S and T. This will become more clear later in Theorem 2, as the product of MPTs based on predicates and relations (i.e., RPTs) can be expressed based on the products of the predicates and the product of the relations:

$$(\{p\} \circ [r]) \otimes (\{p'\} \circ [r']) = \{p \times p'\} \circ [r \times r']$$

where the product of relations $r \times r'$ is defined similarly to the product on predicates.

Lemma 6. *For arbitrary S and T, $S \otimes T$ is monotonic.*

The neutral element for the product composition is the Skip MPT that has Unit as input and output type.

In order to define a feedback operation on MPTs, we first define two auxiliary operations: *Fusion* and *IterateOmega*. Fusion is an extension of a similar operator introduced previously for predicate transformers in [11]. IterateOmega is a novel operator introduced in the sequel.

Intuitively, the fusion operator, when applied to two MPTs S and T, rejects all inputs that are rejected by either S or T, and for a legal input of S and T it nondeterministically chooses an output that could be chosen by both S and T. If for some legal input σ of S and T there is no common output of S and T, then the fusion of S and T behaves magically when executed with input σ. Below we define the fusion operator not for just two MPTs but more generally for an arbitrary family of MPTs, S_i, $i \in I$.

Definition 16 (Fusion)**.** *If $S = \{S_i\}_{i\in I}$, $S_i : (\Sigma_y^\omega \to \mathbb{B}) \to (\Sigma_x^\omega \to \mathbb{B})$ is a collection of MPTs, then the* fusion *of S is the MPT $\mathsf{Fusion}_{i\in I}(S_i) : (\Sigma_y^\omega \to \mathbb{B}) \to (\Sigma_x^\omega \to \mathbb{B})$ defined by*

$$(\mathsf{Fusion}_{i\in I}(S_i))(q) = \{\sigma \mid \exists w : I \to \Sigma_y^\omega \to \mathbb{B} : \bigcap_{i\in I} w(i) \subseteq q \wedge \sigma \in \bigcap_{i\in I} S_i(w(i))\}$$

In particular, the fusion of two MPTs S and T is the MPT:

$$\mathsf{Fusion}(S,T)(q) = \{\sigma \mid \exists w_1, w_2 : \Sigma_y^\omega \to \mathbb{B} : w_1 \cap w_2 \subseteq q \wedge \sigma \in S(w_1) \cap T(w_2)\}.$$

Similarly to the product operator, the fusion operator of two MPTs S and T models the simultaneous execution of S and T, but on the same state. If S and T perform choices from a starting sequence σ, then $\mathsf{Fusion}(S,T)$ will output only sequences that can be chosen by both S and T starting from σ. The set of legal input traces of $\mathsf{Fusion}(S,T)$ is the intersection of the legal traces of S and legal traces of T. These facts can be derived from Lemma 7 which describes the effect of Fusion on RPTs:

Lemma 7. *For $I \neq \emptyset$ we have*

$$\mathsf{Fusion}_{i\in I}(\{p_i\} \circ [r_i]) = \{\bigcap_{i\in I} p_i\} \circ [\bigcap_{i\in I} r_i].$$

4.1.4 Operators on MPTs: Iteration and Feedback

In this section we introduce some novel operators, including the IterateOmega operator, used in the semantical definition of feedback. In order to explain the intuition behind IterateOmega we first illustrate it on an example. Consider the IncDelay system shown in Figure 5. Assuming 0 is the initial value of the Unit Delay block, IncDelay maps input sequence $x_0, x_1, \ldots$ into output sequence $0, x_0+1, x_1+1, \ldots$. Also consider the system `fdbk`(IncDelay) where the output y is connected to the input x. Intuitively, `fdbk`(IncDelay) should output the sequence $0, 1, 2, \cdots$.

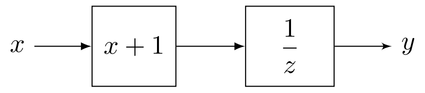


Figure 5: IncDelay.

Let us now calculate $\mathsf{IncDelay}^2 = \mathsf{IncDelay} \circ \mathsf{IncDelay}$, $\mathsf{IncDelay}^3 = \mathsf{IncDelay} \circ \mathsf{IncDelay} \circ \mathsf{IncDelay}$, etc. $\mathsf{IncDelay}^2$ maps $x_0, x_1, \cdots$ into $0, 1, x_0+2, x_1+2, \cdots$. $\mathsf{IncDelay}^3$ maps $x_0, x_1, \cdots$ into $0, 1, 2, x_0+3, x_1+3, \cdots$. From this we observe that the first n elements of the output of $\mathsf{IncDelay}^n$ are the same as the first n elements of the output of `fdbk`(IncDelay). That is, the output of $\mathsf{IncDelay}^n$ *converges* to the output of `fdbk`(IncDelay) as $n \to \omega$. This mechanism is captured in the formal definition of IterateOmega. In general, IterateOmega calculates the feedback of a system with no additional inputs and outputs: the entire output of S is connected in feedback to the entire input of S.

Definition 17 (IterateOmega)**.**

$$\mathsf{IterateOmega}(S) = \mathsf{Fusion}_{n\in\mathbb{N}}(S^n \circ [\sigma \leadsto \sigma' \mid \forall i : i+1 < n \Rightarrow \sigma_i = \sigma'_i])$$

The IterateOmega(S) *is the fusion of the following transformers:*

$$\begin{aligned}
T_0 &= [\sigma \leadsto \sigma' \mid \mathtt{true}] \\
T_1 &= S \circ [\sigma \leadsto \sigma' \mid \mathtt{true}] \\
T_2 &= S^2 \circ [\sigma \leadsto \sigma' \mid \sigma_0 = \sigma'_0] \\
T_3 &= S^3 \circ [\sigma \leadsto \sigma' \mid \sigma_0 = \sigma'_0 \wedge \sigma_1 = \sigma'_1] \\
&\ldots
\end{aligned}$$

The operator IterateOmega applies to an MPT S with the same type Σ for the input and output and its result is a MPT that has again the same input and output types. The idea of IterateOmega is that it applies the serial compositions of S to itself iteratively. The composition S^2 when started on a sequence σ will result into a sequence with the first component σ'_0. The assumption is that, for the feedback to work, the first component stabilizes in subsequent serial compositions. For example the first component of S^3 will also be σ'_0, and the second element of the result of S^3 also stabilizes in subsequent compositions. In general we expect that S^n, when started from a trace σ, produces a trace σ' where $\sigma'_0, \ldots \sigma'_{n-1}$ are the correct outputs of the IterateOmega(S). To be able to construct the final trace of IterateOmega(S), we use the composition $T_n = S^n \circ [\sigma \leadsto \sigma' \mid \forall i : i+1 < n \Rightarrow \sigma_i = \sigma'_i]$ which sets all components i of the output with $i + 1 \geq n$ to arbitrary values. For input trace σ we assume that possible resulting traces for $T_0, T_1, \ldots$ are

$$\begin{array}{lllllll}
T_0: & \sim & \sim & \sim & \sim & \sim & \ldots \\
T_1: & \sim & \sim & \sim & \sim & \sim & \ldots \\
T_2: & \sigma'_0 & \sim & \sim & \sim & \sim & \ldots \\
T_3: & \sigma'_0 & \sigma'_1 & \sim & \sim & \sim & \ldots \\
T_4: & \sigma'_0 & \sigma'_1 & \sigma'_2 & \sim & \sim & \ldots \\
\ldots
\end{array}$$

where $\sim$ expresses any possible value. In this situation, the trace $\sigma' = \sigma'_0\sigma'_1 \ldots$ is a possible output trace of the fusion of all T_i, i.e. a possible output trace of IterateOmega(S).

The *feedback* operator consists of connecting the first output of an MPT S with its first input. Formally, feedback is defined as follows.

Definition 18 (Feedback)**.** *Let* $S : (\Sigma_u^\omega \times \Sigma_y^\omega \to \mathbb{B}) \to (\Sigma_u^\omega \times \Sigma_x^\omega \to \mathbb{B})$ *be an MPT. The* feedback operator *on* S*, denoted* Feedback(S)*, is given by the MPT*

$$\begin{aligned}
\mathsf{Feedback}(S) \quad = \quad & \{\sigma_x \leadsto \sigma_u, \sigma_y, \sigma_x\} \\
& \circ\ \mathsf{IterateOmega}([\sigma_u, \sigma_y, \sigma_x \leadsto \sigma_u, \sigma_x, \sigma_x] \circ (S \otimes \mathsf{Skip})) \\
& \circ\ [\sigma_u, \sigma_y, \sigma_x \leadsto \sigma_y]
\end{aligned}$$

The idea of the feedback operator is similar to the IterateOmega operator, but from one iteration to the next we reuse the input x and we expect the output u to stabilize. While iterating we ignore the y component of the output. To achieve this we apply IterateOmega to $T = [\sigma_u, \sigma_y, \sigma_x \leadsto \sigma_u, \sigma_x, \sigma_x] \circ (S \otimes \mathsf{Skip})$. T is represented graphically in Figure 6. For the calculation of the feedback we need a mechanism of starting the computation, i.e. we need a way of assigning a suitable value to the feedback variable u. One obvious choice would be to assign to u some arbitrary value, using demonic nondeterminism. However this does not work because if S fails for some values of u, then the entire feedback will fail, even if u would stabilize to some value that is legal for S. To solve this problem we assign an arbitrary value to u using angelic nondeterminism. The consequence of this is that u will be chosen such that further failures are avoided. This approach results in meaningful computations when successive computations of S overwrite the previous values of u with values depending on the x input only. The last part of the definition of the feedback selects only the y output component as the output of the feedback.

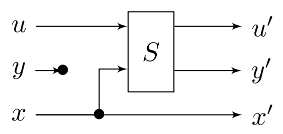


Figure 6: Padding of S for applying $\mathsf{IterateOmega}$. The figure depicts the component $[\sigma_u, \sigma_y, \sigma_x \rightsquigarrow \sigma_u, \sigma_x, \sigma_x] \circ (S \otimes \mathsf{Skip})$.

Example 14. As an example we show how to derive $\mathsf{Feedback}(S)$ for $S = [\sigma_u, \sigma_x \rightsquigarrow 0{\cdot}\sigma_x{+}0{\cdot}\sigma_u, 0{\cdot}\sigma_x{+}0{\cdot}\sigma_u]$, where $\sigma{+}\sigma' = (\lambda i : \sigma(i){+}\sigma'(i))$, and $0{\cdot}\sigma$ is 0 concatenated with σ. For now, we note that S is the semantics of the composite component `Add`;`UnitDelay`;`Split`, which corresponds to the inner part of the diagram of Fig. 2, before applying feedback. We will complete the formal definition of the semantics of this diagram in §4.3 (see Example 15). For now, we focus on deriving $\mathsf{Feedback}(S)$, in order to illustrate how the $\mathsf{Feedback}$ operator works.

Let

$$T = [\sigma_u, \sigma_y, \sigma_x \rightsquigarrow \sigma_u, \sigma_x, \sigma_x] \circ (S \otimes \mathsf{Skip}) = [\sigma_u, \sigma_y, \sigma_x \rightsquigarrow 0 \cdot \sigma_x + 0 \cdot \sigma_u, 0 \cdot \sigma_x + 0 \cdot \sigma_u, \sigma_x]$$

Then, we have

$$\begin{array}{lcl} T \circ T & = & [\sigma_u, \sigma_y, \sigma_x \rightsquigarrow 0 \cdot \sigma_x + 0 \cdot 0 \cdot \sigma_x + 0 \cdot 0 \cdot \sigma_u, 0 \cdot \sigma_x + 0 \cdot 0 \cdot \sigma_x + 0 \cdot 0 \cdot \sigma_u, \sigma_x] \\ \ldots & & \\ T^n & = & [\sigma_u, \sigma_y, \sigma_x \rightsquigarrow 0 \cdot \sigma_x + \ldots + 0^n \cdot \sigma_x + 0^n \cdot \sigma_u, 0 \cdot \sigma_x + \ldots + 0^n \cdot \sigma_x + 0^n \cdot \sigma_u, \sigma_x] \end{array}$$

where 0^n is a finite sequence of n 0s. We also have

$$\begin{array}{ll} & T^n \circ [\sigma \rightsquigarrow \sigma' \mid \forall i : i + 1 < n \Rightarrow \sigma_i = \sigma'_i] \\ = & \\ & [\sigma_u, \sigma_y, \sigma_x \rightsquigarrow \sigma \mid \forall i : i + 1 < n \Rightarrow \sigma(i) = (\Sigma_{j<i}\sigma_x(j), \Sigma_{j<i}\sigma_x(j), \sigma_x(i))] \end{array} \quad (3)$$

Then

$$\begin{array}{cl} & \mathsf{Feedback}(S) \\ = & \{\text{Definition of } \mathsf{Feedback}\} \\ & \{\sigma_x \rightsquigarrow \sigma_u, \sigma_y, \sigma_x\} \circ \mathsf{IterateOmega}(T) \circ\ [\sigma_u, \sigma_y, \sigma_x \rightsquigarrow \sigma_y] \\ = & \{\text{Definition of } \mathsf{IterateOmega}\} \\ & \{\sigma_x \rightsquigarrow \sigma_u, \sigma_y, \sigma_x\} \circ \mathsf{Fusion}_{n\in\mathbb{N}}(T^n \circ [\sigma \rightsquigarrow \sigma' \mid \forall i : i + 1 < n \Rightarrow \sigma_i = \sigma'_i]) \circ\ [\sigma_u, \sigma_y, \sigma_x \rightsquigarrow \sigma_y] \\ = & \{\text{Calculation (3)}\} \\ & \{\sigma_x \rightsquigarrow \sigma_u, \sigma_y, \sigma_x\} \circ \mathsf{Fusion}_{n\in\mathbb{N}}([\sigma_u, \sigma_y, \sigma_x \rightsquigarrow \sigma \mid \forall i : i + 1 < n \Rightarrow \\ & \qquad \sigma(i) = (\Sigma_{j<i}\sigma_x(j), \Sigma_{j<i}\sigma_x(j), \sigma_x(i))]) \circ\ [\sigma_u, \sigma_y, \sigma_x \rightsquigarrow \sigma_y] \\ = & \{\text{Lemma 7}\} \\ & \{\sigma_x \rightsquigarrow \sigma_u, \sigma_y, \sigma_x\} \circ\ [\sigma_u, \sigma_y, \sigma_x \rightsquigarrow \sigma \mid \forall i : \sigma(i) = (\Sigma_{j<i}\sigma_x(j), \Sigma_{j<i}\sigma_x(j), \sigma_x(i))] \circ\ [\sigma_u, \sigma_y, \sigma_x \rightsquigarrow \sigma_y] \\ = & \{\text{Lemma 5}\} \\ & [\sigma_x \rightsquigarrow \sigma_y \mid \forall i : \sigma_y(i) = \Sigma_{j<i}\sigma_x(j)] \end{array}$$

Finally we obtain

$$\mathsf{Feedback}(S) = [\sigma_x \rightsquigarrow \sigma_y \mid \forall i : \sigma_y(i) = \Sigma_{j<i}\sigma_x(j)] \quad (4)$$

This is the system that outputs the trace $(\lambda i : \Sigma_{j<i}\sigma_x(j))$ for input trace σ_x.

4.1.5 Refinement

A key element of RCRS, as of other compositional frameworks, is the notion of refinement, which enables substitutability and other important concepts of compositionality. Semantically, refinement is defined as follows:

Definition 19 (Refinement)**.** *Let $S, T : (\Sigma_y^\omega \to \mathbb{B}) \to (\Sigma_x^\omega \to \mathbb{B})$ be two MPTs. We say that T* refines *S (or that S* is refined by *T), written $S \sqsubseteq T$, if and only if $\forall q : S(q) \subseteq T(q)$.*

We note that in some other frameworks and in particular the one of relational interfaces [86], notation $S \sqsubseteq T$ is used to mean that S refines T. In contrast, in RCRS we follow the notation of standard refinement calculus [12] and use $S \sqsubseteq T$ to mean that T refines S.

All operations introduced on MPTs preserve the refinement relation:

Theorem 1. *If S, T, S', T' are MPTs of appropriate types such that $S \sqsubseteq S'$ and $T \sqsubseteq T'$, then*

1. *$S \circ T \sqsubseteq S' \circ T'$ and $S \otimes T \sqsubseteq S' \otimes T'$ and $\mathsf{Fusion}(S, T) \sqsubseteq \mathsf{Fusion}(S', T')$ ([11, 12])*
2. *$\mathsf{IterateOmega}(S) \sqsubseteq \mathsf{IterateOmega}(S')$*
3. *$\mathsf{Feedback}(S) \sqsubseteq \mathsf{Feedback}(S')$*

4.2 Other Subclasses of MPTs and Closure Properties

As mentioned in Section 4.1.2, in practice we often deal with subclasses of MPTs. The subclass of RPTs has already been introduced in Section 4.1.2. Here, we introduce additional subclasses of MPTs and also discuss closure properties with respect to the various operators introduced earlier. As we mentioned in the introduction to this section, similar closure results are known in the literature in other contexts, e.g., see [12, 90, 86].

