Automatic Repair and Deadlock Detection for Parameterized Systems

Swen Jacobs
CISPA, Saarbrücken, Germany

Mouhammad Sakr
SnT, University of Luxembourg

Marcus Völp
SnT, University of Luxembourg

***Abstract*—We present an algorithm for the repair of parameterized systems. The repair problem is, for a given process implementation, to find a refinement such that a given safety property is satisfied by the resulting parameterized system, and deadlocks are avoided. Our algorithm uses a parameterized model checker to determine the correctness of candidate solutions and employs a constraint system to rule out candidates. We apply this algorithm on systems that can be represented as well-structured transition systems (WSTS), including disjunctive systems, rendezvous systems, and broadcast protocols. Moreover, we show that parameterized deadlock detection can be decided in EXPTIME for disjunctive systems, and that deadlock detection is in general undecidable for broadcast protocols.**

I. Introduction

Concurrent systems are hard to get correct, and are therefore a promising application area for formal methods. For systems that are composed of an *arbitrary* number of processes n, methods such as *parameterized* model checking can provide correctness guarantees that hold regardless of n. While the parameterized model checking problem (PMCP) is undecidable even if we restrict systems to uniform finite-state processes [41], there exist several approaches that decide the problem for specific classes of systems and properties [2]–[4], [15], [20], [22]–[24], [32].

However, if parameterized model checking detects a fault in a given system, it does not tell us how to repair the latter such that it satisfies the specification. To repair the system, the user has to find out which behavior of the system causes the fault, and how it can be corrected. Both tasks may be nontrivial.

For faults in the internal behavior of a process, the approach we propose is based on a similar idea as existing repair approaches [5], [37]: we start with a *non-deterministic* implementation, and restrict non-determinism to obtain a correct implementation. This non-determinism may have been added by a designer to "propose" possible repairs for a system that is known or suspected to be faulty.

However, repairing a process internally will not be enough in the presence of concurrency. We need to go beyond existing repair approaches, and also repair the *communication* between processes to ensure the large number of possible interactions between processes is correct as well. We do so by choosing the right options out of a set of possible interactions, combining the idea above with that of synchronization synthesis [9], [38].

In addition to guaranteeing safety properties, we aim for an approach that avoids introducing *deadlocks*, which is particularly important for a repair algorithm, since often the easiest way to "repair" a system is to let it run into a deadlock as quickly as possible. Unlike non-determinism for repairing internal behavior, we are even able to introduce non-determinism for repairing communication automatically.

Regardless of whether faults are fixed in the internal behavior or in the communication of processes, we aim for a parameterized correctness guarantee, i.e., the repaired implementation should be correct in a system with any number of processes. We show how to achieve this by integrating techniques from parameterized model checking into our repair approach.

High-Level Parameterized Repair Algorithm. Figure 1 sketches the basic idea of our parameterized repair algorithm.

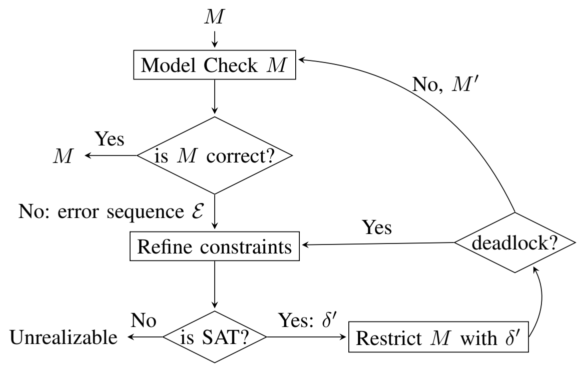

Fig. 1: Parameterized repair of concurrent systems

SJ: remove figure if we need space

The algorithm starts with a representation M of the parameterized system, based on non-deterministic models of the components, and checks if error states are reachable for any size of M. If not, the components are already correct. Otherwise, the parameterized model checker returns an error sequence $\mathcal{E}$, i.e., one or more concrete error paths. $\mathcal{E}$ is then encoded into constraints that ensure that any component that satisfies them will avoid the error paths detected so far. A SAT solver is used to find out if any solution still exists, and if so we restrict M to components that avoid previously found errors. To guarantee that this restriction does not introduce deadlocks, the next step is a parameterized deadlock detection. This provides similar information as the model checker, and is used to refine the constraints if deadlocks are reachable. Otherwise, M' is sent to the parameterized model checker for the next iteration.

Research Challenges. Parameterized model checking in general is known to be undecidable, but different decision procedures exist for certain classes of systems, such as guarded protocols with disjunctive guards (or disjunctive systems) [20], rendezvous systems [32] and broadcast protocols [24]. However, these theoretical solutions are not uniform and do not provide practical algorithms that allow us to extract the information needed for our repair approach. Therefore, the following challenges need to be overcome to obtain an effective parameterized repair algorithm for a broad class of systems:

C1 The parameterized model checking algorithm should be uniform, and needs to provide information about error paths in the current candidate model that allow us to avoid such error paths in future repair candidates.

C2 We need an effective approach for parameterized deadlock detection, preferably supplying similar information as the model checker.

