Information Flow Guided Synthesis

Bernd Finkbeiner[1], Niklas Metzger[1(✉)], and Yoram Moses[2]

[1] CISPA Helmholtz Center of Information Security, Saarbrücken, Germany
{finkbeiner,niklas.metzger}@cispa.de

[2] The Andrew and Erna Viterbi Faculty of Electrical and Computer Engineering and the Taub Faculty of Computer Science, Technion, Haifa, Israel
moses@technion.ac.il

Abstract. Compositional synthesis relies on the discovery of assumptions, i.e., restrictions on the behavior of the remainder of the system that allow a component to realize its specification. In order to avoid losing valid solutions, these assumptions should be *necessary* conditions for realizability. However, because there are typically many different behaviors that realize the same specification, necessary behavioral restrictions often do not exist. In this paper, we introduce a new class of assumptions for compositional synthesis, which we call *information flow assumptions*. Such assumptions capture an essential aspect of distributed computing, because components often need to act upon information that is available only in other components. The presence of a certain flow of information is therefore often a necessary requirement, while the actual behavior that establishes the information flow is unconstrained. In contrast to behavioral assumptions, which are properties of individual computation traces, information flow assumptions are *hyperproperties*, i.e., properties of sets of traces. We present a method for the automatic derivation of information-flow assumptions from a temporal logic specification of the system. We then provide a technique for the automatic synthesis of component implementations based on information flow assumptions. This provides a new compositional approach to the synthesis of distributed systems. We report on encouraging first experiments with the approach, carried out with the BoSyHyper synthesis tool.

1 Introduction

In *distributed synthesis*, we are interested in the automatic translation of a formal specification of a distributed system's desired behavior into an implementation that satisfies the specification [22]. What makes distributed synthesis far more interesting than the standard synthesis of reactive systems, but also more challenging, is that the result consists of a set of implementations of subsystems, each of which operates based only on partial knowledge of the global system state. While algorithms for distributed synthesis have been studied since the

This work was funded by the German Israeli Foundation (GIF) Grant No. I-1513-407./2019. and by DFG grant 389792660 as part of TRR 248 – CPEC.

S. Shoham and Y. Vizel (Eds.): CAV 2022, LNCS 13372, pp. 505–525, 2022.
https://doi.org/10.1007/978-3-031-13188-2_25

1990s [10,18,22], their high complexity has resulted in applications of distributed synthesis being, so far, very limited.

One of the most promising approaches to making distributed synthesis more scalable is *compositional synthesis* [7,9,14,19,23]. The compositional synthesis of a distributed system with two processes, p and q, avoids the construction of the product of p and q and instead focuses on one process at a time. Typically, it is impossible to realize one process without making certain assumptions about the other process. Compositional synthesis therefore critically depends on finding the assumption that p must make about q, and vice versa: once the assumptions are known, one can build each individual process, relying on the fact that the assumption will be satisfied by the synthesized implementation of the other process. Ideally, the assumptions should be both *sufficient* (i.e., the processes are realizable under the assumptions) and *necessary* (i.e., any implementation that satisfies the specification would also satisfy the assumptions). Without sufficiency, the synthesis cannot find a compositional solution; without necessity, the synthesis loses valid solutions. While sufficiency is obviously checked as part of the synthesis process, it is often impossible to find necessary conditions, because the specifications can be realized by many different behaviors. Any concrete implementation would lead to a specific assumption; however, this implementation is only known once the synthesis is complete, and an assumption that is satisfied by *all* implementations often does not exist.

In this paper, we propose a way out of this chicken-and-egg type of situation. Previous work on generating assumptions for compositional synthesis has focused on *behavioral* restrictions on the environment of a subsystem. We introduce a new class of more abstract assumptions that, instead, focus on the *flow of information.* Consider a system architecture (depicted in Fig. 1a) where two processes a and b are linked by a communication channel `c`, such that a can write to `c` and b can read from `c`. Suppose also that a reads a boolean input `in` from the environment that is, however, not directly visible to b. We are interested in a distributed implementation for a specification that demands that b should eventually output the value of input `in`. Since b cannot observe `in`, its synthesis must rely on the assumption that the value of `in` will be communicated over the channel `c` by process a. Expressing this as a *behavioral assumption* is difficult, because there are many different behaviors that accomplish this. Process a could, for example, literally copy the value of `in` to `c`. It could also encode the value, for example by writing to `c` the negation of the value of `in`. Alternatively, it could delay the transmission of `in` by an arbitrary number of steps, and even use the length of the delay to encode information about the value of `in`. Fixing any such communication protocol, by a corresponding behavioral assumption on a, would unnecessarily eliminate potential implementations of b. The minimal assumption that subsystem a must satisfy is in fact an information-flow assumption, namely that b will eventually be able to determine the value of `in`.

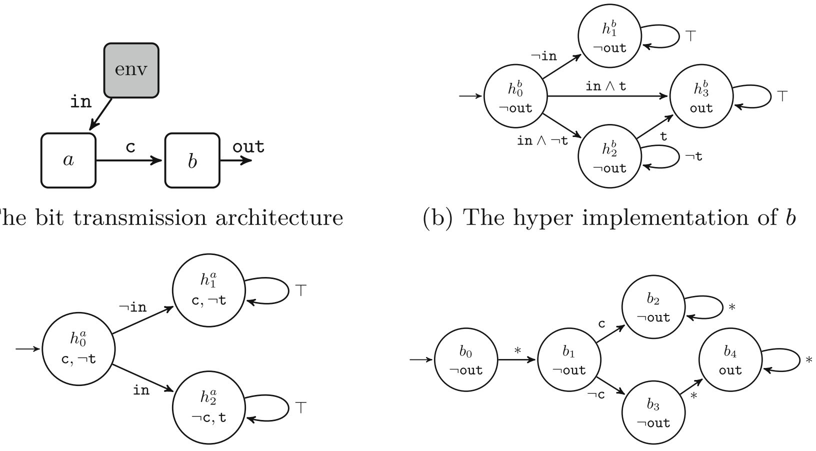
We present a method that derives necessary information flow assumptions automatically. A fundamental difference between behavioral and information flow assumptions is that behavioral assumptions are *trace properties*, i.e., properties of individual traces; by contrast, information flow assumptions are *hyperproperties*, i.e., properties of *sets* of traces. In our example, the assumption that a will eventually communicate the value of `in` to b is the hyperproperty that any two traces that differ in the value of `in` must eventually also differ in `c`. The precise difference between the two traces depends on the communication protocol chosen in the implementation of a; however, any correct implementation of a must ensure that some difference in b's input (on channel `c`) in the two traces occurs, so that b can then respond with a different output.

Once we have obtained information flow assumptions for all of the subsystems, we proceed to synthesize each subsystem under the assumption generated for its environment. It is important to note that, at this point, the implementation of the environment is not known yet; as a result, we only know *what* information will be provided to process b, but not *how*. This also means that we cannot yet construct an executable implementation of the process under consideration; after all, this implementation would need to correctly decode the information provided by its partner processes. Clearly, we cannot determine how to *decode* the information before we know how the implementation of the sending process *encodes* the information!

Our solution to this quandary is to synthesize a prototype of an implementation for the process that works with *any* implementation of the sender, as long as the sender satisfies the information flow requirement. The prototype differs from the actual implementation in that it has access to the original (unencoded) information. Because of this information the prototype, which we call a *hyper implementation*, can determine the correct output that satisfies the specification. Later, in the actual implementation, the information is no longer available in its original, unencoded form, but must instead be decoded from the communication received from the environment. However, the information flow assumption guarantees that this is actually possible, and access to the original information is, therefore, no longer necessary.