4.2.1 Closure properties of RPTs

Theorem 2 (RPTs are closed under serial, parallel and fusion compositions)**.** *Let $S = \{p\} \circ [r]$ and $S' = \{p'\} \circ [r']$ be two RPTs, with p, p', r and r' of appropriate types. Then*

$$S \circ S' = \{\sigma \mid p(\sigma) \wedge (\forall \sigma' : r(\sigma)(\sigma') \Rightarrow p'(\sigma'))\} \circ [r \circ r']$$

and

$$S \otimes S' = \{\sigma_x, \sigma_y \mid p(\sigma_x) \wedge p'(\sigma_y)\} \circ [\sigma_x, \sigma_y \leadsto \sigma'_x, \sigma'_y \mid r(\sigma_x)(\sigma'_x) \wedge r'(\sigma_y)(\sigma'_y)]$$

and

$$\mathsf{Fusion}(S, S') = \{p \wedge p'\} \circ [r \wedge r']$$

Theorem 2 states that RPTs are closed under serial composition, product and fusion. In the case of serial composition, the update part $[r \circ r']$ of the composite RPT $S \circ S'$ is formed by the composition of the corresponding relations r and r' of the update parts of S and S', respectively. The assert part of $S \circ S'$ is formed by considering as legal inputs only those inputs σ which: (1) satisfy the input condition p of S, and (2) are such that for any output σ' that may be produced by S from input σ, σ' is guaranteed to satisfy the input condition p' of S'. This is similar to the way *composition by connection* is defined in the theory of relational interfaces [86]. Note however that in the case of RCRS this property is not a definition, but a theorem that follows from the semantical definition of serial/function composition of MPTs. In the case of product, a pair of inputs σ_x and σ_y is legal for the composite RPT $S \otimes S'$ when σ_x is a legal input of S and σ_y is legal for S'. An output pair is formed by an output σ'_x of S and an output σ'_y of S'. In the case of fusion, both the assert and update parts of the composite $\mathsf{Fusion}(S, S')$ are formed by taking the conjunction of the corresponding parts of S and S'.

RPTs are not closed under $\mathsf{Feedback}$. For example, we have

$$\mathsf{Feedback}([\sigma_x, \sigma_z \leadsto \sigma_x, \sigma_x]) = \{\sigma \leadsto \sigma' \mid \mathtt{true}\}$$

which is a non-relational, angelic update transformer as we said above.

The next theorem shows that the refinement of RPTs can be reduced to proving a first order property.

Theorem 3. *For p, p', r, r' of appropriate types we have:*

$$\{p\} \circ [r] \sqsubseteq \{p'\} \circ [r'] \iff (p \subseteq p' \wedge (\forall \sigma_x : p(\sigma_x) \Rightarrow r'(\sigma_x) \subseteq r(\sigma_x)))$$

That is, an RPT $S' = \{p'\} \circ [r']$ refines another RPT $S = \{p\} \circ [r]$ iff (1) the input condition p of S is stronger than the input condition p' of S' (in other words, every input which is legal in S is also legal in S'), and (2) for every legal input σ_x of S, the set of outputs that can be produced by S' is a subset of the set of outputs that can be produced by S. This property matches the definition of refinement in the theory of relational interfaces [86] but again we note that in the case of relational interfaces this is a definition, whereas in RCRS it is a theorem which follows from the semantical definition of refinement in terms of MPTs (Definition 19).

4.2.2 Guarded MPTs

Relational property transformers correspond to systems that have natural syntactic representations, as the composition $\{p\} \circ [r]$, where the predicate p and the relation r can be represented syntactically in some logic. Unfortunately, RPTs are still too powerful. In particular, they allow system semantics that cannot be implemented. For example, consider the RPT $\mathsf{Magic} = [\sigma \leadsto \sigma' \mid \mathsf{false}]$. It can be shown that for any output property q (including $\bot$), we have $\mathsf{Magic}(q) = \top$. Recall that, viewed as a set, $\top$ is the set of all traces. This means that, no matter what the post-condition q is, Magic somehow manages to produce output traces satisfying q no matter what the input trace is (hence the name "magic"). In general, an MPT S is said to be *non-miraculous* (or to satisfy the *law of excluded miracle*) if $S(\bot) = \bot$. We note that in [31], sequential programs are modeled using predicate transformers that are *conjunctive* and satisfy the law of excluded miracle.

We want to further restrict RPTs so that miraculous behavior does not arise. Specifically, for an RPT $S = \{p\} \circ [r]$ and an input sequence σ, if there is no σ' such that $r(\sigma)(\sigma')$ is satisfied, then we want σ to be illegal, i.e., we want $p(\sigma) = \mathsf{false}$. We can achieve this by taking p to be $\mathsf{in}(r)$. Recall that if $r : X \to Y \to \mathbb{B}$, then $\mathsf{in}(r)(x) = (\exists y : r(x)(y))$. Taking p to be $\mathsf{in}(r)$ effectively means that p and r are combined into a single specification r which can also restrict the inputs. This is also the approach followed in the theory of relational interfaces [86].

Definition 20 (Guarded property transformers)**.** *The* guarded property transformer *(GPT) of a relation r is an RPT, denoted $\{r]$, defined by $\{r] = \{\mathsf{in}(r)\} \circ [r]$.*

It can be shown that an MPT S is a GPT if and only if S is conjunctive and non-miraculous [12]. Fail, Skip, and any assert property transformer are GPTs. Indeed, $\mathsf{Fail} = \{\bot]$ and $\mathsf{Skip} = \{\sigma \leadsto \sigma \mid \top]$. The assert transformer can be written as $\{p\} = \{\sigma \leadsto \sigma \mid p(\sigma)]$. The angelic and demonic update property transformers are generally not GPTs. The angelic update property transformer is not always conjunctive in order to be a GPT. The demonic update property transformer is not in general a GPT because is not always non-miraculous ($\mathsf{Magic}(\bot) = \top \neq \bot$). The demonic update transformer $[r]$ is a GPT if and only if $\mathsf{in}(r) = \top$ and in this case we have $[r] = \{r]$.

Theorem 4 (GPTs are closed under serial and parallel compositions)**.** *Let $S = \{r]$ and $S' = \{r']$ be two GPTs with r and r' of appropriate types. Then*

$$S \circ S' = \{\sigma_x \leadsto \sigma_z \mid \mathsf{in}(r)(\sigma_x) \wedge \big(\forall \sigma_y : r(\sigma_x)(\sigma_y) \Rightarrow \mathsf{in}(r')(\sigma_y)\big) \wedge (r \circ r')(\sigma_x, \sigma_z)]$$

and

$$S \otimes S' = \{\sigma_x, \sigma_y \leadsto \sigma'_x, \sigma'_y \mid r(\sigma_x)(\sigma'_x) \wedge r'(\sigma_y)(\sigma'_y)]$$

GPTs are not closed under Fusion neither $\mathsf{Feedback}$. Indeed, we have already seen in the previous section that $\mathsf{Feedback}$ applied to the GPT $[\sigma_x, \sigma_z \rightsquigarrow \sigma_x, \sigma_x]$ is not an RPT, and therefore not a GPT either. For the fusion operator, we have $\mathsf{Fusion}([x \rightsquigarrow 0], [x \rightsquigarrow 1]) = [\bot]$, which is not a GPT.

A corollary of Theorem 3 is that refinement of GPTs can be checked as follows:

Corollary 1.

$$\{r\} \sqsubseteq \{r'\} \iff (\mathsf{in}(r) \subseteq \mathsf{in}(r') \wedge (\forall \sigma_x : \mathsf{in}(r)(\sigma_x) \Rightarrow r'(\sigma_x) \subseteq r(\sigma_x)))$$

4.2.3 Other subclasses and overview

The containment relationships among the various subclasses of MPTs are illustrated in Fig. 7. In addition to the subclasses discussed above, we introduce several more subclasses of MPTs in the sections that follow, when we assign semantics (in terms of MPTs) to the various atomic components in our component language. For instance, QLTL components give rise to *QLTL property transformers*. Similarly, STS components, stateless STS components, etc., give rise to corresponding subclasses of MPTs. The containment relationships between these classes will be proven in the sections that follow. For ease of reference, we provide some forward links to these results also here. The fact that QLTL property transformers are GPTs follows by definition of the semantics of QLTL components: see §4.3, equation (5). The fact that STS property transformers are a special case of QLTL property transformers follows from the transformation of an STS component into a semantically equivalent QLTL component: see §5.1 and Theorem 7. The inclusions for subclasses of STS property transformers follow by definition of the corresponding components (see also Fig. 8).

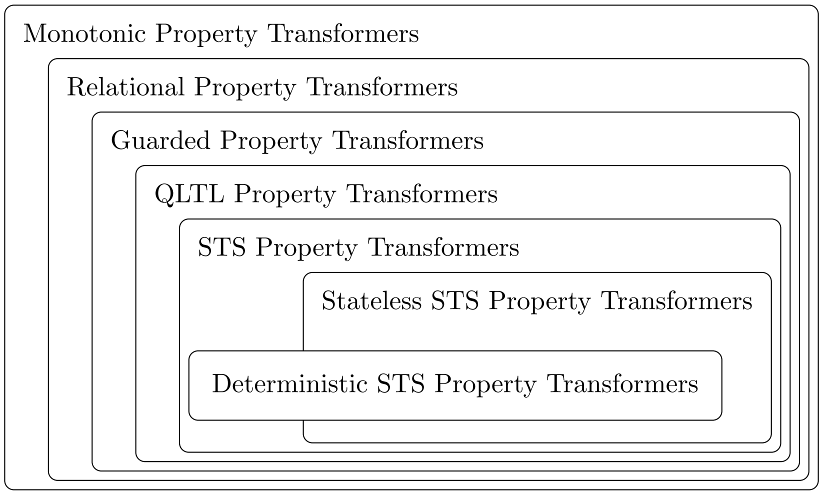

Monotonic Property Transformers
Relational Property Transformers
Guarded Property Transformers
QLTL Property Transformers
STS Property Transformers
Stateless STS Property Transformers
Deterministic STS Property Transformers

Figure 7: Overview of the property transformer classes and their containment relations.

4.3 Semantics of Components as MPTs

We are now ready to define the semantics of our language of components in terms of MPTs. Let C be a well formed component. The semantics of C, denoted $[\![C]\!]$, is a property transformer of the form:

$$[\![C]\!] : ((\Sigma_{out}(C))^\omega \to \mathbb{B}) \to ((\Sigma_{in}(C))^\omega \to \mathbb{B}).$$

We define $[\![C]\!]$ by induction on the structure of C. First we give the semantics of QLTL components and composite components:

$$\begin{aligned}
[\![\texttt{qltl}(x, y, \varphi)]\!] &= \{\sigma_x \rightsquigarrow \sigma_y \mid (\sigma_x, \sigma_y) \models \varphi\} && (5)\\
[\![C\,;C']\!] &= [\![C]\!] \circ [\![C']\!] && (6)\\
[\![C \parallel C']\!] &= [\![C]\!] \otimes [\![C']\!] && (7)\\
[\![\texttt{fdbk}(C)]\!] &= \mathsf{Feedback}([\![C]\!]) && (8)
\end{aligned}$$

To define the semantics of STS components, we first introduce some auxiliary notation.

Consider an STS component $C = \texttt{sts}(x, y, s, \mathit{init\_exp}, \mathit{trs\_exp})$. We define the predicate $\mathsf{run}_C : \Sigma_s^\omega \times \Sigma_x^\omega \times \Sigma_y^\omega \to \mathbb{B}$ as

$$\mathsf{run}_C(\sigma_s, \sigma_x, \sigma_y) = (\forall i : \mathit{trs\_exp}(\sigma_s(i), \sigma_x(i))(\sigma_s(i+1), \sigma_y(i))).$$

Intuitively, if $\sigma_x \in \Sigma_x^\omega$ is the input sequence, $\sigma_y \in \Sigma_y^\omega$ is the output sequence, and $\sigma_s \in \Sigma_s^\omega$ is the sequence of values of state variables, then $\mathsf{run}_C(\sigma_s, \sigma_x, \sigma_y)$ holds if at each step of the execution, the current state, current input, next state, and current output, satisfy the $\mathit{trs\_exp}$ predicate.

We also formalize the illegal input traces of STS component C as follows:

$$\begin{aligned}
\mathsf{illegal}_C(\sigma_x) = (\exists \sigma_s, \sigma_y, k : \mathit{init\_exp}(\sigma_s(0)) \;\wedge (\forall i < k : \mathit{trs\_exp}(\sigma_s(i), \sigma_x(i))(\sigma_s(i+1), \sigma_y(i))) \;\wedge \\
\neg \mathsf{in}(\mathit{trs\_exp})(\sigma_s(k), \sigma_x(k)))
\end{aligned}$$

Essentially, $\mathsf{illegal}_C(\sigma_x)$ states that there exists some point in the execution where the current state and current input violate the precondition $\mathsf{in}(\mathit{trs\_exp})$ of predicate $\mathit{trs\_exp}$, i.e., there exist no output and next state to satisfy $\mathit{trs\_exp}$ for that given current state and input.

Then, the semantics of an STS component C is given by:

$$[\![C]\!] = \{\neg \mathsf{illegal}_C\} \circ [\sigma_x \rightsquigarrow \sigma_y \mid (\exists \sigma_s : \mathit{init\_exp}(\sigma_s(0)) \wedge \mathsf{run}_C(\sigma_s, \sigma_x, \sigma_y))] \tag{9}$$

We give semantics to stateless and/or deterministic STS components using the corresponding mappings from general STS components. If C is a stateless STS, C' is a deterministic STS, and C'' is a stateless deterministic STS, then:

$$\begin{aligned}
[\![C]\!] &= [\![\texttt{stateless2sts}(C)]\!] && (10)\\
[\![C']\!] &= [\![\texttt{det2sts}(C')]\!] && (11)\\
[\![C'']\!] &= [\![\texttt{stateless\_det2det}(C'')]\!] && (12)
\end{aligned}$$

Note that the semantics of a stateless deterministic STS component C'' is defined by converting C'' into a deterministic STS component, by Equation (12) above. Alternatively, we could have defined the semantics of C'' by converting it into a stateless STS component, using the mapping `stateless_det2stateless`. In order for our semantics to be well-defined, we need to show that regardless of which conversion we choose, we obtain the same result. Indeed, this is shown by the lemma that follows:

Lemma 8. *For a stateless deterministic STS C'' we have:*

$$[\![\texttt{stateless\_det2det}(C'')]\!] = [\![\texttt{stateless\_det2stateless}(C'')]\!] \tag{13}$$

Observe that, by definition, the semantics of QLTL components are GPTs. The semantics of STS components are defined as RPTs. However, they will be shown to be GPTs in §5.1. Therefore, the semantics of all atomic RCRS components are GPTs. This fact, and the closure of GPTs w.r.t. parallel and serial composition (Theorem 4), ensure that we stay within the GPT realm as long as no feedback operations are used. In addition, as we shall prove in Corollary 2, components with feedback are also GPTs, as long as they are deterministic and do not contain *algebraic loops*. An example of a component whose semantics is *not* a GPT is:

$$C = \texttt{fdbk}(\texttt{stateless}((x, z), (y_1, y_2), y_1 = x \wedge y_2 = x))$$

Then, we have $[\![C]\!] = \mathsf{Feedback}([\sigma_x, \sigma_z \rightsquigarrow \sigma_x, \sigma_x])$. As stated earlier, $\mathsf{Feedback}([\sigma_x, \sigma_z \rightsquigarrow \sigma_x, \sigma_x])$ is equal to $\{\sigma \rightsquigarrow \sigma' \mid \mathtt{true}\}$, which is not a GPT neither an RPT. The problem with C is that it contains an algebraic loop: the first output y_1 of the internal stateless component where feedback is applied directly depends on its first input x. Dealing with such components is beyond the scope of this paper, and we refer the reader to [73].

4.3.1 Example: Two Alternative Derivations of the Semantics of `Sum`

To illustrate our semantics, we provide two alternative derivations of the semantics of the `Sum` system of Fig. 2, Example 1.