C3 We need to identify an encoding of the discovered information into constraints such that the repair process is sufficiently flexible[1], and sufficiently efficient to handle examples of interesting size.

MV: We show in the below example only a part of our approach but I'm not sure whether we should say sth. like Let us illustrate how our approach deals with communication faults

Parameterized Repair: an Example. Consider a system with one scheduler (Fig. 2) and an arbitrary number of reader-writer processes (Fig. 3), running concurrently and communicating via pairwise rendezvous, i.e., every send actions (e.g. $write!$) needs to synchronize with a receive action (e.g. $write?$) by another process. In this system, multiple processes can be in the $writing$ state at the same time, which must be avoided if they use a shared resource. We want to repair the system by restricting communication of the scheduler.

According to the idea in Fig. 1, the parameterized model checker searches for reachable errors, and it may find that after two sequential $write!$ transitions by different reader-writer processes, they both occupy the $writing$ state at the same time. This information is then encoded into constraints on the behavior of processes, which restrict non-determinism and communication and make the given error path impossible. However, in our example all errors could be avoided by simply removing all outgoing transitions of state $q_{A,0}$ of the scheduler. To avoid such repairs, our algorithm uses *initial constraints* (see section IV) that enforce totality on the transition relation. Another undesirable solution would be the scheduler shown in Fig. 4, because the resulting system will deadlock immediately. This is avoided by checking reachability of deadlocks on candidate repairs. We get a solution that is safe and deadlock-free if we take Fig. 4 and flip all transitions.

Contributions. Our main contribution is a counterexample-guided parameterized repair approach, based on model checking of well-structured transition systems (WSTS) [1], [29]. We investigate which information a parameterized model checker needs to provide to guide the search for candidate repairs, and how this information can be encoded into propositional constraints. Our repair algorithm supports internal repairs and repairs of the communication behavior, while systematically avoiding deadlocks in many classes of systems, including disjunctive systems, rendezvous systems and broadcast protocols.

[1]For example, to allow the user to specify additional properties of the repair, such as keeping certain states reachable.

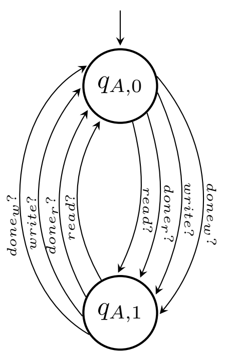

Fig. 2: Scheduler

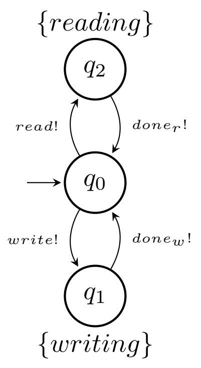

Fig. 3: Reader-Writer

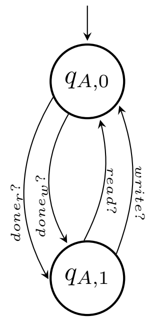

Fig. 4: deadlocked Scheduler

Since existing model checking algorithms for WSTS do not support deadlock detection, our approach has a subprocedure for this problem, which relies on *new theoretical results*: (i) for disjunctive systems, we provide a novel deadlock detection algorithm, based on an abstract transition system, that improves on the complexity of the best known solution; (ii) for broadcast protocols we prove that deadlock detection is in general undecidable, so approximate methods have to be used. We also discuss approximate methods to detect deadlocks in pairwise systems, which can be used as an alternative to the existing approach that has a prohibitive complexity.

Finally, we evaluate an implementation of our algorithm on benchmarks from different application domains, including a distributed lock service and a robot-flocking protocol.

II. System Model

For simplicity, we first restrict our attention to disjunctive systems, other systems will be considered in Sect. V-B. In the following, let Q be a finite set of states.

Processes. A *process template* is a transition system $U = (Q_U, \mathsf{init}_U, \mathcal{G}_U, \delta_U)$, where $Q_U \subseteq Q$ is a finite set of states including the initial state init_U, $\mathcal{G}_U \subseteq \mathcal{P}(Q)$ is a set of guards, and $\delta_U : Q_U \times \mathcal{G}_U \times Q_U$ is a guarded transition relation.

We denote by t_U a transition of U, i.e., $t_U \in \delta_U$, and by $\delta_U(q_U)$ the set of all outgoing transitions of $q_U \in Q_U$. We assume that δ_U is *total*, i.e., for every $q_U \in Q_U$, $\delta_U(q_U) \neq \emptyset$. Define the *size* of U as $|U| = |Q_U|$. An instance of template U will be called a U*-process*.

Disjunctive Systems. Fix process templates A and B with $Q = Q_A \dot\cup Q_B$, and let $\mathcal{G} = \mathcal{G}_A \cup \mathcal{G}_B$ and $\delta = \delta_A \cup \delta_B$. We consider systems $A\|B^n$, consisting of one A-process and n B-processes in an interleaving parallel composition.[2]

The systems we consider are called "disjunctive" since guards are interpreted disjunctively, i.e., a transition with a guard g is enabled if there *exists* another process that is currently in one of the states in g. Figures 5 and 6 give examples of process templates. An example disjunctive

[2]The form $A\|B^n$ is only assumed for simplicity of presentation. Our results extend to systems with an arbitrary number of process templates.

system is $A\|B^n$, where A is the writer and B the reader, and the guards determine which transition can be taken by a process, depending on its own state and the state of other processes in the system. Transitions with the trivial guard $g = Q$ are displayed without a guard. We formalize the semantics of disjunctive systems in the following.