In Sect. 2, we explain our approach in more detail, continuing the discussion of the bit transmission example mentioned above. The paper then proceeds to make the following contributions:

- We introduce the notion of *necessary information flow assumptions* (Sect. 4.1) for distributed systems with two processes and present a method for the automatic derivation of such assumptions from process specifications given in linear-time temporal logic (LTL).
- We strengthen information flow assumptions to the notion of *time-bounded* information flow assumptions (Sect. 4.2), which characterizes information that must be received in finite time. We introduce the notion of *uniform distinguishability* and prove that uniform distinguishability guarantees the necessity of the information flow assumption.
- We introduce the notion of *hyper implementations* (Sect. 5) and provide a synthesis method for their automatic construction. We also explain how to transform hyper implementations into actual process implementations.



(a) The bit transmission architecture

(b) The hyper implementation of b

(c) The hyper implementation of a

(d) The implementation of b

Fig. 1. The distributed system of the *bit transmission* protocol. The architecture is given in (a), the hyper implementation of b in (b), the hyper implementation of a in(c), and the resulting local implementation of b in (d).

- We present a more restricted *practical approach* (Sect. 6) that simplifies the synthesis for cases where the information flow assumption refers to a finite amount of information.
- Finally, we report on encouraging experimental results (Sect. 7).

2 The Bit Transmission Problem

We use the *bit transmission* example from the introduction to motivate our approach. The example consists of two processes a and b that are combined into the distributed architecture shown in Fig. 1a. Process a observes the (binary) input of the environment through variable `in` and can communicate with the second process b via a channel (modeled by the shared variable `c`). Process b observes its own local input from a and has a local output `out`. We are interested in synthesizing an implementation for our distributed system consisting of two strategies, one for each process, whose combined behavior satisfies the specification. In this example, the specification for process b is to transmit the initial value of `in`, an input of a, to b's own output; this is expressed by the linear-time temporal logic (LTL) formula $\varphi_b = \texttt{in} \leftrightarrow \Diamond\, \texttt{out}$. The specification does not restrict a's behavior, and so $\varphi_a = \mathit{true}$. Since the value of `out` is controlled by b, whereas `in` is determined by the environment and observed by a, this specification forces b to react to an input that b neither observes nor controls. To satisfy the goal, `out` must remain *false* forever if `in` is initially *false*, while `out` must eventually become *true* at least once if `in` starts with value *true*. Indeed, in order to set `out` to *true*, process b must *know* that `in` is initially *true*, which can only be satisfied via information flow from a to b. We can capture this information flow requirement as the following hyperproperty: For every pair of traces that disagree on the initial value of in, process a must (eventually) behave differently on c. The requirement can be expressed in HyperLTL by the formula $\Psi = \forall \pi, \pi'.(\mathtt{in}_\pi \not\leftrightarrow \mathtt{in}_{\pi'}) \rightarrow \Diamond(\mathtt{c}_\pi \not\leftrightarrow \mathtt{c}_{\pi'})$. The information flow requirement does not restrict a to behave in a particular manner; the *encoding* of the information about in on the channel c depends on a's behavior. Under the assumption that a will behave according to the information flow requirement Ψ, one can synthesize a solution of b that is correct for every implementation of a. Given its generality, we call such a solution a *hyper implementation.* The hyper implementation of process b is shown in Fig. 1b. Since the point in time when the information is received by b is unknown during the local synthesis process, an additional auxiliary boolean variable t is added to the specification of b. This variable signals that the information has been transmitted and is later derived by a's implementation. Setting out to *true* is only allowed after t is observed by process b. When the hyper implementation is composed with the actual implementation of a, as shown in Fig. 1c, both local specifications are satisfied. The resulting local implementation of b, depicted in Fig. 1d, branches only on local inputs and, together with a, satisfies the specification. While changing state b_0 to b_1, process b cannot distinguish in from $\neg$in. It has to wait for one time step, i.e., the first difference in outputs of process a, to observe the difference in the shared communication channel. The value of t is obtained from a's implementation and set to *true* with the first difference in c, forbidding the edge from h_0^b to h_3^b in the local implementation of b.

3 Preliminaries

Architectures. For ease of exposition we focus in this paper on systems with two processes. Let $\mathcal{V}$ be a set of variables. An architecture with two black-box processes p and q is given as a tuple $(I_p, I_q, O_p, O_q, I_e)$, where I_p, I_q, O_p, O_q, and I_e are all subsets of $\mathcal{V}$. O_p and O_q are the *output variables* of p and q. O_e are the output variables of the uncontrollable environment. The three sets O_p, O_q and O_e form a partition of $\mathcal{V}$. I_p and I_q are the *input variables* of processes p and q, respectively. For each black-box process, the inputs and outputs are disjoint, i.e., $I_p \cap O_p = \emptyset$ and $I_q \cap O_q = \emptyset$. The inputs I_p and I_q of the black-box processes are all either outputs of the environment or outputs of the other black-box process, i.e., $I_p \subseteq O_q \cup O_e$ and $I_q \subseteq O_p \cup O_e$. We assume that all variables are of boolean type. For a set $V \subseteq \mathcal{V}$, every subset $V' \subseteq V$ defines a *valuation* of V, where the variables in V' have value *true* and the variables in $V \setminus V'$ have value *false*.

Implementations. An implementation of an architecture $(I_p, I_q, O_p, O_q, I_e)$ is a pair (s_p, s_q), consisting of a strategy for each of the two black-box processes. A *strategy* for a black-box process p is a function $s_p : (2^{I_p})^* \rightarrow (2^{O_p})$ that maps finite sequences of valuations of p's input variables (i.e., *histories* of inputs) to a valuation of p's output variables. The (synchronous) *composition* $s_p || s_q$ of the two strategies is the function $s : (2^{O_e})^* \rightarrow (2^{\mathcal{V}})$ that maps finite sequences of valuations of the environment's output variables to valuations of all variables: we define $s(\epsilon) = s_p(\epsilon) \cup s_q(\epsilon)$ and, for $v \in (2^{O_e})^*, x \in 2^{O_e}$, $s(v \cdot x) = (s_p(f_p(v)) \cup s_q(f_q(v)) \cup x)$, where f_p and f_q map sequences of environment outputs to sequences of process inputs with $f_p(\epsilon) = \epsilon, f_p(v \cdot x) = f_p(v) \cdot ((x \cup s_q(f_q(v))) \cap I_p)$ and $f_q(\epsilon) = \epsilon, f_q(v \cdot x) = f_p(v) \cdot ((x \cup s_p(f_p(v))) \cap I_q)$.

Specifications. Our specifications refer to traces over the set $\mathcal{V}$ of all variables. In general, for a set $V \subseteq \mathcal{V}$ of variables, a *trace* over V is an infinite sequence $x_0 x_1 x_2 \ldots \in (2^{\mathcal{V}})^\omega$ of valuations of V. A *specification* $\varphi \subseteq (2^{\mathcal{V}})^\omega$ is a set of traces over $\mathcal{V}$. Two traces of disjoint sets $V, V' \subset \mathcal{V}$ can be *combined* by forming the union of their valuations at each position, i.e., $x_0 x_1 x_2 \ldots \sqcup y_0 y_1 y_2 \ldots = (x_0 \cup y_0)(x_1 \cup y_1)(x_2 \cup y_2) \ldots$. Likewise, the *projection* of a trace onto a set of variables $V' \subseteq \mathcal{V}$ is formed by intersecting the valuations with V' at each position: $x_0 x_1 x_2 \ldots \downarrow_{V'} = (x_0 \cap V')(x_1 \cap V')(x_2 \cap V') \ldots$.