Example 15. First, let us consider `Sum` as a composite component, as in Example 1:

$$\mathtt{Sum} = \mathtt{fdbk}(\mathtt{Add}\,;\mathtt{UnitDelay}\,;\mathtt{Split})$$

where

$$\begin{array}{lcl} \mathtt{Add} & = & \mathtt{stateless\_det}((u,x),\mathtt{true},u+x) \\ \mathtt{UnitDelay} & = & \mathtt{det}(x,s,0,\mathtt{true},x,s) \\ \mathtt{Split} & = & \mathtt{stateless\_det}(x,\mathtt{true},(x,x)) \end{array}$$

We have

$$\begin{array}{lcl} [\![\mathtt{Sum}]\!] & = & [\![\mathtt{fdbk}(\mathtt{Add}\,;\mathtt{UnitDelay}\,;\mathtt{Split})]\!] \\ & = & \mathsf{Feedback}([\![\mathtt{Add}]\!] \circ [\![\mathtt{UnitDelay}]\!] \circ [\![\mathtt{Split}]\!]) \end{array}$$

For `Add`, `UnitDelay` and `Split`, all inputs are legal, so $\mathsf{illegal}_C = \bot$ for all $C \in \{\mathtt{Add}, \mathtt{UnitDelay}, \mathtt{Split}\}$. After simplifications, we get:

$$\begin{array}{lcl} [\![\mathtt{Add}]\!] & = & [\sigma_u, \sigma_x \rightsquigarrow \sigma_u + \sigma_x] \\ [\![\mathtt{UnitDelay}]\!] & = & [\sigma_x \rightsquigarrow 0 \cdot \sigma_x] \\ [\![\mathtt{Split}]\!] & = & [\sigma_x \rightsquigarrow \sigma_x, \sigma_x] \end{array}$$

The semantics of `Sum` is given by

$$\begin{array}{ll} & [\![\mathtt{Sum}]\!] \\ = & \\ & \mathsf{Feedback}([\![\mathtt{Add}]\!] \circ [\![\mathtt{UnitDelay}]\!] \circ [\![\mathtt{Split}]\!] \\ = & \\ & \mathsf{Feedback}([\sigma_u, \sigma_x \rightsquigarrow \sigma_u + \sigma_x] \circ [\sigma_x \rightsquigarrow 0 \cdot \sigma_x] \circ [\sigma_x \rightsquigarrow \sigma_x, \sigma_x]) \\ = & \\ & \mathsf{Feedback}([\sigma_u, \sigma_x \rightsquigarrow 0 \cdot \sigma_x + 0 \cdot \sigma_u, 0 \cdot \sigma_x + 0 \cdot \sigma_u]) \\ = & \{\text{Using } (4)\} \\ & [\sigma_x \rightsquigarrow \sigma_y \mid \forall i : \sigma_y(i) = \Sigma_{j<i} \sigma_x(j)] \end{array}$$

We obtain:

$$[\![\mathtt{Sum}]\!] = [\sigma_x \rightsquigarrow \sigma_y \mid \forall i : \sigma_y(i) = \Sigma_{j<i} \sigma_x(j)]. \tag{14}$$

Next, let us assume that the system has been characterized already as an atomic component:

$$\mathtt{SumAtomic} = \mathtt{sts}(x, y, s, s = 0, y = s \wedge s' = s + x).$$

The semantics of `SumAtomic` is given by

$$[\![\mathtt{SumAtomic}]\!] = \{\neg \mathsf{illegal}_{\mathtt{SumAtomic}}\} \circ [\sigma_x \rightsquigarrow \sigma_y \mid \exists \sigma_s : \sigma_s(0) = 0 \wedge \mathsf{run}_{\mathtt{SumAtomic}}(\sigma_s, \sigma_x, \sigma_y)]$$

where $\mathsf{illegal}_{\mathtt{SumAtomic}} = \bot$ because there are no restrictions on the inputs of `SumAtomic`, and

$$\mathsf{run}_{\mathtt{SumAtomic}}(\sigma_s, \sigma_x, \sigma_y) = \big(\forall i : \sigma_y(i) = \sigma_s(i) \wedge \sigma_s(i+1) = \sigma_s(i) + \sigma_x(i)\big)$$

We have

$$[\![\mathtt{SumAtomic}]\!] = [\sigma_x \rightsquigarrow \sigma_y \mid \exists \sigma_s : \sigma_s(0) = 0 \wedge \big(\forall i : \sigma_y(i) = \sigma_s(i) \wedge \sigma_s(i+1) = \sigma_s(i) + \sigma_x(i)\big)]$$

which is equivalent to (14).

4.3.2 Characterization of Legal Input Traces

The following lemma characterizes the legal input traces for various types of MPTs:

Lemma 9. *The set of legal input traces of an RPT $\{p\} \circ [r]$ is p:*

$$\mathsf{legal}(\{p\} \circ [r]) = p.$$

The set of legal input traces of a GPT $\{r]$ is $\mathsf{in}(r)$:

$$\mathsf{legal}(\{r]) = \mathsf{in}(r).$$

The set of legal input traces of an STS component $C = \mathtt{sts}(x, y, s, init, r)$ is equal to $\neg\mathsf{illegal}_C$:

$$\mathsf{legal}([\![C]\!]) = \neg\mathsf{illegal}_C.$$

The set of legal input traces of a QLTL component $C = \mathtt{qltl}(x, y, \varphi)$ is:

$$\mathsf{legal}([\![C]\!]) = \{\sigma_x \mid \sigma_x \models \exists y : \varphi\}.$$

The first two properties of Lemma 9 give the semantic characterization of legal inputs for RPTs and GPTs. The last two properties link the semantic definition of legal inputs to the operational definition of legal inputs for STS and QLTL components. Lemma 9 is important for the results of Section 5.9.

4.3.3 Semantic Equivalence and Refinement for Components

Definition 21. *Two components C and C' are* (semantically) equivalent, *denoted $C \equiv C'$, if $[\![C]\!] = [\![C']\!]$. Component C is refined by component C', denoted $C \preceq C'$, if $[\![C]\!] \sqsubseteq [\![C']\!]$.*

The relation $\equiv$ is an equivalence relation, and $\preceq$ is a preorder relation (i.e., reflexive and transitive). We also have

$$(C \preceq C' \wedge C' \preceq C) \iff (C \equiv C')$$

The notions of semantic equivalence and refinement for components are needed in order to express the compositionality properties in Section 4.3.4 that follows. Semantic equivalence is also necessary in order to establish the correctness of the symbolic transformations proposed in Section 5.

4.3.4 Compositionality Properties

Several desirable compositionality properties follow from our semantics:

Theorem 5. *Let C_1, C_2, C_3, and C_4 be four (possibly composite) components. Then:*

1. *(Serial composition is associative:)* $(C_1 \,; C_2) \,; C_3 \equiv C_1 \,; (C_2 \,; C_3)$.
2. *(Parallel composition is associative:)* $(C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3)$.
3. *(Parallel composition distributes over serial composition:) If $[\![C_1]\!]$ and $[\![C_2]\!]$ are GPTs and $[\![C_3]\!]$ and $[\![C_4]\!]$ are RPTs, then* $(C_1 \parallel C_2) \,; (C_3 \parallel C_4) \equiv (C_1 \,; C_3) \parallel (C_2 \,; C_4)$.
4. *(Refinement is preserved by composition:) If $C_1 \preceq C_2$ and $C_3 \preceq C_4$, then:*
   - *(a)* $C_1 \,; C_3 \preceq C_2 \,; C_4$
   - *(b)* $C_1 \parallel C_3 \preceq C_2 \parallel C_4$
   - *(c)* $\mathtt{fdbk}(C_1) \preceq \mathtt{fdbk}(C_2)$

In addition to the above, the requirements a component satisfies are preserved by refinement. Informally, if C satisfies some requirement φ and $C \preceq C'$ then C' also satisfies φ. Although we have not formally defined what requirements are and what it means for a component to satisfy a requirement, these concepts are naturally captured in the RCRS framework via the semantics of components as MPTs. In particular, since our components are generally *open* systems (i.e., they have inputs), we can express requirements using *Hoare triples* of the form $p\{C\}q$, where C is a component, p is an input property, and q is an output property. Then, $p\{C\}q$ holds iff the outputs of C are guaranteed to satisfy q provided the inputs of C satisfy p. Formally: $p\{C\}q \iff p \subseteq [\![C]\!](q)$. This definition is the same as the one for predicate transformers from [12] and has been also explored in other contexts, e.g. [8].

Theorem 6. *$C \preceq C'$ iff $\forall p, q : p\{C\}q \Rightarrow p\{C'\}q$.*

Theorem 6 shows that refinement is equivalent to *substitutability.* Substitutability states that a component C' can replace another component C in any context, i.e., $\forall p, q : p\{C\}q \Rightarrow p\{C'\}q$.

5 Symbolic Reasoning

So far we have defined the syntax and semantics of RCRS. These already allow us to specify and reason about systems in a compositional manner. However, such reasoning is difficult to do "by hand". For example, if we want to check whether a component C is refined by another component C', we must resort to proving the refinement relation $[\![C]\!] \sqsubseteq [\![C']\!]$ of their corresponding MPTs, $[\![C]\!]$ and $[\![C']\!]$. This is not an easy task, as MPTs are complex mathematical objects. Instead, we would like to have computer-aided, and ideally fully automatic techniques. In the above example of checking refinement, for instance, we would like ideally to have an algorithm that takes as input the syntactic descriptions of C and C' and replies yes/no based on whether $[\![C]\!] \sqsubseteq [\![C']\!]$ holds. We say "ideally" because we know that in general such an algorithm cannot exist. This is because we are not making a-priori any restrictions on the logics used to describe C and C', which means that the existence of an algorithm will depend on the decidability of these logics. In this section, we describe how reasoning in RCRS can be done *symbolically*, by automatically manipulating the formulas used to specify the components involved. As we shall show, most of the transformations are purely syntactic, and the remaining problems can be reduced to checking satisfiability of first-order formulas formed by combinations of the formulas of the original components. This means that these problems are decidable whenever the corresponding first-order logics are decidable.

5.1 Syntactic Transformation of STS Components to QLTL Components

Our framework allows the specification of several types of atomic components, some of which are special cases of others, as summarized in Fig. 8. In §3, we have already shown how the different types of STS components are related, from the most specialized deterministic stateless STS components, to the general STS components. By definition, the semantics of the special types of STS components is defined via the semantics of general STS components (see §4). In this subsection, we show that STS components can be viewed as special cases of QLTL components.

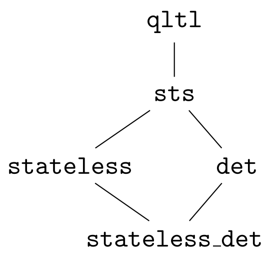


Figure 8: Lattice of atomic components: lower types are special cases of higher types.

Specifically, we show how an STS component can be syntactically transformed into a semantically equivalent QLTL component. This transformation also shows that STS property transformers are a special case of QLTL property transformers, as already claimed in Fig. 7. Note that this containment is not obvious simply by looking at the definitions of these MPT subclasses (c.f. §4.3), as QLTL property transformers are defined as GPTs (equation 5), whereas STS property transformers are defined as RPTs (equation 9). Although RPTs are generally a superclass, not a subclass, of GPTs, the transformation proposed below shows that the RPTs obtained from STS components can indeed be captured as GPTs. The transformation of STS into QLTL components also enables us to apply several algorithms which are available for QLTL components to STS components as well.

We can transform an STS component $C = \mathtt{sts}(x, y, s, \mathit{init}, \mathit{trs})$ into a QLTL component using the syntactic transformation operator `sts2qltl`:

$$\mathtt{sts2qltl}(\mathtt{sts}(x, y, s, \mathit{init}, \mathit{trs})) = \mathtt{qltl}(x, y, (\forall s, y : \mathit{init} \Rightarrow (\varphi \,\mathbf{L}\, \varphi')) \wedge (\exists s : \mathit{init} \wedge \mathbf{G}\, \varphi)) \tag{15}$$

where: $\varphi = \mathit{trs}[s' := \bigcirc s]$; $e[z := e']$ denotes the substitution of all free occurrences of variable z by expression e' in expression e; and $\varphi' = (\exists s', y : \mathit{trs})$. Here we use the $\mathbf{L}$ operator to express the fact that however the computation proceeds, starting with an initial state, and an input sequence x, if we reach the computation step n $((\forall 0 \le i < n : \sigma^i \models \phi) \Leftrightarrow (\forall 0 \le i < n : \mathit{trs}(s_i, x_i, s_{i+1}, y_i)))$, then φ' must hold at n, i.e. we must be able to continue the computation $(\exists s_{n+1}, y_n : \mathit{trs}(s_n, x_n, s_{n+1}, y_n))$.

The theorem that follows demonstrates the correctness of the above transformation, that is, that the resulting QLTL component is semantically equivalent to the original STS component:

Theorem 7. *For any STS component $C = \mathtt{sts}(x, y, s, \mathit{init}, \mathit{trs})$ s.t. init is satisfiable, $C \equiv \mathtt{sts2qltl}(C)$.*

Example 16. It is instructive to see how the above transformation specializes to some special types of STS components. In particular, we will show how it specializes to stateless STS components.

Let $C = \mathtt{stateless}(x, y, \mathit{trs})$ and let $C' = \mathtt{stateless2sts}(C) = \mathtt{sts}(x, y, (), \mathtt{true}, \mathit{trs})$. Applying the `sts2qltl` transformation to C', for which $s = ()$ and $\mathit{init} = \mathtt{true}$, we obtain:

$$\mathtt{sts2qltl}(C') = \mathtt{qltl}\big(x, y, (\forall y : (\varphi \,\mathbf{L}\, \varphi')) \wedge \mathbf{G}\, \varphi)\big)$$

where $\varphi = \mathit{trs}[() := \bigcirc ()] = \mathit{trs}$ and $\varphi' = (\exists y : \mathit{trs})$. Using the properties of Lemma 2, and the fact that semantically equivalent LTL formulas result in semantically equivalent QLTL components, we can simplify $\mathtt{sts2qltl}(C')$ further:

$$\begin{array}{cl}
 & \mathtt{sts2qltl}(C') \\
= & \{\text{Definition of } \mathtt{sts2qltl}\} \\
 & \mathtt{qltl}\big(x, y, (\forall y : (\mathit{trs}\ \ \mathbf{L}\ \ (\exists y : \mathit{trs}))) \wedge \mathbf{G}\ \mathit{trs}\big) \\
\equiv & \{\text{Lemma 2, } \mathit{trs} \text{ does not contain temporal operators, and } y \text{ is not free in } (\exists y : \mathit{trs})\} \\
 & \mathtt{qltl}\big(x, y, ((\exists y : \mathit{trs})\ \ \mathbf{L}\ \ (\exists y : \mathit{trs})) \wedge \mathbf{G}\ \mathit{trs}\big) \\
\equiv & \{\text{Lemma 2}\} \\
 & \mathtt{qltl}\big(x, y, \mathbf{G}\ (\exists y : \mathit{trs}) \wedge \mathbf{G}\ \mathit{trs}\big) \\
\equiv & \{\text{Lemma 2 and } \mathit{trs} \text{ does not contain temporal operators}\} \\
 & \mathtt{qltl}\big(x, y, (\exists y : \mathbf{G}\ \mathit{trs}) \wedge \mathbf{G}\ \mathit{trs}\big) \\
\equiv & \{(\exists y : \mathbf{G}\ \mathit{trs}) \wedge \mathbf{G}\ \mathit{trs} \text{ equivalent to } \mathbf{G}\ \mathit{trs}\} \\
 & \mathtt{qltl}\big(x, y,\ \mathbf{G}\ \mathit{trs}\big)
\end{array}$$

Note that in the above derivation we use the equivalence symbol $\equiv$, in addition to the equality symbol $=$. Recall that $\equiv$ stands for semantical equivalence of two components (c.f. §4.3.3). On the other hand, $=$ for components means syntactic equality. By definition of the semantics, if two QLTL formulas are equivalent, then the corresponding QLTL components are equivalent.

Based on the above, we define the transformation `stateless2qltl` of a stateless component $C = \texttt{stateless}(x, y, trs)$, into a QLTL component as follows:

$$\texttt{stateless2qltl}(C) = \texttt{qltl}\big(x, y, \mathbf{G}\ trs\big) \tag{16}$$

5.2 Syntactic and Symbolic Transformations of Special Atomic Components to More General Atomic Components

Based on the lattice in Fig. 8, we define all remaining mappings from more special atomic components to more general atomic components, by composing the previously defined mappings `sts2qltl`, `stateless2qltl`, `stateless2sts`, `det2sts`, `stateless_det2stateless` and `stateless_det2det`, as appropriately.

We note that all these mappings are purely syntactic. However, in order to obtain a final simplified result, more semantic manipulations may be needed, such as checking formula validity or satisfiability, as mentioned in the introduction to this section. This is illustrated in the case of `det2qltl(UnitDelay)` in Example 17 that follows. Because of this, we may use the term *symbolic* rather than *syntactic* to describe these transformations.