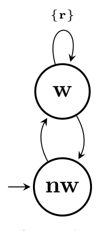

Fig. 5: Writer

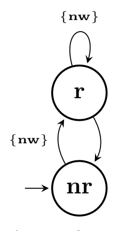

Fig. 6: Reader

Counter System. A *configuration* of a system $A\|B^n$ is a tuple $(q_A, \mathbf{c})$, where $q_A \in Q_A$, and $\mathbf{c} : Q_B \to \mathbb{N}_0$. We identify $\mathbf{c}$ with the vector $(\mathbf{c}(q_0), \ldots, \mathbf{c}(q_{|B|-1})) \in \mathbb{N}_0^{|B|}$, and also use $\mathbf{c}(i)$ to refer to $\mathbf{c}(q_i)$. Intuitively, $\mathbf{c}(i)$ indicates how many processes are in state q_i. We denote by $\mathbf{u}_i$ the unit vector with $\mathbf{u}_i(i) = 1$ and $\mathbf{u}_i(j) = 0$ for $j \neq i$.

Given a configuration $s = (q_A, \mathbf{c})$, we say that the guard g of a local transition $(q_U, g, q'_U) \in \delta_U$ is *satisfied in* s, denoted $s \models_{q_U} g$, if one of the following conditions holds:

(a) $q_U = q_A$, and $\exists q_i \in Q_B$ with $q_i \in g$ and $\mathbf{c}(i) \geq 1$
(A takes the transition, a B-process is in g)
(b) $q_U \neq q_A$, $\mathbf{c}(q_U) \geq 1$, and $q_A \in g$
(B-process takes the transition, A is in g)
(c) $q_U \neq q_A$, $\mathbf{c}(q_U) \geq 1$, and $\exists q_i \in Q_B$ with $q_i \in g$, $q_i \neq q_U$ and $\mathbf{c}(i) \geq 1$
(B-process takes the transition, another B-process is in different state in g)
(d) $q_U \neq q_A$, $q_U \in g$, and $\mathbf{c}(q_U) \geq 2$
(B-process takes the transition, another B-process is in same state in g)

We say that the local transition (q_U, g, q'_U) is *enabled* in s.

Then the *configuration space* of all systems $A\|B^n$, for fixed A, B but arbitrary $n \in \mathbb{N}$, is the transition system $M = (S, S_0, \Delta)$ where:

- $S \subseteq Q_A \times \mathbb{N}_0^{|B|}$ is the set of states,
- $S_0 = \{(init_A, \mathbf{c}) \mid \mathbf{c}(q) = 0 \text{ if } q \neq init_B)\}$ is the set of initial states,
- Δ is the set of transitions $((q_A, \mathbf{c}), (q'_A, \mathbf{c}'))$ s.t. one of the following holds:
  1) $\mathbf{c} = \mathbf{c}' \wedge \exists (q_A, g, q'_A) \in \delta_A : (q_A, \mathbf{c}) \models_{q_A} g$ (transition of A)
  2) $q_A = q'_A \wedge \exists (q_i, g, q_j) \in \delta_B : \mathbf{c}(i) \geq 1 \wedge \mathbf{c}' = \mathbf{c} - \mathbf{u}_i + \mathbf{u}_j \wedge (q_A, \mathbf{c}) \models_{q_i} g$
  (transition of a B-process)

We will also call M the *counter system* (of A and B), and will call configurations *states* of M, or *global states*.

Let $s, s' \in S$ be states of M, and $U \in \{A \cup B\}$. For a transition $(s, s') \in \Delta$ we also write $s \to s'$. If the transition is based on the local transition $t_U = (q_U, g, q'_U) \in \delta_U$, we also write $s \xrightarrow{t_U} s'$ or $s \xrightarrow{g} s'$. Let $\Delta^{local}(s) = \{t_U \mid s \xrightarrow{t_U} s'\}$, i.e., the set of all enabled outgoing local transitions from s, and let $\Delta(s, t_U) = s'$ if $s \xrightarrow{t_U} s'$. From now on we assume wlog. that each guard $g \in \mathcal{G}$ is a singleton.[3]

Runs. A *path* of a counter system is a (finite or infinite) sequence of states $x = s_1, s_2, \ldots$ such that $s_m \to s_{m+1}$ for all $m < |x|$. A *maximal path* is a path that cannot be extended, and a *run* is a maximal path starting in an initial state. We say that a run is *deadlocked* if it is finite. Note that every run $s_1, s_2, \ldots$ of the counter system corresponds to a run of a fixed system $A\|B^n$, i.e., the number of processes does not change during a run. Given a set of error states $E \subseteq S$, an *error path* is a finite path that starts in an initial state and ends in E.