For our specification language, we use propositional linear-time temporal logic (LTL) [21], with the set $\mathcal{V}$ of variables as atomic propositions and the usual temporal operators Next $\bigcirc$, Until $\mathcal{U}$, Globally $\square$, and Eventually $\diamondsuit$. System specifications are given as a conjunction $\varphi_p \wedge \varphi_q$ of two LTL formulas, where φ_p refers only to variables in $O_p \cup O_e$, i.e., the formula relates the outputs of process p to the outputs of the environment, and φ_q refers only to variables in $O_q \cup O_e$. The two formulas represent the *local specifications* for the two black-box processes. An implementation $s = (s_p, s_q)$ defines a set of traces

$$\begin{aligned} \mathit{Traces}(s_p, s_q) = \{x_0 x_1 \ldots \in (2^O)^\omega \mid x_k = s(i_0 i_1 \ldots i_{k-1}) \text{ for all } k \in \mathbb{N} \\ \text{for some } i_o i_1 i_2 \ldots \in (2^{O_e})^\omega\}. \end{aligned}$$

We say that an implementation *satisfies* the specification if the traces of the implementation are contained in the specification, i.e., $\mathit{Traces}(s_p, s_q) \subseteq \varphi$.

The Synthesis Problem. Given an architecture and a specification φ, the synthesis problem is to find an implementation $s = (s_p, s_q)$ that satisfies φ. We say that a specification φ is *realizable* in a given architecture if such an implementation exists, and *unrealizable* if not.

Hyperproperties. We capture information-flow assumptions as hyperproperties. A *hyperproperty over* $\mathcal{V}$ is a set $H \subseteq 2^{(2^{\mathcal{V}})^\omega}$ of sets of traces over $\mathcal{V}$ [6]. An implementation (s_p, s_q) satisfies the hyperproperty H iff its traces are an element of H, i.e., $\mathit{Traces}(s_p, s_q) \in H$. A convenient specification language for hyperproperties is the temporal logic HyperLTL [5]. HyperLTL extends LTL with quantification over trace variables. The syntax of HyperLTL is given by the following grammar $\varphi := \forall \pi.\, \varphi \mid \exists \pi.\, \varphi \mid \psi$ and $\psi := v_\pi \mid \neg\psi \mid \psi \wedge \psi \mid \bigcirc \psi \mid \psi \mathcal{U} \psi$ where $v_\pi \in \mathcal{V}$ is a variable and $\pi \in \mathcal{T}$ is a trace variable. Note that the output variables are indexed by trace variables. The quantification over traces makes it possible to express properties like "ψ *must hold on all traces*", which is expressed by $\forall \pi.\, \psi$. Dually, one can express that "*there exists a trace on which* ψ *holds*", denoted by $\exists \pi.\, \psi$. The temporal operators are defined as in LTL.

In some cases, a hyperproperty can be expressed in terms of a binary relation on traces. A relation $R \subseteq (2^{\mathcal{V}})^\omega \times (2^{\mathcal{V}})^\omega$ of pairs of traces defines the hyperproperty H, where a set T of traces is an element of H iff for all pairs $\pi, \pi' \in T$ of traces in T it holds that $(\pi, \pi') \in R$. We call a hyperproperty defined in this way a *2-hyperproperty.* In HyperLTL, 2-hyperproperties are expressed as formulas with two universal quantifiers and no existential quantifiers. A 2-hyperproperty can equivalently be represented as a set of infinite sequences over the product alphabet Σ^2: for a given 2-hyperproperty $R \subseteq \Sigma^\omega \times \Sigma^\omega$, let $R' = \{(\sigma_0, \sigma_0')(\sigma_1, \sigma_1') \ldots \mid (\sigma_0\sigma_1 \ldots, \sigma_0'\sigma_1' \ldots) \in R\}$. This representation is convenient for the use of automata to recognize 2-hyperproperties.

4 Necessary Information Flow in Distributed Systems

In reactive synthesis it is natural that the synthesized process reacts to different environment outputs. This is also the case for distributed synthesis, where some outputs of the environment are not observable by a local process and the hidden values must be communicated to the process. In the following we show when such information flow is necessary.

4.1 Necessary Information Flow

Our analysis focuses on pairs of situations for which the specification dictates a *different* reaction from a given black-box process p. Such pairs imply the need for information flow that will enable p to distinguish the two situations: if p cannot distinguish the two situations, it will behave in the same manner in both. Consequently, the specification will be violated, no matter how p is implemented, in at least one of the two situations. A process p needs to satisfy a local specification φ_p, which relates its outputs O_p to the outputs O_e of the environment. (Recall that O_e may contain inputs to the other black-box process.) We are therefore interested in pairs of traces over O_e for which φ_p does *not* admit a common valuation of O_p. We collect such pairs of traces in a *distinguishability relation*, denoted by Δ_p:

Definition 1 (Distinguishability). *Given a local specification φ_p for process p, the* distinguishability relation *Δ_p is the set of pairs of traces over O_e (environment outputs) such that no trace over O_p satisfies φ_p in combination with both traces in the pair. Formally:*

$$\begin{aligned}\Delta_p = \{(\pi_e, \pi_e') \in &(2^{O_e})^\omega \times (2^{O_e})^\omega \mid \\ &\forall \pi_p \in (2^{O_p})^\omega.\ \textit{if}\ \ \pi_e \sqcup \pi_p \vDash \varphi_p\ \ \textit{then}\ \ \pi_e' \sqcup \pi_p \nvDash \varphi_p\ \}\end{aligned}$$

By definition of Δ_p, process p must distinguish π_e from π_e', because it cannot respond to both in the same manner. In our running example, Δ_b consists of all pairs of sequences of values of `in` that differ in the first value of `in`. Process b must act differently in such situations: if `in` is initially *true* then b must eventually set `out` to *true*, while if it starts as *false*, then b must keep `out` always set to *false*.

In general, a black-box process p must satisfy its specification φ_p despite having only partial access to O_e. The distinguishability relation therefore directly defines an *information flow* requirement: In order to satisfy φ_p, enough information about O_e must be communicated to p via its local inputs I_p to ensure that p can distinguish any pair of traces in Δ_p. We formalize this information flow assumption as the following 2-hyperproperty, which states that if the outputs of the environment in the two traces must be distinguished, i.e., the projection on O_e is in Δ_p, then there must be a difference in the local inputs I_p:

Definition 2 (Information flow assumption). *The* information flow assumption ψ_p *induced by* Δ_p *is the 2-hyperproperty defined by the relation*

$$R_{\psi_p} = \{(\pi, \pi') \in (2^{\mathcal{V}})^\omega \times (2^{\mathcal{V}})^\omega \mid (\pi{\downarrow_{O_e}}, \pi'{\downarrow_{O_e}}) \in \Delta_p \textit{ then } \pi{\downarrow_{I_p}} \neq \pi'{\downarrow_{I_p}}\}$$

In our running example, the information flow assumption for process b requires that on any two executions that disagree on the initial value of `in`, the values communicated to b over the channel `c` must differ at some point. Observe that the information flow assumption ψ_p specifies neither how the information is to be encoded on `c` nor the point in time when the different communication occurs. However, ψ_p requires that the communication differs eventually if the initial values of `in` are different. Moreover, notice that both Δ_p and ψ_p are determined by p's specification φ_p. The following theorem shows that the information flow assumption ψ_p is a necessary condition, the proof can be found in the full version of this paper [12].