For mapping stateless deterministic STS components to QLTL components, we have two possibilities: `stateless_det` → `det` → `sts` → `qltl` and `stateless_det` → `stateless` → `qltl`. We choose the transformation `stateless_det` → `stateless` → `qltl` because it results in a simpler formula:

$$\texttt{stateless\_det2qltl}(C) = \texttt{stateless2qltl}(\texttt{stateless\_det2stateless}(C)) \tag{17}$$

Example 17. Consider the following STS components:

$$\begin{array}{rcl} C_1 & = & \texttt{stateless}\big(x, y, y > x\big) \\ C_2 & = & \texttt{stateless}\big(x, (), x > 0\big) \\ \texttt{UnitDelay} & = & \texttt{det}\big(x, s, 0, \texttt{true}, x, s\big) \end{array}$$

Then:

$$\begin{array}{rcl} \texttt{stateless2qltl}(C_1) & = & \texttt{qltl}\big(x, y, \mathbf{G}\ y > x\big) \\ \texttt{stateless2qltl}(C_2) & = & \texttt{qltl}\big(x, (), \mathbf{G}\ x > 0\big) \end{array}$$

and

$$\begin{array}{ll}
& \texttt{det2qltl}(\texttt{UnitDelay}) \\
= & \{\text{Definitions}\} \\
& \texttt{qltl}\big(x, y, (\forall s, y : s = 0 \Rightarrow (\texttt{true} \wedge \bigcirc s = x \wedge y = s) \ \mathbf{L} \ (\exists s', y : \texttt{true} \wedge s' = x \wedge y = s)) \wedge \\
& \quad (\exists s : s = 0 \wedge \mathbf{G} \ (\texttt{true} \wedge \bigcirc s = x \wedge y = s))\big) \\
\equiv & \{ \ (\exists s', y : \texttt{true} \wedge s' = x \wedge y = s) \text{ is true and Lemma 2 } \} \\
& \texttt{qltl}\big(x, y, (\exists s : s = 0 \wedge \mathbf{G} \ (\bigcirc s = x \wedge y = s))\big) \\
\equiv & \{\text{Lemma 2}\} \\
& \texttt{qltl}\big(x, y, \ (\exists s : y = 0 \wedge \mathbf{G} \ (y = s) \wedge \mathbf{G} \ (\bigcirc s = x))\big) \\
\equiv & \{ \ \mathbf{G} \ (y = s) \text{ is true so we can replace } s \text{ by } y \ \} \\
& \texttt{qltl}\big(x, y, \ (\exists s : y = 0 \wedge \mathbf{G} \ (y = s) \wedge \mathbf{G} \ (\bigcirc y = x))\big) \\
\equiv & \{ \text{ Logical properties } \} \\
& \texttt{qltl}\big(x, y, \ (\exists s : \mathbf{G} \ (y = s)) \wedge y = 0 \wedge \mathbf{G} \ (\bigcirc y = x)\big) \\
\equiv & \{ \text{ Lemma 2 } \} \\
& \texttt{qltl}\big(x, y, \ (\mathbf{G} \ (\exists s : y = s)) \wedge y = 0 \wedge \mathbf{G} \ (\bigcirc y = x)\big) \\
\equiv & \{ \text{ Logical properties } \} \\
& \texttt{qltl}\big(x, y, \ y = 0 \wedge \mathbf{G} \ (\bigcirc y = x)\big)
\end{array}$$

5.3 Syntactic Computation of Serial Composition

Given a composite component C formed as the serial composition of two atomic components A and B, i.e., $C = A\,;B$, we would like to compute a new, *atomic* component C_a, such that C_a is semantically equivalent to C. Because atomic components are by definition represented syntactically (and symbolically), being able to reduce composite components into atomic components means that we are able to symbolically compute composition operators.

In order to compute serial composition, we introduce the syntactic transformation operator **serial**, which computes $C_a := \textbf{serial}(A, B)$. We start by defining how **serial** works on two atomic components of the same type. Then we generalize **serial** to any two atomic components.

5.3.1 Syntactic Serial Composition of Two QLTL Components

Let $C = \texttt{qltl}(x, y, \varphi)$ and $C' = \texttt{qltl}(y, z, \varphi')$ such that $C\,;C'$ is well formed. Then their *syntactic serial composition*, denoted $\textbf{serial}(C, C')$, is the QLTL component defined by

$$\textbf{serial}(C, C') = \texttt{qltl}\big(x, z, (\forall y : \varphi \Rightarrow (\exists z : \varphi')) \wedge (\exists y : \varphi \wedge \varphi')\big) \tag{18}$$

Note that in the above definition (as well as the ones that follow) we assume that the output variable of C and the input variable of C' have the same name (y) and that the names x, y and z are distinct. In general, this may not be the case. This is not a problem, as we can always rename variables such that this condition is met. Note that variable renaming does not change the semantics of components (c.f. §3.2.2).

The intuition behind the formula in (18) is as follows. The second conjunct $\exists y : \varphi \wedge \varphi'$ ensures that the both contracts φ and φ' of the two components are enforced in the composite contract. The reason we use $\exists y : \varphi \wedge \varphi'$ instead of just the conjunction $\varphi \wedge \varphi'$ is that we want to eliminate ("hide") internal variable y. (Alternatively, we could also have chosen to output y as an additional output, but would then need an additional hiding operator to remove y.) The first conjunct $\forall y : \varphi \Rightarrow (\exists z : \varphi')$ is a formula on the input variable x of the composite component (since all other variables y and z are quantified). This formula restricts the legal inputs of C to those inputs for which, no matter which output C produces, this output is guaranteed to be a legal input for the downstream component C'. For an extensive discussion of the intuition and justification behind this definition, see [86]. Also note the similarities between (18) and the RPT corresponding to $S \circ S'$ in Theorem 2.

Sections 5.3.2 to 5.3.5 that follow provide similar syntactic transformations for the syntactic serial composition of pairs of various types of atomic STS components, from general STS components, to stateless deterministic STS components. All transformations follow the same spirit as (18) and the RPT corresponding to $S \circ S'$ in Theorem 2 mentioned above. However, the more specialized the STSs are, the simpler the resulting formula becomes, which justifies the interest in examining each case separately. Section 5.3.6 defines the syntactic serial composition of two atomic components which are not of the same type, e.g., a stateful component and a stateless component.

5.3.2 Syntactic Serial Composition of Two General STS Components

Let $C = \texttt{sts}(x, y, s, \mathit{init}, \mathit{trs})$ and $C' = \texttt{sts}(y, z, t, \mathit{init}', \mathit{trs}')$ be two general STS components such that $C\,;C'$ is well formed. Then:

$$\mathbf{serial}(C, C') = \texttt{sts}\big(x, z, (s,t), \mathit{init} \wedge \mathit{init}', (\exists s', y : \mathit{trs}) \wedge (\forall s', y : \mathit{trs} \Rightarrow (\exists t', z : \mathit{trs}')) \wedge (\exists y : \mathit{trs} \wedge \mathit{trs}')\big) \quad (19)$$

5.3.3 Syntactic Serial Composition of Two Stateless STS Components

Let $C = \texttt{stateless}(x, y, \mathit{trs})$ and $C' = \texttt{stateless}(y, z, \mathit{trs}')$ be two stateless STS components such that $C\,;C'$ is well formed. Then

$$\mathbf{serial}(C, C') = \texttt{stateless}\big(x, z, (\forall y : \mathit{trs} \Rightarrow (\exists z : \mathit{trs}')) \wedge (\exists y : \mathit{trs} \wedge \mathit{trs}')\big) \quad (20)$$

5.3.4 Syntactic Serial Composition of Two Deterministic STS Components

Let $C = \texttt{det}(x, s, a, p, \mathit{next}, \mathit{out})$ and $C' = \texttt{det}(y, t, b, p', \mathit{next}', \mathit{out}')$ be two deterministic STS components such that their serial composition $C\,;C'$ is well formed. Then:

$$\mathbf{serial}(C, C') = \texttt{det}(x, (s,t), (a,b), p \wedge p'[y := \mathit{out}], (\mathit{next}, \mathit{next}'[y := \mathit{out}]), \mathit{out}'[y := \mathit{out}]) \quad (21)$$

5.3.5 Syntactic Serial Composition of Two Stateless Deterministic STS Components

Finally, let $C = \texttt{stateless\_det}(x, p, \mathit{out})$ and $C' = \texttt{stateless\_det}(y, p', \mathit{out}')$ be two stateless deterministic STS components such that their serial composition $C\,;C'$ is well formed. Then:

$$\mathbf{serial}(C, C') = \texttt{stateless\_det}(x, p \wedge p'[y := \mathit{out}], \mathit{out}'[y := \mathit{out}]) \quad (22)$$

5.3.6 Syntactic Serial Composition of Two Arbitrary Atomic Components

In general, we define the syntactic serial composition of two atomic components C and C' by using the mappings of less general components to more general components (Fig. 8), as appropriate. For example, if C is a deterministic STS component and C' is a stateless STS component, then $\mathbf{serial}(C, C') = \mathbf{serial}(\texttt{det2sts}(C), \texttt{stateless2sts}(C'))$. Similarly, if C is a QLTL component and C' is a deterministic STS component, then $\mathbf{serial}(C, C') = \mathbf{serial}(C, \texttt{det2qltl}(C'))$. Formally, assume that $\mathit{atm}, \mathit{atm}' \in \{\texttt{qltl}, \texttt{sts}, \texttt{stateless}, \texttt{det}, \texttt{stateless\_det}\}$ are the types of the components C and C', and $\mathit{common} = \mathit{atm} \vee \mathit{atm}'$ is the least general component type that is more general than atm and atm' as defined in Fig. 8. Then

$$\mathbf{serial}(C, C') = \mathbf{serial}(\mathit{atm}2\mathit{common}(C), \mathit{atm}'2\mathit{common}(C')) \quad (23)$$

5.3.7 Correctness of Syntactic Serial Composition

The following theorem demonstrates that our syntactic computations of serial composition are correct, i.e., that the resulting atomic component $\mathbf{serial}(C, C')$ is semantically equivalent to the original composite component $C\,; C'$:

Theorem 8. *If C and C' are two atomic components, then*

$$C\,; C' \equiv \mathbf{serial}(C, C').$$

Example 18. Consider the following STS components:

$$\begin{array}{rcl} C_3 & = & \texttt{stateless}(u, (x, y), \texttt{true}) \\ C_4 & = & \texttt{stateless\_det}((x, y), y \neq 0, \dfrac{x}{y}) \end{array}$$

Then:

$$\begin{array}{rcl} \mathbf{serial}(C_3, C_4) & = & \mathbf{serial}(C_3, \texttt{stateless\_det2stateless}(C_4)) \\ & = & \mathbf{serial}(\texttt{stateless}(u, (x, y), \texttt{true}), \texttt{stateless}((x, y), z, y \neq 0 \wedge z = \dfrac{x}{y})) \\ & = & \texttt{stateless}(u, z, (\forall x, y : \texttt{true} \Rightarrow (\exists z : y \neq 0 \wedge z = \dfrac{x}{y})) \\ & & \quad \wedge\ (\exists x, y : \texttt{true} \wedge y \neq 0 \wedge z = \dfrac{x}{y}))) \\ & \equiv & \texttt{stateless}(u, z, \texttt{false}) \end{array}$$

As we can see, the composition results in a stateless STS component with input-output formula `false`. The semantics of such a component is Fail, indicating that C_3 and C_4 are *incompatible*. Indeed, in the case of $C_3\,; C_4$, the issue is that C_4 requires its second input, y, to be non-zero, but C_3 cannot guarantee that. The reason is that the input-output formula of C_3 is `true`, meaning that, no matter what its input u is, C_3 may output *any* value for x and y, non-deterministically. This violates the input requirements of C_4, causing an incompatibility. We will return to this point in §5.8. We also note that this type of incompatibility is impossible to prevent, by controlling the input u. In the example that follows, we see a case where the two components are *not* incompatible, because the input requirements of the downstream component can be met by strengthening the input assumptions of the upstream component:

Example 19. Consider the following QLTL components:

$$\begin{array}{rcl} C_5 & = & \texttt{qltl}(x, y, \mathbf{G}\,(x \Rightarrow \mathbf{F}\,y)) \\ C_6 & = & \texttt{qltl}(y, (), \mathbf{G}\,\mathbf{F}\,y) \end{array}$$

Then:

$$\begin{array}{rcl} \mathbf{serial}(C_5, C_6) & = & \mathbf{serial}(\texttt{qltl}(x, y, \mathbf{G}\,(x \Rightarrow \mathbf{F}\,y)), \texttt{qltl}(y, (), \mathbf{G}\,\mathbf{F}\,y)) \\ & = & \texttt{qltl}(x, (), (\forall y : (\mathbf{G}\,(x \Rightarrow \mathbf{F}\,y)) \Rightarrow \mathbf{G}\,\mathbf{F}\,y) \wedge (\exists y : \mathbf{G}\,(x \Rightarrow \mathbf{F}\,y) \wedge \mathbf{G}\,\mathbf{F}\,y)) \\ & \equiv & \texttt{qltl}(x, (), \mathbf{G}\,\mathbf{F}\,x) \end{array}$$

In this example, the downstream component C_6 requires its input y to be infinitely often true ($\mathbf{G}\,\mathbf{F}\,y$). This can be achieved only if the input x of the upstream component is infinitely often true, which is the condition derived by the serial composition of C_5 and C_6 ($\mathbf{G}\,\mathbf{F}\,x$). Notice that C_5 does not impose any a-priori requirements on its input. However, its input-output relation is the so-called *request-response property* which can be expressed as: *whenever the input x is true, the output y will eventually become true afterwards* ($\mathbf{G}\,(x \Rightarrow \mathbf{F}\,y)$). This request-response property implies that in order for y to be infinitely-often true, x must be infinitely-often true. Moreover, this is the weakest possible condition that can be enforced on x in order to guarantee that the condition on y holds.

5.4 Syntactic Computation of Parallel Composition

Given a composite component C formed as the parallel composition of two atomic components A and B, i.e., $C = A \parallel B$, we would like to compute a new atomic component C_a, such that C_a is semantically equivalent to C. To compute C_a, we introduce the syntactic transformation operator **parallel**, so that $C_a := \textbf{parallel}(A, B)$. The definition of **parallel** follows the same pattern as the one for **serial** (§5.3).

5.4.1 Syntactic Parallel Composition of Two QLTL Components

Let $C = \texttt{qltl}(x, y, \varphi)$ and $C' = \texttt{qltl}(u, v, \varphi')$. Then their *syntactic parallel composition*, denoted $\textbf{parallel}(C, C')$, is the QLTL component defined by

$$\textbf{parallel}(C, C') = \texttt{qltl}\big((x, u), (y, v), \varphi \wedge \varphi'\big) \tag{24}$$

In the above definition we assume that variable names x, y, u, v are all distinct. If this is not the case, then we rename variables as appropriately.

5.4.2 Syntactic Parallel Composition of Two General STS Components

Let $C = \texttt{sts}(x, y, s, \mathit{init}, \mathit{trs})$ and $C' = \texttt{sts}(u, v, t, \mathit{init}', \mathit{trs}')$. Then:

$$\textbf{parallel}(C, C') = \texttt{sts}\big((x, u), (y, v), (s, t), \mathit{init} \wedge \mathit{init}', \mathit{trs} \wedge \mathit{trs}'\big) \tag{25}$$

5.4.3 Syntactic Parallel Composition of Two Stateless STS Components

Let $C = \texttt{stateless}(x, y, \mathit{trs})$ and $C' = \texttt{stateless}(u, v, \mathit{trs}')$. Then

$$\textbf{parallel}(C, C') = \texttt{stateless}\big((x, u), (y, v), \mathit{trs} \wedge \mathit{trs}'\big) \tag{26}$$

5.4.4 Syntactic Parallel Composition of Two Deterministic STS Components

Let $C = \texttt{det}(x, s, a, p, \mathit{next}, \mathit{out})$ and $C' = \texttt{det}(u, t, b, p', \mathit{next}', \mathit{out}')$. Then:

$$\textbf{parallel}(C, C') = \texttt{det}\big((x, u), (s, t), (a, b), p \wedge p', (\mathit{next}, \mathit{next}'), (\mathit{out}, \mathit{out}')\big) \tag{27}$$

5.4.5 Syntactic Parallel Composition of Two Stateless Deterministic STS Components

Let $C = \texttt{stateless\_det}(x, p, \mathit{out})$ and $C' = \texttt{stateless\_det}(u, p', \mathit{out}')$. Then:

$$\textbf{parallel}(C, C') = \texttt{stateless\_det}\big((x, u), p \wedge p', (\mathit{out}, \mathit{out}')\big) \tag{28}$$

5.4.6 Syntactic Parallel Composition of Two Arbitrary Atomic Components

Similar to the syntactic serial composition, we define the syntactic parallel composition of two atomic components C and C' by using the mappings of less general components to more general components (Fig. 8), as appropriate. Formally, assume that $\mathit{atm}, \mathit{atm}' \in \{\texttt{qltl}, \texttt{sts}, \texttt{stateless}, \texttt{det}, \texttt{stateless\_det}\}$ are the types of the components C and C', and $\mathit{common} = \mathit{atm} \vee \mathit{atm}'$ is the least general component type that is more general than atm and atm' as defined in Fig. 8. Then

$$\textbf{parallel}(C, C') = \textbf{parallel}(\mathit{atm}2\mathit{common}(C), \mathit{atm}'2\mathit{common}(C')) \tag{29}$$

5.4.7 Correctness of Syntactic Parallel Composition

The following theorem demonstrates that our syntactic computations of parallel composition are also correct, i.e., that the resulting atomic component $\textbf{parallel}(C, C')$ is semantically equivalent to the original composite component $C \parallel C'$:

Theorem 9. *If C and C' are two atomic components, then*

$$C \parallel C' \equiv \textbf{parallel}(C, C').$$

5.5 Syntactic Computation of Feedback Composition for Decomposable Deterministic STS Components

5.5.1 Decomposable Components

We provide a syntactic closed-form expression for the feedback composition of a deterministic STS component, provided such a component is *decomposable*. Intuitively, decomposability captures the fact that the first output of the component, y_1, does not depend on its first input, x_1. This ensures that the feedback composition (which connects y_1 to x_1) does not introduce any circular dependencies.