The Parameterized Repair Problem. Let $M = (S, S_0, \Delta)$ be the counter system for process templates $A = (Q_A, \mathsf{init}_A, \mathcal{G}_A, \delta_A)$, $B = (Q_B, \mathsf{init}_B, \mathcal{G}_B, \delta_B)$, and $ERR \subseteq Q_A \times \mathbb{N}_0^{|B|}$ a set of error states. The *parameterized repair problem* is to decide if there exist process templates $A' = (Q_A, \mathsf{init}_A, \mathcal{G}_A, \delta'_A)$ with $\delta'_A \subseteq \delta_A$ and $B' = (Q_B, \mathsf{init}_B, \mathcal{G}_B, \delta'_B)$ with $\delta'_B \subseteq \delta_B$ such that the counter system M' for A' and B' does not reach any state in ERR.

If they exist, we call $\delta' = \delta'_A \cup \delta'_B$ a *repair* for A and B. We call M' the *restriction* of M to δ', also denoted $Restrict(M, \delta')$.

Note that by our assumption that the local transition relations are total, a trivial repair that disables all transitions from some state is not allowed.

We can also figure the complexity of the problem as number of possible repairs is known and complexity (2-NEXPTI as model checker is at most double exponen-

III. Parameterized Model Checking of Disjunctive Systems

In this section, we adress research challenges **C1** and **C2**: after establishing that counter systems can be framed as well-structured transition systems (WSTS) (Sect. III-A), we introduce a parameterized model checking algorithm for disjunctive systems that suits our needs (Sect. III-B), and finally show how the algorithm can be modified to also check for the reachability of deadlocked states (Sect. III-C). Full proofs for the lemmas and theorems in this section can be found in App. A.

A. Counter Systems as WSTS

Well-quasi-order. Given a set of states S, a binary relation $\preceq \subseteq S \times S$ is a *well-quasi-order* (wqo) if $\preceq$ is reflexive, transitive, and if any infinite sequence $s_0, s_1, \ldots \in S^\omega$ contains a pair $s_i \preceq s_j$ with $i < j$. A subset $R \subseteq S$ is an *antichain* if any two distinct elements of R are incomparable wrt. $\preceq$. Therefore, $\preceq$ is a wqo on S if and only if it is well-founded and has no infinite antichains.

Upward-closed Sets. Let $\preceq$ be a wqo on S. The *upward closure* of a set $R \subseteq S$, denoted $\uparrow R$, is the set $\{s \in S \mid \exists s' \in R : s' \preceq s\}$. We say that R is *upward-closed* if $\uparrow R = R$. If R is upward-closed, then we call $B \subseteq S$ a *basis* of R if $\uparrow B = R$. If $\preceq$ is also antisymmetric, then any basis of R has a unique

[3]This is not a restriction as any local transition (q_U, g, q'_U) with a guard $g \in \mathcal{G}$ and $|g| > 1$ can be split into $|g|$ transitions $(q_U, g_1, q'_U), \ldots, (q_U, g_{|g|}, q'_U)$ where for all $i \leq |g| : g_i \in g$ is a singleton guard.

subset of minimal elements. We call this set the *minimal basis* of R, denoted $minBasis(R)$.

Compatibility. Given a counter system $M = (S, S_0, \Delta)$, we say that a wqo $\preceq \subseteq S \times S$ is *compatible* with Δ if the following holds: $\forall s, s', r \in S:$ if $s \rightarrow s'$ and $s \preceq r$ then $\exists r'$ with $s' \preceq r'$ and $r \rightarrow^* r'$. We say $\preceq$ is *strongly compatible* with Δ if the above holds with $r \rightarrow r'$ instead of $r \rightarrow^* r'$.

WSTS [1]. We say that $(M, \preceq)$ with $M = (S, S_0, \Delta)$ is a *well-structured transition system* if $\preceq$ is a wqo on S that is compatible with Δ.

Lemma 1: Let $M = (S, S_0, \Delta)$ be a counter system for process templates A, B, and let $\lessapprox \subseteq S \times S$ be the binary relation defined by:

$$(q_A, \mathbf{c}) \lessapprox (q'_A, \mathbf{d}) \ \Leftrightarrow\ (q_A = q'_A \wedge \mathbf{c} \lesssim \mathbf{d}),$$

where $\lesssim$ is the component-wise ordering of vectors. Then $(M, \lessapprox)$ is a WSTS.

Predecessor, Effective $pred$**-basis [29].** Let $M = (S, S_0, \Delta)$ be a counter system and let $R \subseteq S$. Then the set of *immediate predecessors* of R is

$$pred(R) = \{s \in S \mid \exists r \in R : s \rightarrow r\}.$$

A WSTS $(M, \lessapprox)$ *has effective* $pred$*-basis* if there exists an algorithm that takes as input any finite set $R \subseteq S$ and returns a finite basis of $\uparrow pred(\uparrow R)$. Note that, for a given set $R \subseteq S$ that is upward-closed with respect to $\lessapprox$, $pred(R)$ is upward-closed iff $\lessapprox$ is strongly compatible with Δ.

For backward reachability analysis, we want to compute $pred^*(R)$ as the limit of the sequence $R_0 \subseteq R_1 \subseteq \ldots$ where $R_0 = R$ and $R_{i+1} = R_i \cup pred(R_i)$. Note that if we have strong compatibility and effective pred-basis, we can compute $pred^*(R)$ for any upward-closed set R. If we can furthermore check intersection of upward-closed sets with initial states (which is easy for counter systems), then reachability of arbitrary upward-closed sets is decidable.