Theorem 1. *Every implementation that satisfies the local specification φ_p for p also satisfies the information flow assumption ψ_p.*

4.2 Time-Bounded Information Flow

We now introduce a strengthened version of the information flow assumption. As shown in Theorem 1, the information flow assumption is a necessary condition for the existence of an implementation that satisfies the specification. Often, however, the information flow assumption is not strong enough to allow for the separate synthesis of individual components in a compositional approach.

Consider again process b in our motivating example. The information flow assumption guarantees that any pair of traces that differ in the initial value of the global input `in` will differ at some point in the value of the channel `c`. This assumption is not strong enough to allow process b to satisfy the specification that b must eventually set `out` to *true* iff the initial value of `in` is *true*. Suppose that `in` is *true* initially. Then b must at some point set `out` to *true*. Process b can only do so when it *knows* that the initial value of `in` is *true*. The information flow assumption is, however, too weak to guarantee that process b will eventually obtain this knowledge. To see this, consider a hypothetical behavior of process a that sets `c` forever to *true*, if `in` is *true* in the first position, and if `in` is *false* then a keeps `c` true for $n-1$ steps, where $n > 0$ is some fixed natural number, before it sets `c` to *false* at the n^{th} step. This behavior of process a satisfies the information flow assumption for any number n; however, without knowing n, process b does not know how many steps it should wait for `in` to become *false*. If, at any point in time t, the channel `c` has not yet been set to *false*, process b can never rule out the possibility that the initial value of `in` is *true*; it might simply be the case that $t < n$ and, hence, the time when `c` will be set to *false* still lies in the future of t! Hence, process b can never actually set `out` to *true*.

To address this, we present a finer version of the distinguishability relation from Definition 1 that we call *time-bounded distinguishability*. Recall that by Definition 1, a pair (π_e, π'_e) is in the distinguishability relation Δ_p if every output sequence π_p for p violates p's specification φ_p when combined with at least one of the input sequences π_e or π'_e. Equivalently, if φ_p is satisfied by π_p combined with π_e, then it is violated when π_p is combined with π'_e. Observe that for p to behave differently in two scenarios, a difference must occur at a finite time t. Clearly, this will only happen if p's input shows a difference in finite time. To capture this, we say that a pair (π_e, π'_e) of environment output sequences is in the *time-bounded* distinguishability relation if the violation with π'_e is guaranteed to happen in finite time. In order to avoid this violation, process p must act in finite time, before the violation occurs on π'_e. We say that a trace π *finitely violates* an LTL formula φ, denoted by $\pi \nvDash_f \varphi$, if there exists a finite prefix w of π such that every (infinite) trace extending w violates φ.

Definition 3 (Time-bounded distinguishability). *Given a local specification φ_p for process p, the* time-bounded distinguishability relation *Λ_p is the set of pairs $(\pi_e, \pi'_e) \in (2^{O_e})^\omega \times (2^{O_e})^\omega$ of traces of global inputs such that every trace of local outputs $\pi_p \in O_p$ either violates the specification φ_p when combined with π_e, or finitely violates p's local specification φ_p when combined with π'_e:*

$$\Lambda_p = \{(\pi_e, \pi'_e) \in (2^{O_e})^\omega \times (2^{O_e})^\omega \mid \\ \forall \pi_p \in (2^{O_p})^\omega.\ \textit{if}\ \ \pi_e \sqcup \pi_p \vDash \varphi_p \ \ \textit{then}\ \ \pi'_e \sqcup \pi_p \nvDash_f \varphi_p \ \}$$

Note that, unlike the distinguishability relation Δ_p, the *time-bounded* distinguishability relation Λ_p is not symmetric: For (π_e, π'_e), the trace $\pi'_e \sqcup \pi_p$ has to finitely violate φ_p, while the trace $\pi_e \sqcup \pi_p$ only needs to violate φ_p in the infinite evaluation. As a result, the corresponding *time-bounded* information flow assumption will also be asymmetric: we require that on input π_e, process p eventually obtains the knowledge that the input is different from π'_e. For input π'_e we do not impose such a requirement. The intuition behind this definition is that on environment output π'_e, process p must definitely produce some output that does *not* finitely violate φ_p. This output can safely be produced without ever knowing that the input is π'_e. However, on input π_e, it becomes necessary for process p to eventually deviate from the output that would work for π'_e. In order to safely do so, p needs to realize after some finite time that the input is not π'_e. In our running example, π_e would be an input in which `in` is initially *true*, while π'_e will be one in which it starts out being *false*.

Suppose we have a function $t : (2^{O_e})^\omega \to \mathbb{N}$ that identifies, for each environment output π_e, the time $t(\pi_e)$ by which process p is guaranteed to know that the environment output is not π'_e. We define the information flow assumption for this particular function t as a 2-hyperproperty. Since we do not know t in advance, the time-bounded information flow assumption is the (infinite) union of all 2-hyperproperties corresponding to the different possible functions t.

Definition 4 (Time-bounded information flow assumption). *Given the time-bounded distinguishability relation Λ_p for process p, the* time-bounded information flow assumption χ_p *for p is the (infinite) union over the 2-hyperproperties induced by the following relations R_t, for all possible functions $t : (2^{O_e})^\omega \to \mathbb{N}$:*

$$R_t = \{(\pi, \pi') \in (2^{\mathcal{V}})^\omega \times (2^{\mathcal{V}})^\omega \mid \\ \textit{if } (\pi{\downarrow_{O_e}}, \pi'{\downarrow_{O_e}}) \in \Lambda_p, \textit{ then } \pi[0...t(\pi{\downarrow_{O_e}})]{\downarrow_{I_p}} \neq \pi'[0...t(\pi{\downarrow_{O_e}})]{\downarrow_{I_p}}\}$$

Unlike the information flow assumption (cf. Theorem 1), the *time-bounded* information flow assumption is not in general a necessary assumption. Consider a modification of our motivating example, where there is an additional environment output `start`, which is only visible to process a, not to process b. The previous specification φ_b is modified so that if `in` is *true* initially, then `out` must be *true* two steps after `start` becomes *true* for the first time; if `in` is *false* initially, then `out` must become *false* after two positions have passed since the first time `start` has become *true*. The specification φ_a ensures that the channel `c` is set to *true* until `start` becomes *true*. Clearly, this is realizable: if `in` is *false* initially, process a sets `c` to *false* once `start` becomes *true*, otherwise `c` stays *true* forever. Process b starts by setting `out` to *true*. It then waits for `c` to become *false*, and, if and when that happens, sets `out` to *false*. In this way, process b accomplishes the correct reaction within two steps after `start` has occurred. However, the function t required by the time-bounded information flow assumption does not exist, because the time of the communication depends on the environment: the prefix needed to distinguish an environment output π_e, where `in` is *true* initially from an environment output π'_e, where `in` is *false* initially, depends on the time when `start` becomes *true* on π'_e.