Definition 22 (Decomposability). *Let C be a deterministic STS component* $\texttt{det}\big((x_1,\dots,x_n), s, a, p, next, (e_1,\dots,e_m)\big)$ *or a stateless deterministic STS component* $\texttt{stateless\_det}\big((x_1,\dots,x_n), p, (e_1,\dots,e_m)\big)$. *$C$ is called* decomposable *if x_1 is not free in e_1.*

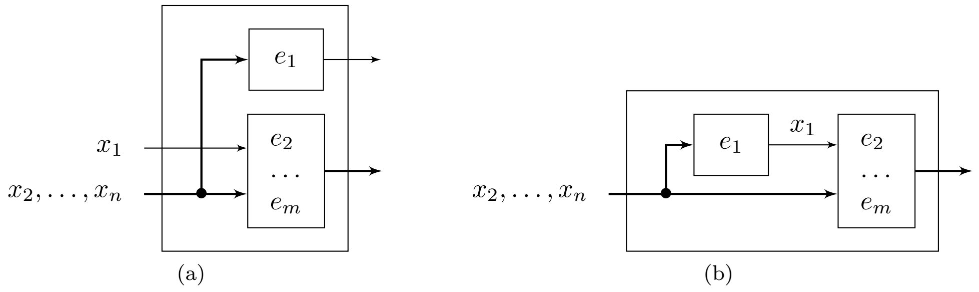


Figure 9: (a) Decomposable deterministic component; (b) the same component after applying feedback, connecting its first output to x_1.

Decomposability is illustrated in Fig. 9a. The figure shows that expression e_1 depends only on inputs $x_2,\dots,x_n$.

5.5.2 Syntactic Feedback of a Decomposable Deterministic STS Component

For a decomposable deterministic STS component $C = \texttt{det}((x_1,\dots,x_n), s, a, p, next, (e_1,\dots,e_m))$, its *syntactic feedback composition*, denoted **feedback**(C), is the deterministic STS component defined by

$$\textbf{feedback}(C) = \texttt{det}\big((x_2,\dots,x_n), s, a, p[x_1 := e_1], next[x_1 := e_1], (e_2[x_1 := e_1],\dots,e_m[x_1 := e_1])\big) \quad (30)$$

Thus, computing feedback syntactically consists in removing the first input of the component and replacing the corresponding variable x_1 by the expression of the first output, e_1, everywhere where x_1 appears. The **feedback** operator is illustrated in Fig. 9b.

5.5.3 Syntactic Feedback of a Decomposable Stateless Deterministic STS Component

For a decomposable stateless deterministic STS component $C = \texttt{stateless\_det}((x_1,\dots,x_n), p, (e_1,\dots,e_m))$, **feedback**$(C)$ is the stateless deterministic STS component defined by

$$\textbf{feedback}(C) = \texttt{stateless\_det}\big((x_2,\dots,x_n), p[x_1 := e_1], (e_2[x_1 := e_1],\dots,e_m[x_1 := e_1])\big) \quad (31)$$

5.5.4 Correctness of Syntactic Feedback Composition

Theorem 10. *If C is a decomposable deterministic STS component, then*

$$\texttt{fdbk}(C) \equiv \textbf{feedback}(C).$$

Providing closed-form syntactic computations of feedback composition for general components, including possibly non-deterministic STS and QLTL components, is an open problem, beyond the scope of the current paper. We remark that the straightforward idea of adding to the contract the equality constraint $x = y$ where y is the output connected in feedback to input x, does not work.[3]

In fact, even obtaining a semantically consistent compositional definition of feedback for non-deterministic and non-input-receptive systems is a challenging problem [73]. Nevertheless, the results that we provide here are sufficient to cover the majority of cases in practice. In particular, the operator **feedback** can be used to handle Simulink diagrams, provided these diagrams do not contain *algebraic loops*, i.e., circular and instantaneous dependencies (see §5.7).

5.6 Closure Properties of MPT Subclasses w.r.t. Composition Operators

In addition to providing symbolic computation procedures, the results of the above subsections also prove closure properties of the various MPT subclasses of RCRS with respect to the three composition operators. These closure properties are summarized in Tables 1 and 2.

In a nutshell, both serial and parallel composition preserve the most general type of the composed components, according to the lattice in Fig. 8. For instance, the serial (or parallel) composition of two stateless STS components is a stateless STS component; the serial (or parallel) composition of a stateless STS component and a general STS component is a general STS component; and so on. Feedback preserves the type of its component (deterministic or stateless deterministic).

; and ‖	`qltl`	`sts`	`stateless`	`det`	`stateless_det`
`qltl`	`qltl`	`qltl`	`qltl`	`qltl`	`qltl`
`sts`	`qltl`	`sts`	`sts`	`sts`	`sts`
`stateless`	`qltl`	`sts`	`stateless`	`sts`	`stateless`
`det`	`qltl`	`sts`	`sts`	`det`	`det`
`stateless_det`	`qltl`	`sts`	`stateless`	`det`	`stateless_det`

Table 1: Closure properties of serial and parallel compositions. The table is to be read as follows: given atomic components C_1, C_2 of types as specified in a row/column pair, the serial or parallel composition of C_1 and C_2 is an atomic component of type as specified in the corresponding table entry.

`fdbk`	`det` and decomposable	`stateless_det` and decomposable
	`det`	`stateless_det`

Table 2: Closure properties of feedback composition.

5.7 Syntactic Simplification of Arbitrary Composite Components

The results of the previous subsections show how to simplify into an atomic component the serial or parallel composition of two atomic components, or the feedback composition of an atomic decomposable component. We can combine these techniques in order to provide a general syntactic simplification algorithm: the algorithm takes as input an arbitrarily complex composite component, and returns an equivalent atomic component. The algorithm is shown in Fig. 10.

The algorithm fails only in case it encounters the feedback of a non-decomposable component. Recall that decomposability implies determinism (c.f. §5.5.1), which means that the test C'' *is decomposable* means

[3]One of several problems of this definition is that it does not preserve refinement. For example, the stateless component with contract $x \neq y$ refines the stateless component with contract `true`. Adding the constraint $x = y$ to both contracts yields the components with contracts $x = y$ and `false`, respectively, where the latter no longer refines the former. For a more detailed discussion, see [73].

```
$\mathbf{atomic}(C)$ :
    if $C$ is atomic then
        return $C$
    else if $C$ is $C' \,;C''$ then
        return $\mathbf{serial}(\mathbf{atomic}(C'), \mathbf{atomic}(C''))$
    else if $C$ is $C' \parallel C''$ then
        return $\mathbf{parallel}(\mathbf{atomic}(C'), \mathbf{atomic}(C''))$
    else if $C$ is $\texttt{fdbk}(C')$ then
        $C'' := \mathbf{atomic}(C')$
        if $C''$ is decomposable then
            return $\mathbf{feedback}(C'')$
        else
            fail
    else /* impossible by definition of syntax */
        fail
```

Figure 10: Simplification algorithm for arbitrary composite components.

that C'' is of the form $\texttt{det}((x_1, \ldots), s, a, p, next, (e_1, \ldots))$ or $\texttt{stateless\_det}((x_1, \ldots), p, (e_1, \ldots))$ and x_1 is not free in e_1.

We note that in practice, our RCRS implementation on top of Isabelle performs more simplifications in addition to those performed by the procedure **atomic**. For instance, our implementation may be able to simplify a logical formula ϕ into an equivalent but simpler formula ϕ' (e.g., by eliminating quantifiers from ϕ), and consequently also simplify a component, say, $\texttt{qltl}(x, y, \phi)$ into an equivalent but simpler component $\texttt{qltl}(x, y, \phi')$. These simplifications very much depend on the logic used in the components. Describing the simplifications that our implementation performs is outside the scope of the current paper, as it belongs in the realm of computational logic. It suffices to say that our tool is not optimized for this purpose, and could leverage specialized tools and relevant advances in the field of computational logic.

5.7.1 Deterministic and Algebraic Loop Free Composite Components

In order to state and prove correctness of the algorithm, we extend the notion of determinism to a composite component. We also introduce the notion of *algebraic loop free* components, which capture systems with no circular and instantaneous input-output dependencies.

A (possibly composite) component C is said to be *deterministic* if every atomic component of C is either a deterministic STS component or a stateless deterministic STS component. Formally, C is deterministic iff $\mathbf{determ}(C)$ is true, where $\mathbf{determ}(C)$ is defined inductively on the structure of C:

$$\begin{array}{lcl}
\mathbf{determ}(\texttt{det}(x, s, a, p, n, e)) & = & \texttt{true} \\
\mathbf{determ}(\texttt{stateless\_det}(x, p, e)) & = & \texttt{true} \\
\mathbf{determ}(\texttt{sts}(x, y, s, init, trs)) & = & \texttt{false} \\
\mathbf{determ}(\texttt{stateless}(x, y, trs)) & = & \texttt{false} \\
\mathbf{determ}(C \,; C') & = & \mathbf{determ}(C) \wedge \mathbf{determ}(C') \\
\mathbf{determ}(C \parallel C') & = & \mathbf{determ}(C) \wedge \mathbf{determ}(C') \\
\mathbf{determ}(\texttt{fdbk}(C)) & = & \mathbf{determ}(C)
\end{array}$$

Notice that this notion of determinism applies to a generally *composite* component C, i.e., a syntactic term in our algebra of components, involving atomic components possibly composed via the three composition operators. This notion of determinism is the generalization of the syntactically deterministic STS components, which are atomic. This notion of determinism is also distinct from any *semantic* notion of determinism (which we have not introduced at all in this paper, as it is not needed).

For a deterministic component we define its *output input dependency relation*. Let C be deterministic, and let $\Sigma_{in}(C) = X_1 \times \ldots \times X_n$ and $\Sigma_{out}(C) = Y_1 \times \ldots \times Y_m$. The relation $\mathbf{OI}(C) \subseteq \{1,\ldots,m\} \times \{1,\ldots,n\}$ is defined inductively on the structure of C:

$$\begin{array}{lcl}
\mathbf{OI}(\mathtt{det}((x_1,\ldots,x_n),s,a,p,next,(e_1,\ldots,e_m))) & = & \{(i,j) \mid x_j \text{ is free in } e_i\} \\
\mathbf{OI}(\mathtt{stateless\_det}((x_1,\ldots,x_n),p,(e_1,\ldots,e_m))) & = & \{(i,j) \mid x_j \text{ is free in } e_i\} \\
\mathbf{OI}(C\,;C') & = & \mathbf{OI}(C') \circ \mathbf{OI}(C) \\
\mathbf{OI}(C \parallel C') & = & \mathbf{OI}(C) \cup \{(i+m, j+n) \mid (i,j) \in \mathbf{OI}(C')\} \\
 & & \text{where } \Sigma_{in}(C) = X_1 \times \ldots \times X_n \\
 & & \text{and } \Sigma_{out}(C) = Y_1 \times \ldots \times Y_m \\
\mathbf{OI}(\mathtt{fdbk}(C)) & = & \{(i,j) \mid i > 0 \wedge j > 0 \wedge ((i+1,j+1) \in \mathbf{OI}(C) \\
 & & \quad \vee ((i+1,1) \in \mathbf{OI}(C) \wedge (1,j+1) \in \mathbf{OI}(C)))\}
\end{array}$$

The intuition is that $(i,j) \in \mathbf{OI}(C)$ iff the i-th output of C depends on its j-th input.

The **OI** relation is preserved by the syntactic operations, as shown by the following lemma:

Lemma 10. *If C and C' are deterministic STS components, then*

$$\begin{array}{rcl}
\mathbf{OI}(C\,;C') & = & \mathbf{OI}(\mathbf{serial}(C,C')) \\
\mathbf{OI}(C \parallel C') & = & \mathbf{OI}(\mathbf{parallel}(C,C')).
\end{array}$$

If C is also decomposable, then

$$\mathbf{OI}(\mathtt{fdbk}(C)) \quad = \quad \mathbf{OI}(\mathbf{feedback}(C)).$$

We introduce now the the notion of algebraic loop free component. Intuitively, a (possibly composite) deterministic component C is algebraic loop free if, whenever C contains a subterm of the form $\mathtt{fdbk}(C')$, the first output of C' does not depend on its first input. This implies that whenever a feedback connection is formed, no circular dependency is introduced. It also ensures that the simplification algorithm will never fail. Formally, for a component C such that $\mathbf{determ}(C)$ is true, $\mathbf{loop\text{-}free}(C)$ is defined inductively on the structure of C:

$$\begin{array}{lcl}
\mathbf{loop\text{-}free}(C) & = & \mathtt{true} \text{ if } C = \mathtt{det}(x,s,a,p,next,out) \\
\mathbf{loop\text{-}free}(C) & = & \mathtt{true} \text{ if } C = \mathtt{stateless\_det}(x,p,out) \\
\mathbf{loop\text{-}free}(C\,;C') & = & \mathbf{loop\text{-}free}(C) \wedge \mathbf{loop\text{-}free}(C') \\
\mathbf{loop\text{-}free}(C \parallel C') & = & \mathbf{loop\text{-}free}(C) \wedge \mathbf{loop\text{-}free}(C') \\
\mathbf{loop\text{-}free}(\mathtt{fdbk}(C)) & = & \mathbf{loop\text{-}free}(C) \wedge (1,1) \notin \mathbf{OI}(C)
\end{array}$$

That is, deterministic atomic components are by definition algebraic loop free. A serial composite component $C\,;C'$ is algebraic loop free if both C and C' are algebraic loop free, and similarly for a parallel composite component $C \parallel C'$. A feedback composite component $\mathtt{fdbk}(C)$ is algebraic loop free if C is algebraic loop free and the first output of C does not depend on its first input.

5.7.2 Correctness of the Simplification Algorithm

Theorem 11. *Let C be a (possibly composite) component.*

1. *If C does not contain any* `fdbk` *operators then* $\mathbf{atomic}(C)$ *does not fail and returns an atomic component such that* $\mathbf{atomic}(C) \equiv C$.

2. *If* $\mathbf{determ}(C) \wedge \mathbf{loop\text{-}free}(C)$ *is true then* $\mathbf{atomic}(C)$ *does not fail and returns an atomic component such that* $\mathbf{atomic}(C) \equiv C$. *Moreover,* $\mathbf{atomic}(C)$ *is a deterministic STS component and*

$$\mathbf{OI}(C) = \mathbf{OI}(\mathbf{atomic}(C)).$$

Proof. The first part of the theorem is a consequence of the fact that the syntactic serial and parallel compositions are defined for all atomic components and return equivalent atomic components, by Theorems 8 and 9.

For the second part, since we have a recursive procedure, we prove its correctness by assuming the correctness of the recursive calls. Additionally, the termination of this procedure is ensured by the fact that all recursive calls are made on "smaller" components. Specifically: we assume that both $\mathbf{determ}(C)$ and $\mathbf{loop\text{-}free}(C)$ hold; and we prove that $\mathbf{atomic}(C)$ does not fail, $\mathbf{atomic}(C)$ is a deterministic STS component, $\mathbf{atomic}(C) \equiv C$, and $\mathbf{OI}(C) = \mathbf{OI}(\mathbf{atomic}(C))$.

We only consider the case $C = \mathtt{fdbk}(C')$. All other cases are similar, but simpler. Because $\mathbf{determ}(C)$ and $\mathbf{loop\text{-}free}(C)$ hold, we have that $\mathbf{determ}(C')$ and $\mathbf{loop\text{-}free}(C')$ also hold, and in addition $(1,1) \notin \mathbf{OI}(C')$. Using the correctness assumption for the recursive call we have that $\mathbf{atomic}(C')$ does not fail, $C'' = \mathbf{atomic}(C')$ is a deterministic STS component, $C'' \equiv C'$, and $\mathbf{OI}(C'') = \mathbf{OI}(C')$.

Because C'' is a deterministic STS component and $(1,1) \notin \mathbf{OI}(C') = \mathbf{OI}(C'')$, C'' is decomposable. From this we have that $D := \mathbf{feedback}(C'')$ is defined. Therefore, $\mathbf{atomic}(C)$ returns D and does not fail. It remains to show that D has the desired properties. By the definition of $\mathbf{feedback}(C'')$ and the fact that C'' is a decomposable deterministic STS component, D is also a deterministic STS component. We also have:

$$D = \mathbf{feedback}(C'') \equiv \mathtt{fdbk}(C'') \equiv \mathtt{fdbk}(C') = C$$

where $\mathbf{feedback}(C'') \equiv \mathtt{fdbk}(C'')$ follows from Theorem 10 and $\mathtt{fdbk}(C'') \equiv \mathtt{fdbk}(C')$ follows from $C'' \equiv C'$ and the semantics of $\mathtt{fdbk}$.