The following lemma, like Lemma 1, can be considered folklore. We present it here mainly to show *how* we can effectively compute the predecessors, which is an important ingredient of our model checking algorithm.

Lemma 2: Let $M = (S, S_0, \Delta)$ be a counter system for guarded process templates A, B. Then $(M, \lessapprox)$ has effective $pred$-basis.

B. Model Checking Algorithm

MS: It is important to compute that complexity of the algorithm

Our model checking algorithm is a variant of the known backwards reachability algorithm for WSTS [1]. We present it in detail to show how it stores intermediate results to return an *error sequence*, from which we derive concrete error paths.

Algorithm 1. Given a counter system M and a finite basis ERR of the set of error states, the algorithm iteratively computes the set of predecessors until it reaches an initial state, or a fixed point. The procedure returns either $True$, i.e. the system is safe, or an *error sequence* $E_0, \ldots, E_k$, where $E_0 = ERR$, $\forall 0 < i < k : E_i = minBasis(pred(\uparrow E_{i-1}))$, and $E_k = minBasis(pred(\uparrow E_{k-1})) \cap S_0$. That is, every E_i is the minimal basis of the states that can reach ERR in i steps.

Algorithm 1 Parameterized Model Checking

```
procedure ModelCheck(Counter System $M$,$ERR$)
    $tempSet \leftarrow ERR$, $E_0 \leftarrow ERR$, $i \leftarrow 1$, $visited \leftarrow \emptyset$
// A fixed point is reached if $visited = tempSet$
    while $tempSet \neq visited$ do
        $visited \leftarrow tempSet$
        $E_i \leftarrow minBasis(pred(\uparrow E_{i-1}))$
//$pred$ is computed as in the proof of Lemma 2
        if $E_i \cap S_0 \neq \emptyset$ then    //intersect with initial states?
            return $False, \{E_0, \ldots, E_i \cap S_0\}$
        $tempSet \leftarrow minBasis(visited \cup E_i)$
        $i \leftarrow i + 1$
    return $True, \emptyset$
```

Example. Consider the reader-writer system in Figures 5 and 6. Suppose the error states are all states where the writer is in w while a reader is in r. In other words, the error set of the corresponding counter system M is $\uparrow E_0$ where $E_0 = \{(w, (0, 1))\}$ and $(0, 1)$ means zero reader-processes are in nr and one in r. Note that $\uparrow E_0 = \{(w, (i_0, i_1)) \mid (w, (0, 1)) \lessapprox (w, (i_0, i_1))\}$, i.e. all elements with the same w, $i_0 \geq 0$ and $i_1 \geq 1$. If we run Algorithm 1 with the parameters $M, \{(w, (0, 1))\}$, we get the following error sequence: $E_0 = \{(w, (0, 1))\}$, $E_1 = \{(nw, (0, 1))\}$, $E_2 = \{(nw, (1, 0))\}$, with $E_2 \cap S_0 \neq \emptyset$, i.e., the error is reachable.

Properties of Algorithm 1. Correctness of the algorithm follows from the correctness of the algorithm by Abdulla et al. [1], and from Lemma 2. Termination follows from the fact that a non-terminating run would produce an infinite minimal basis, which is impossible since a minimal basis is an antichain.

SJ: mention complexity here?

C. Deadlock Detection in Disjunctive Systems

The repair of concurrent systems is much harder than fixing monolithic systems. One of the sources of complexity is that a repair might introduce a deadlock, which is usually an unwanted behavior. In this section we show how we can detect deadlocks in disjunctive systems.

A set of deadlocked states is in general not upward-closed under $\lessapprox$ (defined in in Sect. III-A): let $s = (q_A, \mathbf{c}), r = (q_A, \mathbf{d})$ be global states with $s \lessapprox r$. If s is deadlocked, then $\mathbf{c}(i) = 0$ for every q_i that appears in a guard of an outgoing local transition from s. Now if $\mathbf{d}(i) > 0$ for one of these q_i, then some transition is enabled in r, which is therefore not deadlocked.

A natural idea is to refine the wqo such that deadlocked states are upward closed. To this end, consider $\lesssim_0 \subseteq \mathbb{N}_0^{|B|} \times \mathbb{N}_0^{|B|}$ where

$$\mathbf{c} \lesssim_0 \mathbf{d} \ \Leftrightarrow\ (\mathbf{c} \lesssim \mathbf{d} \wedge \forall i \leq |B| : (\mathbf{c}(i) = 0 \Leftrightarrow \mathbf{d}(i) = 0)),$$

and $\lessapprox_0 \subseteq S \times S$ where $(q_A, \mathbf{c}) \lessapprox_0 (q'_A, \mathbf{d}) \Leftrightarrow (q_A = q'_A \wedge \mathbf{c} \lesssim_0 \mathbf{d})$.

Then, deadlocked states are upward closed with respect to $\lesssim_0$. However, it is not easy to adopt the WSTS approach to this case, since for our counter systems $pred(R)$ will in general not be upward closed if R is upward closed. Instead of using $\lesssim_0$ to define a WSTS, in the following we will use it to define a counter abstraction (similar to the approach of Pnueli et al. [40]) that can be used for deadlock detection.