We now characterize a set of situations in which the time-bounded information flow requirement is still a necessary requirement. For this purpose we consider time-bounded distinguishability relations where the safety violation occurs after a bounded number of steps. We call such time-bounded distinguishability relations *uniform*; the formal definition follows below.

Definition 5 (Uniform distinguishability). *A time-bounded distinguishability relation Λ_p is* uniform *if for every trace $\pi_e \in (2^{O_e})^\omega$ of global inputs, and every trace $\pi_p \in (2^{O_p})^\omega$ of local outputs of p, there exists a natural number $n \in N$ such that for all $\pi'_e \in (2^{O_e})^\omega$ s.t. $(\pi_e, \pi'_e) \in \Lambda_p$ if $\pi_e \sqcup \pi_p \vDash \varphi_p$ then $\pi'_e \sqcup \pi_p \nvDash_n \varphi_p$.*

Theorem 2. *Let Λ_p be a uniform time-bounded distinguishability relation derived from process p's local specification φ_p. Every computation tree that satisfies φ_p also satisfies the time-bounded information flow assumption χ_p.*

The proof of Theorem 2 can be found in the full version of this paper [12]. The relations presented in this section as well as the uniformity check can be represented by and verified with automata, also shown in [12].

5 Compositional Synthesis

We now use the time-bounded information flow assumptions to split the distributed synthesis problem for an architecture $(I_p, I_q, O_p, O_q, I_e)$ into two separate synthesis problems. The local implementations are then composed and form a correct system, whose decomposition returns the solution for each process.

5.1 Constructing the Hyper Implementations

We begin with the synthesis of local processes. Let Λ_p and Λ_q be the time-bounded distinguishability relations for p and q, and let χ_p and χ_q be the resulting time-bounded information flow assumptions. In the individual synthesis problems, we ensure that process p provides the information needed by process q, i.e., that the implementation of p satisfies χ_q, and, similarly, that q provides the information needed by p, i.e., q's implementation satisfies χ_p.

We carry out the individual synthesis of a process implementation on trees that branch according to the input of the process (including $\mathtt{t}_p$) *and* the environment's output. In such a tree, the synthesized process thus has access to full information. We call this tree a *hyper implementation*, rather than an implementation, because the hyper implementation describes how the process will react to certain information, without specifying *how* the process will receive information. This detail is left open until we know the other process' hyper implementation: at that point, both hyper implementations can be turned into standard strategies, which are trees that branch according to the process' own inputs.

Definition 6 (Hyper implementation). *Let p and q be processes and e be the environment. A $2^{O_e \cup I_p \cup \{\mathtt{t}_p\}}$-branching $2^{O_p \cup \{\mathtt{t}_q\}}$-labeled tree h_p is a* hyper implementation *of p.*

Since the hyper implementation has access to the full global information, while the time-bounded information flow assumption only guarantees that the relevant information arrives after some bounded time, the strategy has "too much" information. We compensate for this by introducing a *locality condition*: on two traces $(\pi_e, \pi'_e) \in \Lambda_p$ in the distinguishability relation of process p, as long as the input to the process from the external environment is identical, process p's output must be identical until $\mathtt{t}_p$ happens (which signals that the bound for the transmission of the information has been reached). For traces $(\pi_e, \pi'_e) \notin \Lambda_p$ outside the distinguishability relation, process p's output must be identical until there is a difference in the input to process p or in the value of $\mathtt{t}_p$.

Definition 7 (Locality condition). *Given the time-bounded distinguishability relation Λ_p for process p, the* locality condition *η_p for p is the 2-hyperproperty induced by the following relation R:*

$$\begin{aligned} R = \{(\pi, \pi') \in (2^{O_e \cup I_p \cup \{\mathtt{t}_p\}})^\omega \times (2^{O_e \cup I_p \cup \{\mathtt{t}_p\}})^\omega \mid \\ \textit{if } (\pi{\downarrow_{O_e}}, \pi'{\downarrow_{O_e}}) \in \Lambda_p,\ \textit{then } \pi[0...t]{\downarrow_{O_p}} = \pi'[0...t]{\downarrow_{O_p}} \textit{ and} \\ \textit{if } (\pi{\downarrow_{O_e}}, \pi'{\downarrow_{O_e}}) \notin \Lambda_p,\ \textit{then } \pi[0...t']{\downarrow_{O_p}} = \pi'[0...t']{\downarrow_{O_p}}\} \end{aligned}$$

where t is the smallest natural number such that $\mathtt{t}_p \in \pi[0...t]$ or $\pi[0...t] \downarrow_{I_p} \neq \pi'[t] \downarrow_{I_p}$ (and ∞ if no such t exists), and t' is the smallest natural number such that $\pi[0...t'] \downarrow_{I_p} \neq \pi'[0...t'] \downarrow_{I_p}$ or $\pi[0...t'] \downarrow_{\{\mathtt{t}_p\}} \neq \pi'[0...t'] \downarrow_{\{\mathtt{t}_p\}}$ (and ∞ if no such t' exists).

We now use HyperLTL to formulate the locality condition for process b in our running example. Based on the time-bounded distinguishability relation Λ_b, which relates every trace with $\mathtt{in} =$ true in the first step to all traces on which $\mathtt{in} =$ false holds there, we can write the locality condition:

$$\begin{aligned}&\forall \pi, \pi'.(\mathtt{in}_\pi \wedge \neg \mathtt{in}_{\pi'}) \rightarrow ((\mathtt{t}_\pi \vee \mathtt{c}_\pi \not\leftrightarrow \mathtt{c}_{\pi'})\, \mathcal{R}(\mathtt{out}_\pi \leftrightarrow \mathtt{out}_{\pi'}))\\ &\quad \wedge(\neg(\mathtt{in}_\pi \wedge \neg \mathtt{in}_{\pi'})) \rightarrow (\mathtt{t}_\pi \not\leftrightarrow \mathtt{t}_{\pi'} \vee \mathtt{c}_\pi \not\leftrightarrow \mathtt{c}_{\pi'})\, \mathcal{R}(\mathtt{out}_\pi \leftrightarrow \mathtt{out}_{\pi'}))\end{aligned}$$

The order in the formula is analogous to the order in Definition 7. For all pairs of traces that are in the distinguishability relation, i.e., $\mathtt{in}$ is *true* on π and *false* on π', the outputs being equivalent on both traces can only be released by $\mathtt{t}$ on trace π or by a difference in the local inputs ($\mathtt{c}$). Moreover, if the traces are not in the distinguishability relation, i.e., $\neg(\mathtt{in}_\pi \wedge \neg \mathtt{in}_{\pi'})$, then only a difference in $\mathtt{t}$ or $\mathtt{c}$ can release $\mathtt{out}$ to be equivalent on both traces. With the locality condition at hand, we define when a hyper implementation is locally correct:

Definition 8 (Local correctness of hyper implementations). *Let p and q be processes, let φ_p be the local specification of p, let η_p be its locality condition, and let χ_q be the information flow assumption of q. The hyper implementation h_p of p is* locally correct *if it satisfies φ_p, η_p, and χ_q.*

The specification φ_p is a trace property, while η_p and χ_q are hyperproperties. Since all properties that need to be satisfied by the process are guarantees, it is not necessary to assume explicit behaviour of process q to realize process p. Local correctness relies on the guarantee that the other process satisfies the current process' own information flow assumption. Note that both the locality condition and the information flow assumption for p build on the time-bounded distinguishability relation of p.