Finally, using Lemma 10 and $\mathbf{OI}(C'') = \mathbf{OI}(C')$, we have

$$\mathbf{OI}(D) = \mathbf{OI}(\mathbf{feedback}(C'')) = \mathbf{OI}(\mathtt{fdbk}(C'')) = \mathbf{OI}(\mathtt{fdbk}(C')) = \mathbf{OI}(C)$$

□

Corollary 2. *If a component C does not contain any* $\mathtt{fdbk}$ *operators or if* $\mathbf{determ}(C) \wedge \mathbf{loop\text{-}free}(C)$ *is true, then* $[\![C]\!]$ *is a GPT.*

Note that condition $\mathbf{determ}(C) \wedge \mathbf{loop\text{-}free}(C)$ is sufficient, but not necessary, for $[\![C]\!]$ to be a GPT. For example:

Example 20. Consider the following components:

$$\begin{array}{lcl}
\mathtt{Const}_{\mathtt{false}} & = & \mathtt{stateless\_det}\big(x, \mathtt{true}, \mathtt{false}\big) \\
\mathtt{And} & = & \mathtt{stateless\_det}\big((x,y), \mathtt{true}, (x \wedge y, x \wedge y)\big) \\
C & = & \mathtt{Const}_{\mathtt{false}} \,;\mathtt{fdbk}(\mathtt{And})
\end{array}$$

$\mathtt{Const}_{\mathtt{false}}$ outputs the constant $\mathtt{false}$. $\mathtt{And}$ is a version of logical and with two identical outputs (we need two copies of the output, because one will be eliminated once we apply feedback). C is a composite component, formed by first connecting the first output of $\mathtt{And}$ in feedback to its first input, and then connecting the output of $\mathtt{Const}_{\mathtt{false}}$ to the second input of $\mathtt{And}$ (in reality, to the only remaining input of $\mathtt{fdbk}(\mathtt{And})$). Observe that C has algebraic loops, that is, $\mathbf{loop\text{-}free}(C)$ does not hold. Yet it can be shown that $[\![C]\!]$ is a GPT (in particular, we can show that $C \equiv \mathtt{Const}_{\mathtt{false}}$). Handling cases like this is beyond the scope of this paper and part of future work.

Example 21. The simplification algorithm applied to the component from Fig. 2 results in

$$\mathbf{atomic}(\mathtt{Sum}) = \mathbf{atomic}(\mathtt{fdbk}(\mathtt{Add}\,;\mathtt{UnitDelay}\,;\mathtt{Split})) = \mathtt{det}(y, s, 0, s+y, s).$$

To see how the above is derived, let us first calculate **atomic**(`Add` ; `UnitDelay` ; `Split`):

$$
\begin{array}{rl}
& \textbf{atomic}(\texttt{Add}\,;\texttt{UnitDelay}\,;\texttt{Split}) \\
= & \{\text{Definition of } \textbf{atomic}\} \\
& \textbf{serial}(\textbf{atomic}(\texttt{Add}\,;\texttt{UnitDelay}),\ \texttt{Split}) \\
= & \{\text{Definition of } \textbf{atomic}\} \\
& \textbf{serial}(\textbf{serial}(\texttt{Add},\ \texttt{UnitDelay}),\ \texttt{Split}) \\
= & \{\text{Expanding } \texttt{Add} \text{ and } \texttt{UnitDelay} \text{ and choosing suitable variable names}\} \\
& \textbf{serial}(\textbf{serial}(\texttt{stateless\_det}((x,y),\texttt{true},x+y),\ \texttt{det}(z,s,0,\texttt{true},z,s)),\ \texttt{Split}) \\
= & \{\text{Replacing } \texttt{stateless\_det}((x,y),\texttt{true},x+y) \text{ by } \texttt{det}((x,y),(),(),\texttt{true},(),x+y) \text{ according to (1)}\} \\
& \textbf{serial}(\textbf{serial}(\texttt{det}((x,y),(),(),\texttt{true},(),x+y),\ \texttt{det}(z,s,0,\texttt{true},z,s)),\ \texttt{Split}) \\
= & \{\text{Syntactic computation of inner } \textbf{serial} \text{ according to (21)}\} \\
& \textbf{serial}(\texttt{det}((x,y),s,0,\texttt{true},x+y,s),\ \texttt{Split}) \\
= & \{\text{Expanding } \texttt{Split} \text{ and choosing suitable variable names}\} \\
& \textbf{serial}(\texttt{det}((x,y),s,0,\texttt{true},x+y,s),\ \texttt{stateless\_det}(z,\texttt{true},(z,z))) \\
= & \{\text{Replacing } \texttt{stateless\_det}(z,\texttt{true},(z,z)) \text{ by } \texttt{det}(z,(),(),\texttt{true},(),(z,z)) \text{ according to (1)}\} \\
& \textbf{serial}(\texttt{det}((x,y),s,0,\texttt{true},x+y,s),\ \texttt{det}(z,(),(),\texttt{true},(),(z,z))) \\
= & \{\text{Syntactic computation of remaining } \textbf{serial} \text{ according to (21)}\} \\
& \texttt{det}((x,y),s,0,\texttt{true},x+y,(s,s))
\end{array}
$$

The result $\texttt{det}((x,y),s,0,\texttt{true},x+y,(s,s))$ is a deterministic and input-receptive component with two input variables x, y, one state variable s, initial state $s = 0$, next state $s' = x + y$, and two outputs which are both equal to s. This component corresponds precisely to the one illustrated in Figure 3. This component is decomposable according to Definition 22, because its first input variable x is not free (in fact, it does not appear at all) in its first output expression s. Because $\texttt{det}((x,y),s,0,\texttt{true},x+y,(s,s))$ is decomposable, we can now calculate $\textbf{atomic}(\texttt{fdbk}(\texttt{det}((x,y),s,0,\texttt{true},x+y,(s,s))))$ by expanding the definition of **atomic** as follows:

$$
\begin{array}{rl}
& \textbf{atomic}(\texttt{fdbk}(\texttt{det}((x,y),s,0,\texttt{true},x+y,(s,s)))) \\
= & \{\text{Definition of } \textbf{atomic},\ \texttt{det}((x,y),s,0,\texttt{true},x+y,(s,s)) \text{ is decomposable}\} \\
& \textbf{feedback}(\texttt{det}((x,y),s,0,\texttt{true},x+y,(s,s))) \\
= & \{\text{Syntactic computation of } \textbf{feedback} \text{ according to (30)}\} \\
& \texttt{det}(y,s,0,\texttt{true},s+y,s)
\end{array}
$$

This example is available in the public distribution of RCRS in the theory file `RCRS_Demo.thy`, which can also be accessed at http://rcrs.gitlab.io/theories/RCRS/RCRS-All/RCRS_Demo.html.

5.8 Checking Validity and Compatibility

Recall Example 18 given in §5.3.7, of the serial composition of components C_3 and C_4, resulting in a component with input-output relation `false`, implying that $[\![C_3\,;C_4]\!] = \mathsf{Fail}$. When this occurs, we say that C_3 and C_4 are *incompatible*. We would like to catch such incompatibilities. This amounts to first simplifying the serial composition $C_3\,;C_4$ into an atomic component C, and then checking whether $[\![C]\!] = \mathsf{Fail}$.

In general, we say that a component C is *valid* if $[\![C]\!] \neq \mathsf{Fail}$. Given a component C, we can check whether it is valid, as follows. First, we simplify C to obtain an atomic component $C' = \textbf{atomic}(C)$. If C' is a QLTL component of the form $\texttt{qltl}(x,y,\varphi)$ then C' is valid iff φ is satisfiable. The same is true if C' is a stateless STS component of the form $\mathtt{stateless}(x, y, \varphi)$. If C' is a general STS component then we can first transform it into a QLTL component and check satisfiability of the resulting QLTL formula.

Theorem 12. *If C is an atomic component of the form $\mathtt{qltl}(x, y, \varphi)$ or $\mathtt{stateless}(x, y, \varphi)$, then $[\![C]\!] \neq \mathsf{Fail}$ iff φ is satisfiable.*

As mentioned in the introduction, one of the goals of RCRS is to function as a behavioral type system for reactive system modeling frameworks such as Simulink. In the RCRS setting, type checking consists in checking properties such as compatibility of components, as in $C_3 \,;C_4$ of Example 18 given in §5.3.7. When components are compatible, computing new (stronger) input preconditions like those for $C_5 \,;C_6$ of Example 19 given in §5.3.7 can be seen as behavioral type inference. Indeed, the new derived condition $\mathbf{G}\,\mathbf{F}\,x$ in the above example can be seen as an inferred type of the composite component $C_5 \,;C_6$.

We note that the decidability of the satisfiability question for φ depends on the logic used and the domains of the variables in the formula. For instance, although φ can be a QLTL formula, if it restricts the set of constants to $\mathtt{true}$, has no functional symbols, and only equality as predicate symbol, then it is equivalent to a QPTL formula,[4] for which we can use available techniques [82].

5.9 Checking Input-Receptiveness and Computing Legal Inputs Symbolically

Given a component C, we often want to check whether it is *input-receptive*, i.e., whether $\mathsf{legal}([\![C]\!]) = \top$, or equivalently, $[\![C]\!](\top) = \top$. More generally, we may want to compute the legal input values for C, which is akin to type inference as discussed above. To do this, we will provide a symbolic method to compute $\mathsf{legal}([\![C]\!])$ as a formula $\mathbf{legal}(C)$. Then, checking that C is input-receptive amounts to checking that the formula $\mathbf{legal}(C)$ is valid, or equivalently, checking that $\neg\mathbf{legal}(C)$ is unsatisfiable. Note that this is also showing how to automate domain/precondition calculations. We assume that C is atomic (otherwise, we first simplify C using the algorithm of §5.7).

Definition 23. *Given an atomic component C, we define $\mathbf{legal}(C)$, a formula characterizing the legal inputs of C. $\mathbf{legal}(C)$ is defined based on the type of C:*

$$\begin{aligned}
\mathbf{legal}\big(\mathtt{qltl}(x, y, \varphi)\big) &= (\exists y : \varphi) && (32)\\
\mathbf{legal}\big(\mathtt{sts}(x, y, s, init, trs)\big) &= (\forall s, y : init \Rightarrow (r_1 \,\mathbf{L}\, r_2)) && (33)\\
\mathbf{legal}\big(\mathtt{stateless}(x, y, trs)\big) &= \mathbf{G}\,(\exists y : trs) && (34)\\
\mathbf{legal}\big(\mathtt{det}(x, s, a, p, next, out)\big) &= (\forall s, y : s = a \Rightarrow (r_3 \,\mathbf{L}\, p)) && (35)\\
\mathbf{legal}\big(\mathtt{stateless\_det}(x, p, out)\big) &= \mathbf{G}\, p && (36)
\end{aligned}$$

where $r_1 = trs[s' := \bigcirc\, s]$, $r_2 = (\exists s', y : trs)$, and $r_3 = (\bigcirc\, s = next \wedge y = out)$.

(32) states that the legal input traces of a QLTL component $\mathtt{qltl}(x, y, \varphi)$ are characterized by the QLTL formula $\exists y : \phi$. The latter formula is satisfied by all input traces σ_x over x for which there exists output trace σ_y over y such that $(\sigma_x, \sigma_y) \models \varphi$. This characterization follows directly from Lemma 9.

Similarly, (33) provides the QLTL formula $\forall s, y : init \Rightarrow (r_1 \,\mathbf{L}\, r_2)$ characterizing the legal input traces of an STS component $\mathtt{sts}(x, y, s, init, trs)$. This formula is satisfied by an input trace σ_x over x iff for any state trace σ_s over s and output trace σ_y over y, if $init$ is satisfied (i.e., if the system starts in a legal initial state) then $r_1 \,\mathbf{L}\, r_2$ is satisfied. r_1 expands into $trs[s' := \bigcirc\, s]$ and characterizes the transition relation trs written in QLTL syntax (we have to replace the next-state variable notation s' in trs with $\bigcirc\, s$ which denotes the same thing in QLTL). r_2 expands into $\exists s', y : trs$ and characterizes all the (x, s) (input, current state) pairs for which there exist next state s' and output y such that the transition relation trs is satisfied. In other words, input trace σ_x is legal if regardless of the nondeterministic choices that the system makes, it does not get stuck (i.e., it can always continue by making one more step).

The next theorem shows that $\mathbf{legal}$ correctly characterizes the semantic predicate legal:

[4] For example, the atomic QLTL formula $\bigcirc\bigcirc\, x = \bigcirc\, y$ can be translated into the LTL formula $\mathbf{X}\,\mathbf{X}\,x \Leftrightarrow \mathbf{X}\,y$, and the formula $\bigcirc\bigcirc\bigcirc\, x = \mathtt{true}$ into $\mathbf{X}\,\mathbf{X}\,\mathbf{X}\,x$.

Theorem 13. *If C is an atomic component , then*

$$\mathsf{legal}(\llbracket C \rrbracket) = \{\sigma_x \mid \sigma_x \models \mathbf{legal}(C)\}.$$

It follows from Theorem 13 that a component C is input-receptive iff the formula $\mathbf{legal}(C)$ is valid.

5.10 Checking Refinement Symbolically

We end this section by showing how to check whether a component refines another component. Again, we will assume that the components in question are atomic (if not, they can be simplified using the **atomic** procedure).

Theorem 14. *Let $C_1 = \mathtt{sts}(x, y, s, init, r_1)$, $C_1' = \mathtt{sts}(x, y, s, init', r_1')$, $C_2 = \mathtt{stateless}(x, y, r_2)$, $C_2' = \mathtt{stateless}(x, y, r_2')$, $C_3 = \mathtt{qltl}(x, y, \varphi)$, and $C_3' = \mathtt{qltl}(x, y, \varphi')$. Then:*

1. *C_1 is refined by C_1' if the formula*
$$(init' \Rightarrow init) \land ((\exists s', y : r_1) \Rightarrow (\exists s', y : r_1')) \land ((\exists s', y : r_1) \land r_1' \Rightarrow r) \tag{37}$$
is valid.

2. *C_2 is refined by C_2' if and only if the formula*
$$\big((\exists y : r_2) \Rightarrow (\exists y : r_2')\big) \land \big((\exists y : r_2) \land r_2' \Rightarrow r_2\big) \tag{38}$$
is valid.

3. *C_3 is refined by C_3' if and only if the formula*
$$\big((\exists y : \varphi) \Rightarrow (\exists y : \varphi')\big) \land \big(((\exists y : \varphi) \land \varphi') \Rightarrow \varphi\big) \tag{39}$$
is valid.

As the above theorem shows, checking refinement amounts to checking validity (or equivalently, satisfiability of the negation) of first-order formulas formed by the various symbolic expressions in the component specifications. The exact logic of these formulas depends on the logics used by the components. For example, if C_3 and C_3' both use quantifier-free LTL for ϕ and ϕ', then in order to check refinement we need to check satisfiability of a first-order QLTL formula.

Specifically, Theorem 14 states that checking that a stateless component C_2' refines another stateless component C_2 is equivalent to checking that the input condition of C_2 is stronger than that of C_2', and that the input-output relation of C_2', restricted to the legal inputs of C_2, is stronger than the input-output relation of C_2. This result follows from Corollary 1 and the fact that stateless components are GPTs. The same is true for QLTL components C_3 and C_3'. For STS components C_1 and C_1' a similar property holds, with the additional condition that the initial state predicate of C_1' must be stronger than that of C_1. We also remark that for STS components the validity of formula (37) is a sufficient but not necessary condition for refinement. We return to this point towards the end of this section.

Example 22. Recall the QLTL component $C = \mathtt{qltl}((), t, \mathtt{oven})$, introduced in Example 10:

$$\begin{aligned} \mathtt{oven} &= (t = 20 \land ((t < \bigcirc t \land t < 180)\ \mathbf{U}\ \mathtt{thermostat})) \\ \mathtt{thermostat} &= \mathbf{G}\,(180 \leq t \land t \leq 220) \end{aligned}$$

Let us introduce a refined version C' of C:

$$\begin{aligned} C' &= \mathtt{sts}((),\ t,\ (s, sw),\ \mathtt{init},\ \mathtt{trs}) \text{ where} \\ \mathtt{init} &= s = 20 \land sw = \mathsf{on} \\ \mathsf{trs} &= (t = s) \land \\ &\quad (\mathtt{if}\ sw = \mathsf{on}\ \mathtt{then}\ s < s' < s + 5\ \mathtt{else}\ (\mathtt{if}\ s > 10\ \mathtt{then}\ s - 5 < s' < s\ \mathtt{else}\ s' = s)) \land \\ &\quad (\mathtt{if}\ sw = \mathsf{on} \land s > 210\ \mathtt{then}\ sw' = \mathsf{off}\ \mathtt{else} \\ &\qquad (\mathtt{if}\ sw = \mathsf{off} \land s < 190\ \mathtt{then}\ sw' = \mathsf{on}\ \mathtt{else}\ sw' = sw)) \end{aligned}$$

C' is an STS component with no input variables, output variable t, and state variables s and sw, recording the current temperature of the oven, and the on/off status of the switch, respectively. When sw is on, the temperature increases nondeterministically by up to 5 units, otherwise the temperature decreases nondeterministically by up to 5 units. When the temperature exceeds 210, the switch is turned off; when the temperature is below 190, the switch is turned on; otherwise sw remains unchanged. The output t is always equal to the current state s. Initially the temperature is 20, and sw is on.