The idea is that we use vectors with counter values from $\{0,1\}$ to represent their upward closure with respect to $\lesssim_0$. These upward closures will be seen as abstract states, and in the usual way define that a transition between abstract states $\hat{s}, \hat{s}'$ exists iff there exists a transition between concrete states $s \in \uparrow\hat{s}, s' \in \uparrow\hat{s}'$. We formalize the abstract system in the following, assuming wlog. that δ_B does not contain transitions of the form $(q_i, \{q_i\}, q_j)$, i.e., transitions from q_i that are guarded by q_i.[4]

01-**Counter System.** For a given counter system M, we define the 01-*Counter System* $\hat{M} = (\hat{S}, \hat{s_0}, \hat{\Delta})$, where:

- $\hat{S} \subseteq Q_A \times \{0,1\}^{|B|}$ is the set of states,
- $\hat{s}_0 = (\mathsf{init}_A, \mathbf{c})$ with $\mathbf{c}(q) = 1$ iff $q = \mathsf{init}_B$ is the initial state,
- $\hat{\Delta}$ is the set of transitions $((q_A, \mathbf{c}), (q'_A, \mathbf{c}'))$ s.t. one of the following holds:
  1) $\mathbf{c} = \mathbf{c}' \wedge \exists (q_A, g, q'_A) \in \delta_A : (q_A, \mathbf{c}) \models_{q_A} g$ (transition of A)
  2) $q_A = q'_A \wedge \exists (q_i, g, q_j) \in \delta_B : (q_A, \mathbf{c}) \models_{q_i} g \wedge \mathbf{c}(i) = 1 \wedge$ $[(\mathbf{c}(j) = 0 \wedge (\mathbf{c}' = \mathbf{c} - \mathbf{u}_i + \mathbf{u}_j \vee \mathbf{c}' = \mathbf{c} + \mathbf{u}_j)) \vee$ $(\mathbf{c}(j) = 1 \wedge (\mathbf{c}' = \mathbf{c} - \mathbf{u}_i \vee \mathbf{c}' = \mathbf{c}))]$ (transition of a B-process)

Define runs and deadlocks of a 01-counter system similarly as for counter systems. For a state $s = (q_A, \mathbf{c})$ of M, define the corresponding abstract state of $\hat{M}$ as $\alpha(s) = (q_A, \hat{\mathbf{c}})$ with $\hat{\mathbf{c}}(i) = 0$ if $\mathbf{c}(i) = 0$, and $\hat{\mathbf{c}} = 1$ otherwise.

Theorem 1: The 01-counter system $\hat{M}$ has a deadlocked run if and only if the counter system M has a deadlocked run.

Corollary 1: Deadlock detection in disjunctive systems is decidable in EXPTIME (in $|Q_B|$).

An Algorithm for Deadlock Detection. Now we can modify Algorithm 1 to detect deadlocks in a 01-counter system $\hat{M}$: instead of passing a basis of the set of errors in the parameter ERR, we pass a finite set of deadlocked states $DEAD \subseteq \hat{S}$. Furthermore, in Line 5 we now compute $pred(E_{i-1})$ instead of $minBasis(pred(\uparrow E_{i-1}))$, and in Line 9 we need to replace $minBasis(visitedSet \cup E_i)$ with $(visitedSet \cup E_i)$.

IV. Parameterized Repair Algorithm

Now, we can introduce a parameterized repair algorithm that interleaves the backwards model checking algorithm (Algorithm 1) with a forward reachability analysis and the computation of candidate repairs.

[4]A system that does not satisfy this assumption can easily be transformed into one that does, with a linear blowup in the number of states, and preserving reachability properties including reachability of deadlocks.

Forward Reachability Analysis. In the following, for a set $R \subseteq S$, let $Succ(R) = \{s' \in S \mid \exists s \in R : s \rightarrow s'\}$. Furthermore, for $s \in S$, let $\Delta^{local}(s, R) = \{t_U \in \delta \mid t_U \in \Delta^{local}(s) \wedge \Delta(s, t_U) \in R\}$.

Given an error sequence $E_0, \ldots, E_k$, let the *reachable error sequence* $\mathcal{RE} = RE_0, \ldots, RE_k$ be defined by $RE_k = E_k$ (which by definition only contains initial states), and $RE_{i-1} = Succ(RE_i) \cap \uparrow E_{i-1}$ for $1 \leq i \leq k$. That is, each RE_i contains a set of states that can reach $\uparrow ERR$ in i steps, and are reachable from S_0 in $k - i$ steps. Thus, it represents a set of concrete error paths of length k.

Constraint Solving for Candidate Repairs. The generation of candidate repairs is guided by constraints over the local transitions δ as atomic propositions, such that a satisfying assignment of the constraints corresponds to the candidate repair, where only transitions that are assigned true remain in δ'. During an execution of the algorithm, these constraints ensure that all error paths discovered so far will be avoided, and include a set of fixed constraints that express additional desired properties of the system, as explained in the following.