5.2 Composition of Hyper Implementations

The hyper implementations of each of the processes are locally correct and satisfy the information flow assumptions of the other process respectively. However, the hyper implementations have full information of the inputs and are dependent on the additional variables $\mathtt{t}_p$ and $\mathtt{t}_q$. To construct practically executable local implementations, we first compose the hyper implementations into one strategy.

Definition 9 (Composition of hyper implementations). *Let p and q be two processes with hyper implementations given as infinite $2^{O_e \cup I_p \cup \{\mathtt{t}_p\}}$-branching $2^{O_p \cup \{\mathtt{t}_q\}}$-labeled tree h_p for process p, and an infinite $2^{O_e \cup I_q \cup \{\mathtt{t}_q\}}$-branching $2^{O_p \cup \{\mathtt{t}_p\}}$-labeled tree h_q for process q.*

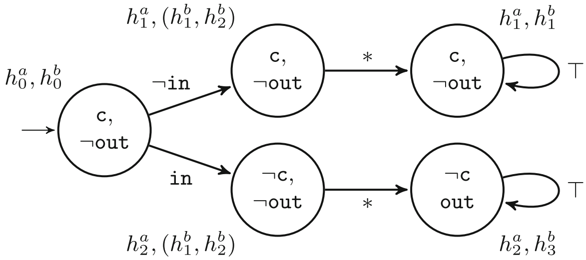


Fig. 2. The composition of the hyper implementations of a in Fig. 1c and b in Fig. 1d. The states are labeled with the combination of states that can be reached for both processes.

Given two hyper implementations h_p and h_q, we define the composition $h = h_p || h_q$ to be a 2^{O_e}-branching $2^{O_p \cup O_q}$-labeled tree, where $h(v) = (h_p(f_p(v)) \cup h_q(f_q(v))) \cap (O_p \cup O_q)$ and f_p, f_q are defined as follows:

$$f_p(\epsilon) = \epsilon \qquad f_p(v \cdot x) = f_p(v) \cdot ((x \cap I_p) \cup (h_q(f_q(v)) \cap (I_p \cup \{\mathtt{t}_p\}))$$
$$f_q(\epsilon) = \epsilon \qquad f_q(v \cdot x) = f_q(v) \cdot ((x \cap I_q) \cup (h_p(f_p(v)) \cap (I_q \cup \{\mathtt{t}_q\}))$$

If each hyper implementation satisfies the time-bounded information flow assumption of the other process, then there exists a strategy for each process (given as a tree that branches according to the local inputs of the process), such that the combined behavior of the two strategies corresponds exactly to the composition of the hyper implementations.

The composition of the hyper implementations of the bit transmission protocol is shown in Fig. 2. The initial state is the combination of both process's initial states with the corresponding outputs. We change the state after the value of `in` is received. While process a directly reacts to `in`, process b cannot observe its value, and the composition can either be in h_0^b or h_1^b. Both states have the same output. In the next step, process a communicates the value of `in` by setting `c` to *true* or *false*, such that the loop states h_1^a, h_1^a and h_2^a, h_3^b are reached.

The local strategies of the processes are constructed from the composed hyper implementations. As an auxiliary notion we introduce the *knowledge set*: the set of finite traces in the composition that cannot be distinguished by a process.

Definition 10 (Knowledge set). *Let p and q be two processes with composed hyper implementations $h = h_p || h_q$. For a finite trace $v \in (2^{I_p})^*$ of inputs to p, we define the knowledge set $K_p(v)$ to be*

$$K_p(v) \triangleq \{w \mid w \textit{ is a finite trace of } (2^{O_e})^* \textit{ and } f_p(w) = v\}.$$

Lemma 1. *For all s $v, v' \in (2^{I_p})^*$, if $K_p(v) = K_p(v')$ then $h(v) \downarrow_{O_p} = h(v') \downarrow_{O_p}$.*

The proof of Lemma 1 can be found in the full version of this paper [12]. The local strategies from the composed hyper implementations are then defined as follows:

Definition 11 (Local strategies from hyper implementations). *Let p and q be two processes with time-bounded information flow assumptions χ_p and χ_q, and $h = h_p || h_q$ be the composition of their hyper implementations. For $j \in \{p, q\}$ the strategy s_j, represented as a 2^{I_j}-branching 2^{O_j}-labeled tree for process j, is defined as follows:*

$$s_j(\epsilon) = \epsilon \qquad s_j(v) = \begin{cases} \emptyset & \text{if } |K_j(v)| = 0 \\ h(min(K_j(v))) \downarrow_{O_j} & \text{if } |K_j(v)| > 0 \end{cases}$$

where $min(K_j(v))$ is the smallest trace based on an arbitrary order over $K_j(v)$.

The base case of the definition inserts a label for unreachable traces in the composed hyper implementation. For example, the local inputs $I_p \backslash O_e$ are determined by s_q, and not all input words in $(2^{I_q})^*$ are possible. Process p's local strategy s_p can discard these input words. The second case of the definition picks the smallest trace in the knowledge set and computes the outputs from h that are local to a process. Intuitively, the outputs of h have to be the same for every trace that a process considers possible in the composed hyper implementations. We therefore pick one of them, compute the output of the composed hyper-strategy, and restrict the output to the local outputs of the process. The following theorem states the correctness of the construction in Definition 11.

Theorem 3. *Let p and q be two processes with time-bounded information flow assumptions χ_p and χ_q, let $h = h_p || h_q$ be the composition of their hyper implementations, and s_p and s_q be their local strategies. Then, for all $v \in (2^{O_e})^*$ it holds that $h(v) = s_p(g_p(v)) \cup s_q(g_q(v))$ where g_p, g_q are defined as follows:*

$$g_p(\epsilon) = \epsilon \qquad g_p(v \cdot x) = g_p(v) \cdot ((x \cap I_p) \cup (s_q(g_q(v)) \cap I_p)$$
$$g_q(\epsilon) = \epsilon \qquad g_q(v \cdot x) = g_q(v) \cdot ((x \cap I_q) \cup (s_p(g_p(v)) \cap I_q)$$

The proof is inductive over the words $v \in (2^{O_e})^*$ and can be found in the full version of this paper [12]. Combining all definitions and theorems of the previous sections, we conclude with the following corollary.

Corollary 1. *Let $(I_p, I_q, O_p, O_q, I_e)$ be an architecture and $\varphi = \varphi_p \wedge \varphi_q$ be a specification. If the hyper-strategies h_p and h_q are locally correct, then the implementation (s_p, s_q) satisfies φ.*

6 A More Practical Approach

A major disadvantage of the synthesis approach of the preceding sections is that the hyper implementations are based on the full set of environment outputs; as a result, hyper implementations branch according to inputs that are not actually available; this, in turn, results in our introduction of the locality condition.

In this section, we develop a more practical approach, where the branching is limited to the information that is actually available to a process: this includes any environment output directly visible to the process and, additionally, the information the process is guaranteed to receive according to the information flow assumption. As a result, the synthesis of the process is sound without need for a locality condition. We develop this approach under two assumptions: First, we assume that the time-bounded information flow assumption only depends on environment outputs the sending process can actually see; second, we assume that the time-bounded information flow assumption can be decomposed into a finite set of classes in the following sense: For a trace π of environment outputs, the information class $[\pi]_p$ describes that, on the trace π, the process p eventually needs to become aware that the current trace is in the set $[\pi]$. The information class is obtained by collecting all traces that are *not* related to π in the time-bounded distinguishability relation.