Using Theorem 14, and the properties of `sts2qltl` we have:

$$\begin{array}{ll}
 & C \preceq C' \\
\iff & \\
 & C \preceq \mathtt{sts2qltl}(C') \\
\iff & \\
 & \mathtt{qltl}((), t, \mathtt{oven}) \preceq \mathtt{qltl}((), t, (\forall s, sw, t : \mathtt{init} \Rightarrow (\varphi \, \mathbf{L} \, \varphi')) \wedge (\exists s, sw : \mathtt{init} \wedge \mathbf{G}\, \varphi)) \\
 & \quad \text{where } \varphi = \mathtt{trs}[s', sw' := \bigcirc s, \bigcirc sw] \text{ and } \varphi' = (\exists s', sw', t : \mathtt{trs}) \\
\iff & \{\text{Using Lemma 2, because } \varphi' \iff \mathtt{true}\} \\
 & \mathtt{qltl}((), t, \mathtt{oven}) \preceq \mathtt{qltl}((), t, (\exists s, sw : \mathtt{init} \wedge \mathbf{G}\, \varphi)) \\
\iff & \{\text{Using Theorem 14}\} \\
 & \big((\exists t : \mathtt{oven}) \Rightarrow (\exists t, s, sw : \mathtt{init} \wedge \mathbf{G}\, \varphi)\big) \wedge \big(((\exists t : \mathtt{oven}) \wedge (\exists s, sw : \mathtt{init} \wedge \mathbf{G}\, \varphi)) \Rightarrow \mathtt{oven}\big) \text{ is valid} \\
\iff & \{\text{Because } (\exists t : \mathtt{oven}) \iff \mathtt{true} \text{ and } (\exists t, s, sw : \mathtt{init} \wedge \mathbf{G}\, \varphi) \iff \mathtt{true}\} \\
 & \big((\exists s, sw : \mathtt{init} \wedge \mathbf{G}\, \varphi) \Rightarrow \mathtt{oven}\big) \text{ is valid}
\end{array}$$

Thus, checking whether C' refines C amounts to checking whether the QLTL formula $\big((\exists s, sw : \mathtt{init} \wedge \mathbf{G}\, \varphi) \Rightarrow \mathtt{oven}\big)$ is valid. This indeed holds for this example and can be shown using logical reasoning.

The above example is relatively simple in the sense that in the end refinement reduces to checking implication between the corresponding contracts. Indeed, this is always the case for input-receptive systems, as in the example above. However, refinement is *not* equivalent to implication in the general case of non-input-receptive systems. For example:

Example 23. Consider the components:

$$\begin{array}{rcl}
C_7 & = & \mathtt{stateless}(x, y, x \geq 0 \wedge y \geq x) \\
C_8 & = & \mathtt{stateless}(x, y, x \leq y \leq x + 10)
\end{array}$$

Using Theorem 14, we have:

$$\begin{array}{ll}
 & C_7 \preceq C_8 \\
\iff & \{\text{Theorem 14}\} \\
 & ((\exists y : x \geq 0 \wedge y \geq x) \Rightarrow (\exists y : x \leq y \leq x + 10)) \wedge \\
 & \quad ((\exists y : x \geq 0 \wedge y \geq x) \wedge x \leq y \leq x + 10 \Rightarrow x \geq 0 \wedge y \geq x) \text{ is valid} \\
\iff & \{\text{Arithmetic and logical reasoning}\} \\
 & \mathtt{true}
\end{array}$$

Note that the second and third parts of Theorem 14 provide necessary *and* sufficient conditions, while the first part only provides a sufficient, but generally not necessary condition. Indeed, the condition is generally not necessary in the case of STS components with state, as state space computation is ignored by the condition. This can be remedied by transforming STS components into equivalent QLTL components and then applying the third part of the theorem. An alternative which may be more tractable, particularly in the case of finite-state systems, is to use techniques akin to strategy synthesis in games, such as those proposed in [86] for finite-state relational interfaces.

Another limitation of the first part of Theorem 14 is that it requires the two STS components to have the same state space, i.e., the same state variable s. This restriction can be lifted using the well-known idea of data refinement [48, 10].

Theorem 15. *Let $C_1 = \mathtt{sts}(x, y, s, init, r)$, $C'_1 = \mathtt{sts}(x, y, t, init', r')$ be two STS components, and D a (data refinement) expression on variables s and t. Let $p = (\exists s', y : r)$ and $p' = (\exists t', y : r')$. If the formulas*

$$(\forall t : init' \Rightarrow (\exists s : D \wedge init)) \tag{40}$$

$$(\forall t, x, s : D \wedge p \Rightarrow p') \tag{41}$$

$$(\forall t, x, s, t', y : D \wedge p \wedge r' \Rightarrow (\exists s' : D[t, s := t', s'] \wedge r)) \tag{42}$$

are valid, then C_1 is refined by C'_1.

Theorem 15 is a generalization of the first part of Theorem 14 to two STS components C_1 and C'_1 with distinct state variables s and t, respectively. Specifically, the first part of Theorem 14 can be seen as a special case of Theorem 15 where D is the relation $s = t$. In general, D may be a different relation linking the state variables of C_1 to those of C'_1. For example, C_1 maybe a more abstract version of C'_1, so that in C_1 state variable s represents a set, whereas in C'_1 that set is implemented as a list represented by state variable t. In that case, D may be the relation representing what it means for t to be the correct implementation of s.

6 Toolset and Case Studies

The RCRS framework comes with a toolset, illustrated in Fig. 11. The toolset is publicly available under the MIT license and can be downloaded from `http://rcrs.gitlab.io/`. The toolset is described in detail in papers [35, 36, 37]. In summary, the toolset consists of:

- A full implementation of the RCRS theory in Isabelle [65]. The implementation consists of 22 theory files and a total of 27588 lines of Isabelle code. A detailed description of the implementation can be found in the file `document.pdf` available in the public distribution of RCRS.
- A formal *Analyzer*, which is a set of procedures implemented on top of Isabelle and the functional programming language SML. The Analyzer performs compatibility checking, automatic contract simplification, and other functions.
- A formalization of Simulink characterizing basic Simulink blocks as RCRS components and implementing those as a library of RCRS/Isabelle. At the time of writing this paper, 48 of Simulink's blocks can be handled.
- A *Translator*: a Python program translating Simulink hierarchical block diagrams into RCRS code.

We implemented in Isabelle a *shallow embedding* [17] of the language introduced in §3. The advantage of a shallow embedding is that all datatypes of Isabelle are available for specification of components, and we can use the existing Isabelle mechanism for renaming bound variables in compositions. The disadvantage of this shallow embedding is that we cannot express Algorithm 10 within Isabelle (hence the "manual" proof that we provide for Theorem 11). A *deep embedding*, in which the syntax of components is defined as a datatype of Isabelle, is possible, and is left as an open future work direction.

We implemented Algorithm 10 in SML, the meta-language of Isabelle. The SML program takes as input a component C and returns not only a simplified atomic component $\textbf{atomic}(C)$, but also a proved Isabelle theorem of the fact $C \equiv \textbf{atomic}(C)$. The simplification program, as well as a number of other procedures to perform compatibility checking, validity checking, etc., form what we call the *Analyzer* in Fig. 11.

The *Translator* takes as input a Simulink model and produces an RCRS/Isabelle theory file containing: (1) the definition of all atomic and composite components representing the Simulink diagram; and (2) embedded bottom-up simplification procedures and the corresponding correctness theorems. By running this theory file in Isabelle, we obtain an atomic component corresponding to the top-level Simulink model, equivalent to

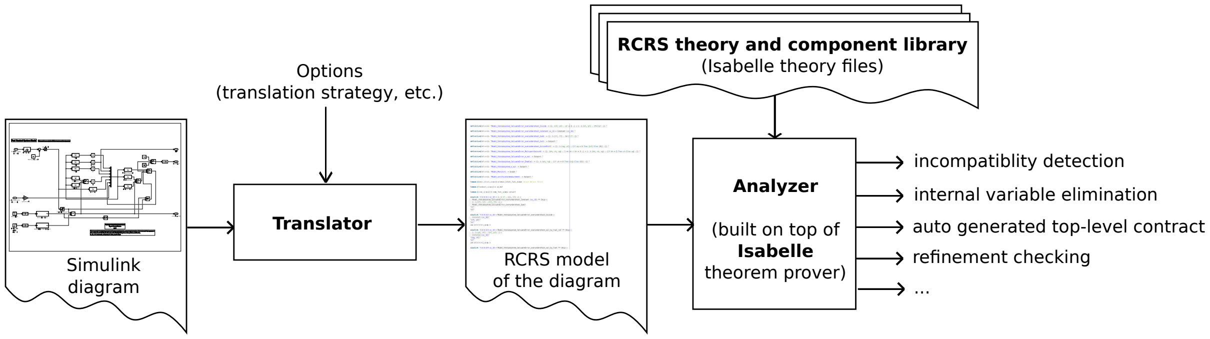

Options
(translation strategy, etc.)
RCRS theory and component library
(Isabelle theory files)
Simulink diagram
Translator
RCRS model of the diagram
Analyzer
(built on top of Isabelle theorem prover)
incompatiblity detection
internal variable elimination
auto generated top-level contract
refinement checking
...

Figure 11: The RCRS toolset.

the original composite component. As a special case, if the Simulink diagram contains inconsistencies (e.g., division by zero), these are detected by obtaining Fail as the top-level atomic component. The error can be localized by finding earlier points in the Simulink hierarchy (subsystems) which already resulted in Fail.

As mentioned earlier, how to obtain a composite component from a graphical block diagram is an interesting problem. This problem is studied in depth in [34], where several translation strategies are proposed. These various strategies all yield semantically equivalent components, but with different trade-offs in terms of size, readability, effectiveness of the simplification procedures, and so on. The Translator implements all these translation strategies, allowing the user to explore these trade-offs. Further details on the translation problem are provided in [34, 69]. A proof that the translation strategies yield semantically equivalent components is provided in [70]. This proof, which has been formalized in Isabelle, is a non-trivial result: the entire formalization of the translation algorithms and the proof is 13579 lines of Isabelle code.

We have used the RCRS toolset on several case studies, including a real-life benchmark provided by the Toyota motor company. The benchmark involves a Fuel Control System (FCS) described in [49, 50]. FCS aims at controlling the air mass and injected fuel in a car engine such that their ratio is always optimal. This problem has important implications on lowering pollution and costs by improving the engine performance. Toyota has made several versions of FCS publicly available as Simulink models at `https://cps-vo.org/group/ARCH/benchmarks`.

We have used the RCRS toolset to process two of the three Simulink models in the FCS benchmark suite (the third model contains blocks that are currently not implemented in the RCRS component library). A typical model in this set has a 3-layer hierarchy with a total of 104 Simulink block instances (97 basic blocks and 7 subsystems), and 101 connections out of which 8 are feedbacks. Each basic Simulink block is modeled in our framework by an atomic STS component (possibly stateless). These atomic STS components are created once, and form part of the RCRS implementation, which is reused for different Simulink models. The particular FCS diagram is translated into RCRS using the Translator, and simplified within Isabelle using our SML simplification procedure. After simplification, we obtain an atomic deterministic STS component with no inputs, 7 outputs, and 14 state variables. Its contract (which is 8337 characters long) includes a condition on the state variables, in particular, that a certain state variable must always be non-negative (as its value is fed into a square-root block). This condition makes it not immediately obvious that the whole system is valid (i.e., not Fail). However, we can show after applying the transformation `sts2qltl` that the resulting formula is satisfiable, which implies that the original model is consistent (i.e., no connections result in incompatibilities, there are no divisions by zero, etc.). This illustrates the use of RCRS as a powerful static analysis tool. More details on the FCS case study are provided in [34, 69, 35].

An additional case study is provided in [71], where the RCRS theory and toolset are applied for modeling systems in languages for Programmable Logic Controllers. As an example [71] models a system written in ladder logic for turning on and off lights according to a certain pattern.

7 Related Work

Several formal compositional frameworks exist in the literature. Most closely related to RCRS are the frameworks of FOCUS [18], input-output automata [58], reactive modules [6], interface automata [25], and Dill's trace theory [32]. RCRS shares with these frameworks many key compositionality principles, such as the notion of refinement. At the same time, RCRS differs and complements these frameworks in important ways. Specifically, FOCUS, IO-automata, and reactive modules, are limited to input-receptive systems, while RCRS is explicitly designed to handle non-input-receptive specifications. The benefits of non-input-receptiveness are discussed extensively in [86] and will not be repeated here (see also [88]). Interface automata are a low-level formalism whereas RCRS specifications and reasoning are symbolic. For instance, in RCRS one can naturally express systems with infinite state-spaces, input-spaces, or output-spaces. (Such systems can even be handled automatically, provided the corresponding logic they are expressed in is decidable.) Both interface automata and Dill's trace theory use a single form of asynchronous parallel composition, whereas RCRS has three primitive composition operators (serial, parallel, feedback) with synchronous semantics.

Our work adapts and extends to the reactive system setting many of the ideas developed previously in a long line of research on correctness and compositionality for sequential programs. This line of research goes back to the works of Floyd, Hoare, Dijkstra, and Wirth, on formal program semantics, weakest preconditions, program development by stepwise refinement, and so on [38, 47, 30, 89]. It also goes back to game-theoretic semantics of sequential programs as developed in the original refinement calculus [12], as well as to contract-based design [61]. Many of the concepts used in our work are in spirit similar to those used in the above works. For instance, an input-output formula ϕ used in an atomic component in our language can be seen as a *contract* between the environment of the component and the component itself: the environment must satisfy the contract by providing to the component legal inputs, and the component must in turn provide legal outputs (for those inputs). On the other hand, several of the concepts used here come from the world of reactive systems and as such do not have a direct correspondence in the world of sequential programs. For instance, this is the case with feedback composition.

RCRS extends refinement calculus from predicate to property transformers. Extensions of refinement calculus to infinite behaviors have also been proposed in the frameworks of action systems [13], fair action systems [14], and Event B [3]. These frameworks use predicate (not property) transformers as semantic foundation; they can handle certain property patterns (e.g., fairness) by providing proof rules for these properties, but they do not treat liveness and LTL properties in general [14, 91, 46]. The Temporal Logic of Actions [53] can be used to specify liveness properties, but does not distinguish between inputs and outputs, and as such cannot express non-input-receptive components. A thorough comparison of the relational interfaces precursor of RCRS and contract frameworks for system design [15] is made in [66].

Our specifications can be seen as "rich", behavioral types [54, 25]. Indeed, our work is closely related to programming languages and type theory, specifically, refinement types [39], behavioral types [64, 55, 29], and liquid types [75].

Behavioral type frameworks have also been proposed in reactive system settings. In the SimCheck framework [76], Simulink blocks are annotated with constraints on input and output variables, much like stateless components in RCRS. RCRS is more general as it also allows one to specify stateful components. RCRS is also a more complete compositional framework, with composition operators and refinement, which are not considered in [76]. Other behavioral type theories for reactive systems have been proposed in [26, 21, 33]. Compared to RCRS, these works are less general. In particular, [26, 33] are limited to specifications which separate the assumptions on the inputs from the guarantees on the outputs, and as such cannot capture input-output relations. [21] considers a synchronous model which allows to specify legal values of inputs and outputs at the *next* step, given the current state. This model does not allow to capture relations between inputs and outputs within the same step, which RCRS allows.

Our work is related to formal verification frameworks for hybrid systems [5]. Broadly speaking, these can be classified into frameworks following a model-checking approach, which typically use automata-based specification languages and state-space exploration techniques, and those following a theorem-proving approach, which typically use logic-based specifications. More closely related to RCRS are the latter, among which, CircusTime [20], KeYmaera [41], and the PVS-based approach in [2]. CircusTime can handle a larger class of Simulink diagrams than the current implementation of RCRS. In particular, CircusTime can handle *multi-rate* diagrams, where different parts of the model work at different rates (periods). On the other hand, CircusTime is based on predicate (not property) transformers, and as such cannot handle liveness properties. KeYmaera is a theorem prover based on differential dynamic logic [67], which is itself based on dynamic logic [43]. The focus of both KeYmaera and the work in [2] is verification, and not compositionality. For instance, these works do not distinguish between inputs and outputs and do not investigate considerations such as input-receptiveness. ClawZ is a translator of Simulink diagrams into Z [9]. The work of [74] distinguishes inputs and outputs, but provides a system model where the output relation is separated from the transition relation, and where the output relation is assumed to be total, meaning that there exists an output for every input and current state combination. This does not allow to specify non-input-receptive stateless components, such as for example the `Div` component from §3.