Initial Constraints. To avoid the construction of repairs that violate the totality assumption on the transition relations of the process templates, every repair for disjunctive systems has to satisfy the following constraint:

$$TRConstr_{Disj} = \bigwedge_{q_A \in Q_A} \bigvee_{t_A \in \delta_A(q_A)} t_A \wedge \bigwedge_{q_B \in Q_B} \bigvee_{t_B \in \delta_B(q_B)} t_B$$

Informally, $TRConstr_{Disj}$ guarantees that a candidate repair returned by the SAT solver never removes all local transitions of a local state in $Q_A \cup Q_B$. Furthermore a designer can add constraints that are needed to obtain a repair that conforms with their requirements, for example to ensure that certain states remain reachable in the repair (see Appendix D-A and D-B for more examples).

Algorithm 2. Given a counter system M, a basis ERR of the error states, and initial Boolean constraints $initConstr$ on the transition relation (including at least $TRConstr_{Disj}$), the algorithm returns either a *repair* δ' or the string $Unrealizable$ to denote that no repair exists.

Properties of Algorithm 2.

Theorem 2 (Soundness): For every repair δ' returned by Algorithm 2:

- $Restrict(M, \delta')$ is safe, i.e., $\uparrow ERR$ is not reachable, and
- the transition relation of $Restrict(M, \delta')$ is total in the first two arguments.

Proof: The parameterized model checker guarantees that the transition relation is safe, i.e., $\uparrow ERR$ is not reachable. Moreover, the transition relation constraint $TRConstr$ is part of $initConstr$ and guarantees that, for any candidate repair returned by the SAT solver, the transition relation is total. ■

Theorem 3 (Completeness): If Algorithm 2 returns "Unrealizable", then the parameterized system has no repair.

Algorithm 2 Parameterized Repair

```
1:  procedure ParamRepair($M$, $ERR$, $InitConstr$)
2:      $M' \leftarrow M$, $accCnstr \leftarrow True$, $isCorrect \leftarrow False$
3:      while $isCorrect = False$ do
4:          $isCorrect, [E_0, \dots, E_k] \leftarrow MC(M', ERR)$
5:          if $isCorrect = False$ then
6:              $RE_k \leftarrow E_k$  //$E_k$ contains only initial states
7:              $RE_{k-1} \leftarrow Succ(RE_k) \cap \uparrow E_{k-1}, \dots,$
                $RE_0 \leftarrow Succ(RE_1) \cap \uparrow E_0$
8:  //for every initial state in $RE_k$ compute the its constraints
9:              $newConstr \leftarrow \bigwedge_{s \in RE_k}$
                $BuildConstr(s, [RE_{k-1}, \dots, RE_0]\})$
10: //accumulate iterations' constraints
11:             $accCnstr \leftarrow newConstr \wedge accCnstr$
12:             $\delta', isSAT \leftarrow SAT(accCnstr \wedge initConstr)$
13:             if $isSAT = False$ then
14:                 return $Unrealizable$
                //compute a new candidate using the repair $\delta'$
15:             $M' = Restrict(M, \delta')$
16:         else return $\delta'$    //a repair is found!

1:  procedure BuildConstr(State $s$, $\mathcal{RE}$)
2:      //$s$ is a state, $\mathcal{RE}[1:]$ is a list obtained by removing
    the first element from $\mathcal{RE}$
3:      if $\mathcal{RE}[1:]$ is empty then
    //if $t_U \in \Delta^{local}(s)$ leads directly to error set, delete it
4:          return $\bigwedge_{t_U \in \Delta^{local}(s, \mathcal{RE}[0])} \neg t_U$
5:      else
    //else delete $t_U$ or all transitions to next error level
6:          return $\bigwedge_{t_U \in \Delta^{local}(s, \mathcal{RE}[0])} (\neg t_U \vee$
                $BuildConstr(\Delta(s, t_U), \mathcal{RE}[1:]))$
```

Proof: Algorithm 2 returns "Unrealizable" if $accCnstr \wedge initConstr$ has become unsatisfiable. We consider an arbitrary $\delta' \subseteq \delta$ and show that it cannot be a repair. Note that for the given run of the algorithm, there is an iteration i of the loop such that δ', seen as an assignment of truth values to atomic propositions δ, was a satisfying assignment of $accCnstr \wedge initConstr$ up to this point, and is not anymore after this iteration.

If $i = 0$, i.e., δ' was never a satisfying assignment, then δ' does not satisfy $initConstr$ and can clearly not be a repair. If $i > 0$, then δ' is a satisfying assignment for $initConstr$ and all constraints added before round i, but not for the constraints $\bigwedge_{s \in RE_k} BuildConstr(s, [RE_{k-1}, \dots, RE_0]\})$ added in this iteration of the loop, based on a reachable error sequence $\mathcal{RE} = RE_k, \dots, RE_0$. By construction of $BuildConstr$, this means we can construct out of δ' and $\mathcal{RE}$ a concrete error path in $Restrict(M, \delta')$, and δ' can also not be a repair. ■

Theorem 4 (Termination): Algorithm 2 always terminates.