Definition 12 (Information classes). *Given a time-bounded distinguishability relation Λ_p for process p, the* information class $[\pi]_p$ *of a trace π over O_e is the following set of traces:* $[\pi]_p = (2^{O_e})^\omega \setminus \{\pi' \in (2^{O_e})^\omega \mid (\pi, \pi') \in \Lambda_p\}$

The next definition relativizes the specification of the processes for a particular information class, reflecting the fact that the process does not know the actual environment output, but only its information class; hence, the process output needs to be correct for all environment outputs in the information class.

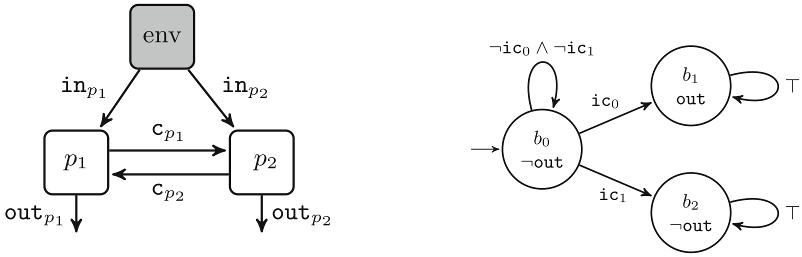
Definition 13 (Relativized specification). *For a process p with specification φ_p and an information class c, the relativized specification $\varphi_{p,c}$ is the following trace property over $(I_p \cap O_e) \cup O_p$:*

$$\varphi_{p,c} = \{\pi_e \sqcup \pi_p \mid \pi_e \in (2^{I_p \cap O_e})^\omega, \pi_p \in (2^{O_p})^\omega \ \textit{s.t.}\ \forall \pi'_e \in c.\ \pi'_e \sqcup \pi_p \vDash \varphi_p\}$$

The component specification, which is the basis for the synthesis of the process, must take into account that the process does not know the information class in advance; the behavior of the other process will only eventually reveal the information class. Let IC be the set of information classes for process p. Assume that this set is finite. We now replace the inputs of the process that come from the other process with new auxiliary input channels IC as new inputs. In the hyper implementation, receiving such an input reveals the information class to the process. In the actual implementation, the information class will be revealed by the actual outputs of the other process that are observable for p. The component specification requires that the processes satisfy the relativized specification under the assumption that the information class is eventually received. We encode this assumption as a trace condition ψ, which requires that exactly one of the elements of IC eventually occurs.

Definition 14 (Component specification). *For process p with specification φ_p, the component specification $\langle\varphi_p\rangle$ over $(I_p \cap O_e) \cup IC \cup O_p$ is defined as*

$$\langle\varphi_p\rangle = \{\pi \in (2^{(I_p \cap O_e) \cup IC \cup O_p})^\omega \mid \ \textit{if}\ \pi \vDash \psi \ \textit{then}\ \pi \vDash \bigwedge_{c \in IC} (\Diamond c \rightarrow \varphi_{p,c}\}$$



(a) (b)

Fig. 3. The architecture used for our experiments in (a) where the number outputs, inputs, and communication channels can vary. Figure 3b shows the implementation of process b for its bit transmission component specification.

where ψ is the following trace property over $(I_p \cap O_e) \cup IC \cup O_p$:

$$\psi = \{\pi \in (2^{(I_p \cap O_e) \cup IC \cup O_p})^\omega \mid \exists \pi' \in (2^{O_e})^\omega.\ \pi \downarrow_{I_p \cap O_e} = \pi' \downarrow_{I_p \cap O_e}$$
$$\textit{and } \pi \vDash \Diamond[\pi'] \textit{ and exactly one element of } IC \textit{ occurs on } \pi\}$$

The component specification allows us to replace the locality condition (Definition 7), which is a hyperproperty, with a trace property. Note, however, that the process additionally needs to satisfy the information flow assumption of the other process, which may in general depend on the full set O_e of environment outputs. This would require us to synthesize the process on the full set O_e, and to re-introduce the locality condition. In practice, however, the information flow assumption of one process often only depends on the information of the other process. In this case, it suffices to synthesize each process based only on the locally visible environment outputs.

Figure 3b shows the implementation of b for its component specification $\langle \varphi_b \rangle$. In contrast to its hyper implementation (cf. Fig. 1b), it does not branch according to `in` and $\mathtt{t}_p$, but only variables in IC. The specification is encoded as the following LTL formula:

$$\langle \varphi_b \rangle = \big(\Box\neg\mathtt{ic}_0 \vee \Box\neg\mathtt{ic}_1\big) \wedge \Diamond\big((\mathtt{ic}_0 \vee \mathtt{ic}_1)\big)$$
$$\rightarrow \big((\Diamond\,\mathtt{ic}_0 \rightarrow \Diamond\,\mathtt{out}) \wedge (\Diamond\,\mathtt{ic}_1 \rightarrow \Box\neg\mathtt{out})\big)$$

The left hand side of the implication represents the assumption ψ, while the right hand side specifies the guarantee for each information class. The composition and decomposition can be performed analogously to the hyper implementations, where we map the value of `ic` to the values of the communication variables. We construct the automata for component specifications in the full version of this paper [12].

7 Experiments

The focus of our experiments is on the performance of the compositional synthesis approach compared to non-compositional synthesis methods for distributed systems. While the time-bounded information flow assumptions and the component specification can be computed automatically by automata constructions, we have, for the purpose of these experiments, built them manually and encoded them as formulas in HyperLTL or LTL, which were then entered to the BoSy/BoSyHyper [11] synthesis tool[1]. Our experiments are based on the following benchmarks:

- **AC.** *Atomic commit.* The atomic commitment protocol specifies that the output of a local process is set to *true* iff the observable input and the unobservable inputs are *true*. We only consider one round of communication, the initial input determines all values. The parameter shows how many input variables each process receives, Par. = 1 for the running example.
- **EC.** *Eventual commit.* The atomic commit benchmark extended to eventual inputs - if all inputs (independently of each other) eventually become *true*, then there needs to be information flow.
- **SA.** *Send all.* Every input of the sender is relevant for the receiver. If an input is set to true, it will eventually be communicated to the receiver. The parameter represents the number of input values and therefore the number of information classes.

Table 1 shows the performance of the compositional synthesis approach. The column architecture (Arch.) determines for each benchmark if the information flow is directional (dir.) or bidirectional (bidir.). Column (Inflow send) indicates the running time for the sending process; where applicable, column (Inflow rec.) indicates the running time for the synthesis of the process that only receives information. We compare the compositional approach to BoSyHyper, based on a standard encoding of distributed synthesis in HyperLTL (Inc. BoSy), and a specialized tool for distributed synthesis [2] (Distr. BoSy). All experiments were performed on a MacBook Pro with a 2,8 GHz Intel Quad Core processor and 16 GB of RAM. The timeout was 30 min.

Information flow guided synthesis outperforms the standard approaches, especially for more complex components. For example, in the atomic commitment benchmark, scaling in the number of inputs does not impact the synthesis of the local processes, while Distr. BoSy eventually times out, and the running time of Inc. BoSy increases faster than for the information flow synthesis. For all approaches, the Send All benchmark is the hardest one to solve. Here, each input that will eventually be set needs to be eventually sent, which leads to non-trivial communication over the shared variables and an increased state space to memorize the individual inputs. Nevertheless, the information flow guided synthesis outperforms the other approaches and times out with parameter 3

[1] The experiments are available at https://doi.org/10.6084/m9.figshare.19697359.