Our component algebra is similar to the algebra of flownomials [83] and to the relational model for non-deterministic dataflow [45]. In [23], graphs and graph operations which can be viewed as block diagrams are represented by algebraic expressions and operations, and a complete equational axiomatization of the equivalence of the graph expressions is given. This is then applied to flow-charts as investigated in [78]. The translation of block diagrams in general and Simulink in particular has been treated in a large number of papers, with various goals, including verification and code generation (e.g., see [87, 60, 22, 56, 79, 16, 92, 93, 94, 62]). Although we share several of the ideas of the above works, our main goal here is not to formalize the language of block diagrams, neither their translation to other formalisms, but to provide a complete compositional framework for reasoning about reactive systems.

RCRS is naturally related to *compositional verification* frameworks, such as [42, 1, 59, 80, 44, 27, 28]. In particular, compositional verification frameworks often make use of a refinement relation such as trace inclusion or simulation [80, 44]. However, the focus of these frameworks is different than that of RCRS. In compositional verification, the focus is to "break down" a large (and usually computationally expensive) verification task into smaller (and hopefully easier to calculate) subtasks. For this purpose, compositional verification frameworks employ several kinds of *decomposition rules*. An example of such a rule is the so-called *precongruence* rule (i.e., preservation of refinement by composition): if P_1 refines Q_1, and P_2 refines Q_2, then the composition $P_1 \| P_2$ refines $Q_1 \| Q_2$. This, together with preservation of properties by refinement, allows us to conclude that $P_1 \| P_2$ satisfies some property ϕ, provided we can prove that $Q_1 \| Q_2$ satisfies ϕ. The latter might be a simpler verification task, if Q_1 and Q_2 are smaller than P_1 and P_2. The essence of compositional verification is in finding such *abstract* versions Q_1 and Q_2 of the concrete processes in question, P_1 and P_2, and employing decomposition rules like the one above in the hope of making verification simpler. RCRS can also be used for compositional verification: indeed, RCRS provides both the precongruence rule, and preservation of properties by refinement. Note that, in traditional settings, precongruence is not always powerful enough, and for this reason most compositional verification frameworks employ more complex decomposition rules (e.g., see [63]). In settings which allow non-input-receptive components, such as ours, there are indications that the precongruence rule is sufficient for compositional verification purposes [81], although more work is required to establish this in the specific context of RCRS. Such work is beyond the scope of the current paper. We also note that, beyond compositional verification with precongruence, RCRS provides a behavioral type theory which allows to state system properties such as compatibility, which is typically not available in compositional verification frameworks.

Refinement can be seen as the inverse of *abstraction*, and as such our framework is related to general frameworks such as *abstract interpretation* [24]. Several abstractions have been proposed in reactive system settings, including *relational* abstractions for hybrid systems, which are related to Simulink [77]. The focus of these works is verification, and abstraction is used as a mechanism to remove details from the model that make verification harder. In RCRS, the simplification procedure that we employ can be seen as an abstraction process, as it eliminates internal variable information. However, RCRS simplification is an *exact* abstraction, in the sense that it does not lose any information: the final system is equivalent to the original one, and not an over- or under-approximation, as is usually the case with typical abstractions for verification purposes.

8 Conclusion

We presented the Refinement Calculus of Reactive Systems (RCRS), a compositional framework for modeling and reasoning about reactive systems. In contrast to other frameworks, in RCRS we are able to model input-output systems which are both non-deterministic and non-input-receptive, which allows for local compatibility checks similar to type checking in programming languages. RCRS contains a rich language which allows one to describe both atomic and composite systems with synchronous behavior. The semantics of RCRS is based on the theory of monotonic property transformers, an extension of the theory of monotonic predicate transformers from classic refinement calculus. Among other methods for symbolic reasoning, we presented techniques for symbolic composition (reducing a composite system to an atomic system), checking compatibility, and checking refinement. We also briefly presented the RCRS toolset which includes a full implementation of RCRS in the Isabelle theorem prover (more than 27k lines of Isabelle code) and a Simulink-to-RCRS translator. This paper focuses on the theory and methodology of RCRS, its formal semantics, and techniques for symbolic and computer-aided reasoning. For more information about the toolset we refer the reader to the relevant papers [34, 69, 35, 36, 70, 37], as well as the toolset's web site `http://rcrs.gitlab.io/` which contains up-to-date information.

RCRS is an ongoing project, and a number of problems remain open. Future work directions include:

- An extension of the framework to systems with algebraic loops, which necessitates handling instantaneous feedback. Here, the preliminary ideas of [73] can be helpful in defining the semantics of instantaneous feedback. However, [73] does not provide solutions on how to obtain symbolic closed-form expression for the feedback of general components.
- Extension of the results of §5.5 to general components, possibly non-deterministic or non-decomposable.
- An extension of the framework to *acausal* systems, i.e., systems without a clear distinction of inputs and outputs [40].
- An extension of the framework to *stochastic* systems.
- The development of better symbolic reasoning techniques, such as simplification of logical formulas, decision procedures, etc.
- Application of the framework to other domains, such as machine learning and AI.

Acknowledgements

This work was partially supported by the Academy of Finland and the U.S. National Science Foundation (awards CNS-1329759 and CNS-1801546). Dragomir was supported during the writing of this paper by the Horizon 2020 Programme Strategic Research Cluster (grant agreement #730080 and #730086).

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, 1995.

[2] E. Ábrahám-Mumm, M. Steffen, and U. Hannemann. Verification of hybrid systems: Formalization and proof rules in PVS. In *7th Intl. Conf. Engineering of Complex Computer Systems (ICECCS 2001)*, pages 48–57, 2001.

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

[4] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181 – 185, 1985.

[5] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

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

[7] R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *CONCUR'98*, volume 1466 of *LNCS*. Springer, 1998.

[8] A. Armstrong, V. Gomes, and G. Struth. Building program construction and verification tools from algebraic principles. *Formal Aspects of Computing*, 28(2), 2016.

[9] R. D. Arthan, P. Caseley, C. O'Halloran, and A. Smith. ClawZ: Control Laws in Z. In *3rd IEEE International Conference on Formal Engineering Methods, ICFEM 2000, York, England, UK, September 4-7, 2000, Proceedings*, pages 169–176, 2000.

[10] R. J. Back. *Correctness preserving program refinements: proof theory and applications*, volume 131 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1980.

[11] R.-J. Back and M. Butler. *Exploring summation and product operators in the refinement calculus*, pages 128–158. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[12] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[13] 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.

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

[15] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. A. Henzinger, and K. G. Larsen. Contracts for system design. *Foundations and Trends in Electronic Design Automation*, 12(2-3):124–400, 2018.

[16] P. Boström. Contract-Based Verification of Simulink Models. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 291–306. Springer Berlin Heidelberg, 2011.

[17] R. J. Boulton, A. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, and J. V. Tassel. Experience with embedding hardware description languages in HOL. In *IFIP TC10/WG 10.2 Intl. Conf. on Theorem Provers in Circuit Design*, pages 129–156. North-Holland Publishing Co., 1992.

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

[19] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuxmv symbolic model checker. In A. Biere and R. Bloem, editors, *Computer Aided Verification: 26th International Conference, CAV 2014,*, pages 334–342, Cham, 2014. Springer.

[20] A. L. C. Cavalcanti, A. Mota, and J. C. P. Woodcock. Simulink timed models for program verification. In Z. Liu, J. C. P. Woodcock, and H. Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 82–99. Springer, 2013.

[21] A. Chakrabarti, L. de Alfaro, T. Henzinger, and F. Mang. Synchronous and bidirectional component interfaces. In *CAV*, LNCS 2404, pages 414–427. Springer, 2002.

[22] C. Chen, J. S. Dong, and J. Sun. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing*, 21(5):451–483, 2009.

[23] B. Courcelle. A representation of graphs by algebraic expressions and its use for graph rewriting systems. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph-Grammars and Their Application to Computer Science, 3rd International Workshop, Warrenton, Virginia, USA, December 2-6, 1986*, volume 291 of *Lecture Notes in Computer Science*, pages 112–132. Springer, 1986.

[24] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. POPL*, 1977.

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

[26] L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *EMSOFT'01*. Springer, LNCS 2211, 2001.

[27] W. de Roever, H. Langmaack, and A. E. Pnueli. *Compositionality: The Significant Difference*. LNCS. Springer, 1998.

[28] W.-P. de Roever, F. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Non-compositional Methods*. Cambridge University Press, 2012.

[29] K. Dhara and G. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE'96: 18th Intl. Conf. on Software Engineering*, pages 258–267. IEEE Computer Society, 1996.

[30] E. Dijkstra. Notes on structured programming. In O. Dahl, E. Dijkstra, and C. Hoare, editors, *Structured programming*, pages 1–82. Academic Press, London, UK, 1972.

[31] E. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*, 18(8):453–457, 1975.

[32] D. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. MIT Press, Cambridge, MA, USA, 1987.

[33] L. Doyen, T. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *8th ACM & IEEE International conference on Embedded software, EMSOFT*, pages 79–88, 2008.

[34] I. Dragomir, V. Preoteasa, and S. Tripakis. Compositional Semantics and Analysis of Hierarchical Block Diagrams. In *SPIN*, pages 38–56. Springer, 2016.

[35] I. Dragomir, V. Preoteasa, and S. Tripakis. The Refinement Calculus of Reactive Systems Toolset. *CoRR*, abs/1710.08195, 2017.

[36] I. Dragomir, V. Preoteasa, and S. Tripakis. The Refinement Calculus of Reactive Systems Toolset. In *TACAS*, 2018.

[37] I. Dragomir, V. Preoteasa, and S. Tripakis. The Refinement Calculus of Reactive Systems Toolset. *International Journal on Software Tools for Technology Transfer*, pages 1–20, 2020.

[38] R. Floyd. Assigning meanings to programs. In *In. Proc. Symp. on Appl. Math. 19*, pages 19–32. American Mathematical Society, 1967.

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

[40] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley, 2 edition, 2014.

[41] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In A. P. Felty and A. Middeldorp, editors, *CADE*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015.

[42] O. Grumberg and D. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

[43] D. Harel, J. Tiuryn, and D. Kozen. *Dynamic Logic.* MIT Press, 2000.

[44] T. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In *CAV'98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.

[45] T. T. Hildebrandt, P. Panangaden, and G. Winskel. A relational model of non-deterministic dataflow. *Mathematical Structures in Computer Science*, 14(5):613–649, 10 2004.

[46] T. S. Hoang and J.-R. Abrial. Reasoning About Liveness Properties in Event-B. In *Proceedings of the 13th International Conference on Formal Methods and Software Engineering*, ICFEM'11, pages 456–471, Berlin, Heidelberg, 2011. Springer-Verlag.

[47] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[48] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4), Dec. 1972.

[49] X. Jin, J. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Benchmarks for model transformations and conformance checking. In *1st Intl. Workshop on Applied Verification for Continuous and Hybrid Systems (ARCH)*, 2014.

[50] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Powertrain Control Verification Benchmark. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC'14, pages 253–262. ACM, 2014.

[51] Y. Kesten, Z. Manna, and A. Pnueli. Temporal verification of simulation and refinement. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency Reflections and Perspectives: REX School/Symposium*, pages 273–346. Springer, 1994.

[52] Y. Kesten and A. Pnueli. Complete Proof System for QPTL. *Journal of Logic and Computation*, 12(5):701, 2002.

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

[54] E. Lee and Y. Xiong. System-level types for component-based design. In *EMSOFT'01: 1st Intl. Workshop on Embedded Software*, pages 237–253. Springer, 2001.

[55] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[56] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams – modularity vs. code size. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09)*, pages 78–89. ACM, Jan. 2009.

[57] R. Lublinerman and S. Tripakis. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Design, Automation, and Test in Europe (DATE'08)*, pages 1504–1509. ACM, Mar. 2008.

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

[59] K. McMillan. A compositional rule for hardware design refinement. In *Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*. Springer-Verlag, 1997.

[60] B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for Translating Simulink Models into Input Language of a Model Checker. In *Formal Methods and Software Engineering*, volume 4260 of *LNCS*, pages 606–620. Springer, 2006.

[61] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[62] S. Minopoli and G. Frehse. SL2SX Translator: From Simulink to SpaceEx Verification Tool. In *19th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2016.

[63] K. S. Namjoshi and R. J. Trefler. On the completeness of compositional reasoning methods. *ACM Trans. Comput. Logic*, 11(3), 2010.

[64] O. Nierstrasz. Regular types for active objects. *SIGPLAN Not.*, 28(10):1–15, 1993.

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

[66] P. Nuzzo, A. Iannopollo, S. Tripakis, and A. L. Sangiovanni-Vincentelli. Are Interface Theories Equivalent to Contract Theories? In *12th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2014.

[67] A. Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.*, 41(2):143–189, 2008.

[68] A. Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[69] V. Preoteasa, I. Dragomir, and S. Tripakis. Type Inference of Simulink Hierarchical Block Diagrams in Isabelle. In *37th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, 2017.

[70] V. Preoteasa, I. Dragomir, and S. Tripakis. Mechanically Proving Determinacy of Hierarchical Block Diagram Translations. In *VMCAI 2019 - 20th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2019.

[71] V. Preoteasa, T. Latvala, and K. Varpaaniemi. Modelling programmable logic controllers in refinement calculus of reactive systems. In K. Ropiak, L. Polkowski, and P. Artiemjew, editors, *Proceedings of the 28th International Workshop on Concurrency, Specification and Programming, Olsztyn, Poland, September 24-26th, 2019*, volume 2571 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[72] V. Preoteasa and S. Tripakis. Refinement calculus of reactive systems. In *Embedded Software (EMSOFT), 2014 International Conference on*, pages 1–10, Oct 2014.

[73] V. Preoteasa and S. Tripakis. Towards Compositional Feedback in Non-Deterministic and Non-Input-Receptive Systems. In *31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2016.

[74] G. Reissig, A. Weber, and M. Rungger. Feedback refinement relations for the synthesis of symbolic controllers. *IEEE Trans. Automat. Contr.*, 62(4):1781–1796, 2017.

[75] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

[76] P. Roy and N. Shankar. SimCheck: An expressive type system for Simulink. In C. Muñoz, editor, *2nd NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 149–160, Langley Research Center, Hampton VA 23681-2199, USA, Apr. 2010. NASA.

[77] S. Sankaranarayanan and A. Tiwari. Relational abstractions for continuous and hybrid systems. In *Computer Aided Verification: 23rd International Conference, CAV 2011*, pages 686–702. Springer, 2011.

[78] H. Schmeck. Algebraic characterization of reducible flowcharts. *Journal of Computer and System Sciences*, 27(2):165 – 199, 1983.

[79] V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, and J. Sifakis. Compositional translation of Simulink models into synchronous BIP. In *Industrial Embedded Systems (SIES), 2010 International Symposium on*, pages 217–220, July 2010.

[80] N. Shankar. Lazy compositional verification. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 541–564, London, UK, 1998. Springer-Verlag.

[81] A. Siirtola, S. Tripakis, and K. Heljanko. When do we not need complex assume-guarantee rules? *ACM Trans. Embed. Comput. Syst.*, 16(2):48:1–48:25, Jan. 2017.

[82] A. P. Sistla, M. Y. Vardi, and P. Wolper. The Complementation Problem for Büchi Automata with Applications to Temporal Logic. *Theoretical Computer Science*, 49:217–237, 1987.

[83] G. Stefănescu. *Network Algebra*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[84] S. Tripakis. Compositionality in the Science of System Design. *Proceedings of the IEEE*, 104(5):960–972, May 2016.

[85] S. Tripakis, B. Lickly, T. Henzinger, and E. Lee. On Relational Interfaces. In *Proceedings of the 9th ACM & IEEE International Conference on Embedded Software (EMSOFT'09)*, pages 67–76. ACM, 2009.

[86] 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.

[87] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating Discrete-time Simulink to Lustre. *ACM Trans. Embed. Comput. Syst.*, 4(4):779–818, Nov. 2005.

[88] S. Tripakis, C. Stergiou, M. groy, and E. A. L. e. Error-Completion in Interface Theories. In *International SPIN Symposium on Model Checking of Software – SPIN 2013*, volume 7976 of *LNCS*, pages 358–375. Springer, 2013.

[89] N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 14(4):221–227, 1971.

[90] J. Woodcock and A. Cavalcanti. A tutorial introduction to designs in unifying theories of programming. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods*, pages 40–66. Springer, 2004.

[91] 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.

[92] C. Yang and V. Vyatkin. Transformation of Simulink models to IEC 61499 Function Blocks for verification of distributed control systems. *Control Engineering Practice*, 20(12):1259–1269, 2012.

[93] C. Zhou and R. Kumar. Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata. *Discrete Event Dynamic Systems*, 22(2):223–247, 2012.

[94] L. Zou, N. Zhany, S. Wang, M. Franzle, and S. Qin. Verifying Simulink diagrams via a Hybrid Hoare Logic Prover. In *Embedded Software (EMSOFT)*, Sept 2013.