Table 1. The results of the experiments with execution times given in seconds. A cell is highlighted if it was faster than the other approaches, where the sum of synthesis times for both sender and receiver is taken as reference.

Bench.	Arch.	Par.	Inflow send.	Inflow rec.	Distr.BoSy	Inc. BoSy
AC	dir	1	0.92	0.70	**1.41**	2.31
	dir	2	**0.36**	**1.28**	2.86	2.30
	dir	3	**0.92**	**0.68**	2.46	2.55
	dir	4	**0.92**	**0.79**	720.60	3.41
	dir	5	**0.92**	**0.68**	TO	9.27
	bidir	1	1.45	-	**0.96**	9.27
	bidir	2	**2.49**	-	TO	TO
	bidir	3	**79.18**	-	TO	TO
	bidir	4	TO	-	TO	TO
EC	dir	1	0.68	1.87	**0.92**	2.556
	dir	2	0.94	1.85	**0.96**	3.90
	dir	3	**202.09**	TO	TO	TO
	dir	4	TO	TO	TO	TO
	bidir	1	**3.77**	-	4.63	147.46
	bidir	2	TO	-	TO	TO
SA	dir	1	1.31	0.92	2.21	**1.579**
	dir	2	**1.78**	**0.92**	27.47	TO
	dir	3	TO	1.08	TO	TO

because BoSyHyper cannot cope with the number of states needed. Synthesizing a receiver that does not satisfy an information flow assumption is close to irrelevant for every benchmark run. Since these processes are synthesized with local LTL specifications, scaling only in the number of local inputs or information that will eventually be received is easily possible. Notably, these receivers are compatible with any implementation of the sender, whereas the solutions of the other approaches are only compatible for the same synthesis run.

8 Related Work

Compositional synthesis is often studied in the setting of *complete information*, where all processes have access to all environment outputs [9,14,17,19]. In the following, we focus on compositional approaches for the synthesis of distributed systems, where the processes have incomplete information about the environment outputs. Compositionality has been used to improve distributed synthesis in various domains, including reactive controllers [1,16]. Closest to our approach is assume-guarantee synthesis [3,4], which relies on behavioral guarantees of the processs behaviour and assumptions about the behavior of the other processes. Recently, an extension of assume-guarantee synthesis for distributed systems was proposed [20], where the assumptions are iteratively refined. Using a weaker winning condition for synthesis, remorse-free dominance [7] avoids the explicit construction of assumptions and guarantees, resulting in implicit assumptions. A recent approach [13] uses behavioral guarantees in the form of certificates to guide the synthesis process. Certificates specify partial behaviour of each component and are iteratively synthesized. The fundamental difference between all these approaches to the current work is that the assumptions are behavioral. To the best of our knowledge, this is the first synthesis approach based on information-flow assumptions. While there is a rich body of work on the verification of information-flow properties (cf. [8,15,24]), and the synthesis from information-flow properties and other hyperproperties has also been studied before (cf. [11]), the idea of utilizing hyperproperties as assumptions for compositional synthesis of distributed systems is new.

9 Conclusion

The approach introduced in this paper provides the foundation for a new class of distributed synthesis algorithms, where the assumptions refer to the flow of information and are represented as hyperproperties. In many situations, necessary information flow assumptions exist even if there are no necessary behavioral assumptions. There are at least two major directions for future work. The first direction concerns the insight that compositional synthesis profits from the generality of hyperproperties; at the same time, synthesis from hyperproperties is much more challenging than synthesis from trace properties. To address this issue, we have presented the more practical method in Sect. 6, which replaces locality, a hyperproperty, with the component specification, a trace property. However, this method is limited to information flow assumptions that refer to a finite amount of information. It is very common for the required amount of information to be infinite in the sense that the same type of information must be transmitted again and again. We conjecture that our method can be extended to such situations.

A second major direction is the extension to distributed systems with more than two processes. The two-process case has the advantage that the assumptions of one process must be guaranteed by the other. With more than two processes, the localization of the assumptions becomes more difficult or even impossible, if multiple processes have access to the required information.

References

1. Alur, R., Moarref, S., Topcu, U.: Compositional synthesis of reactive controllers for multi-agent systems. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 251–269. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_14
2. Baumeister, J.E.: Encodings of bounded synthesis of distributed systems. B.Sc. Thesis, Saarland University (2017)

3. Bloem, R., Chatterjee, K., Jacobs, S., Könighofer, R.: Assume-guarantee synthesis for concurrent reactive programs with partial information. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 517–532. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_50
4. Chatterjee, K., Henzinger, T.A.: Assume-guarantee synthesis. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 261–275. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_21
5. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
6. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
7. Damm, W., Finkbeiner, B.: Automatic compositional synthesis of distributed systems. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 179–193. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_13
8. Dimitrova, R., Finkbeiner, B., Kovács, M., Rabe, M.N., Seidl, H.: Model checking information flow in reactive systems. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 169–185. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27940-9_12
9. Filiot, E., Jin, N., Raskin, J.-F.: Compositional algorithms for LTL synthesis. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 112–127. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15643-4_10
10. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Proceedings of the 20th ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 321–330 (2005)
11. Finkbeiner, B., Hahn, C., Lukert, P., Stenger, M., Tentrup, L.: Synthesizing reactive systems from hyperproperties. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 289–306. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_16
12. Finkbeiner, B., Metzger, N., Moses, Y.: Information flow guided synthesis (full version) (2022). https://doi.org/10.48550/ARXIV.2205.12085
13. Finkbeiner, B., Passing, N.: Compositional synthesis of modular systems. In: Hou, Z., Ganesh, V. (eds.) ATVA 2021. LNCS, vol. 12971, pp. 303–319. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88885-5_20
14. Finkbeiner, B., Passing, N.: Dependency-based compositional synthesis. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 447–463. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_25
15. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3
16. Hecking-Harbusch, J., Metzger, N.O.: Efficient trace encodings of bounded synthesis for asynchronous distributed systems. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 369–386. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_22
17. Kugler, H., Segall, I.: Compositional synthesis of reactive systems from live sequence chart specifications. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 77–91. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_9

18. Kupferman, O., Vardi, M.Y.: Synthesizing distributed systems. In: Logic in Computer Science (LICS) (2001)
19. Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_6
20. Majumdar, R., Mallik, K., Schmuck, A., Zufferey, D.: Assume-guarantee distributed synthesis. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **39**(11), 3215–3226 (2020). https://doi.org/10.1109/TCAD.2020.3012641
21. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October–1 November 1977, pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
22. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: 31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, 22–24 October 1990, Volume II, pp. 746–757. IEEE Computer Society (1990). https://doi.org/10.1109/FSCS.1990.89597
23. Schewe, S., Finkbeiner, B.: Semi-automatic distributed synthesis. Int. J. Found. Comput. Sci. **18**(1), 113–138 (2007)
24. Yasuoka, H., Terauchi, T.: Quantitative information flow - verification hardness and possibilities. In: Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, 17–19 July 2010, pp. 15–27. IEEE Computer Society (2010). https://doi.org/10.1109/CSF.2010.9