Proof: For a counter system based on A and B, the number of possible repairs is bounded by $2^{|\delta|}$. In every iteration of the algorithm, either the algorithm terminates, or it adds constraints that exclude at least the repair that is currently under consideration. Therefore, the algorithm will always terminate. ■

Parameterized Repair with Deadlock Detection. Note that Algorithm 2 does not include any measures that prevent it from producing a repair with deadlocked runs. However, it can be extended with a subprocedure for deadlock detection based on the approach explained in Sect. III-C, called in an interleaving way with the model checker as depicted in Fig. 1.

V. Extensions

A. Beyond Reachability

Algorithm 2 can also be used for repair with respect to general safety properties, based on the automata-theoretic approach to model checking. We assume that the reader is familiar with finite-state automaton and with the automata-theoretic approach to model checking.

Checking Safety Properties. Let $M = (S, S_0, \Delta)$ be a counter system of process templates A and B that violates a safety property φ over the states of A, and let $\mathcal{A} = (Q^{\mathcal{A}}, q_0^{\mathcal{A}}, Q_A, \delta^{\mathcal{A}}, \mathcal{F})$ be the automaton that accepts all words over Q_A that violate φ. To repair M, the composition $M \times \mathcal{A}$ and the set of error states $ERR = \{((q_A, \mathbf{c}), q_{\mathcal{F}}^{\mathcal{A}}) \mid (q_A, \mathbf{c}) \in S \wedge q_{\mathcal{F}}^{\mathcal{A}} \in \mathcal{F}\}$ can be given as inputs to the procedure $ParamRepair$.

Corollary 1: Let $\lesssim_{\mathcal{A}} \subseteq (M \times \mathcal{A}) \times (M \times \mathcal{A})$ be a binary relation defined by:

$$((q_A, \mathbf{c}), q^{\mathcal{A}}) \lesssim_{\mathcal{A}} ((q'_A, \mathbf{c}'), q'^{\mathcal{A}}) \Leftrightarrow \mathbf{c} \lesssim \mathbf{c}' \wedge q_A = q'_A \wedge q^{\mathcal{A}} = q'^{\mathcal{A}}$$

then $((M \times \mathcal{A}), \lesssim_{\mathcal{A}})$ is a WSTS with effective *pred*-basis.

Similarly, the algorithm can be used for any safety property $\varphi(A, B^{(k)})$ over the states of A, and of k B-processes. To this end, we consider the composition $M \times B^k \times \mathcal{A}$ with $M = (S, S_0, \Delta)$, $B = (Q_B, \mathsf{init}_B, \mathcal{G}_B, \delta_B)$, and $\mathcal{A} = (Q^{\mathcal{A}}, q_0^{\mathcal{A}}, Q_A \times Q_{B^k}, \delta^{\mathcal{A}}, \mathcal{F})$ is the automaton that reads states of $A \times B^k$ as actions and accepts all words that violate the property.[5]

Example. Consider again the simple reader-writer system in Figures 5 and 6, and assume that instead of local transition $(nr, \{nw\}, r)$ we have an unguarded transition (nr, Q, r). We want to repair the system with respect to the safety property $\varphi = G[(w \wedge nr_1) \implies (nr_1 W nw)]$ where G, W are the temporal operators *always* and *weak until*, respectively. Figure 7 depicts the automaton equivalent to $\neg\varphi$. To repair the system we first need to split the guards as mentioned in Section II, i.e., (nr, Q, r) is split into $(nr, \{nr\}, r), (nr, \{r\}, r), (nr, \{nw\}, r)$, and $(nr, \{w\}, r)$. Then we consider the composition $\mathcal{C} = M \times B \times \mathcal{A}$ and we run Algorithm 2 on the parameters $\mathcal{C}$, $((-, -, (*, *), q_2^{\mathcal{A}}))$ (where $(-, -)$ means any writer state and any reader state, and $*$ means 0 or 1), and $TRConstr_{Disj}$. The model checker in Line 4 may return the following error sequences, where we only consider states that didn't occur before:

[5]By symmetry, property $\varphi(A, B^{(k)})$ can be violated by these k explicitly modeled processes iff it can be violated by any combination of k processes in the system.

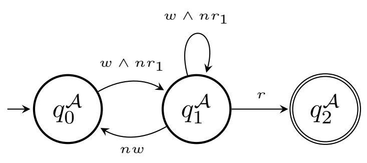

Fig. 7: Automaton for $\neg\varphi$

$E_0 = \{((-,-,(*,*)), q_2^A)\}$,
$E_1 = \{((w, r_1, (0,0)), q_1^A)\}$,
$E_2 = \{((w, nr_1, (0,0)), q_0^A), ((w, nr_1, (0,1)), q_0^A), ((w, nr_1, (1,0)), q_0^A)\}$,
$E_3 = \{((nw, nr_1, (0,0)), q_0^A), ((nw, nr_1, (0,1)), q_0^A), ((w, r_1, (0,0)), q_0^A), ((w, r_1, (0,1)), q_0^A), ((w, r_1, (1,0)), q_0^A)\}$

In Line 12 we find out that the error sequence can be avoided if we remove the transitions $\{(nr, \{nr\}, r), (nr, \{r\}, r), (nr, \{w\}, r)\}$. Another call to the model checker in Line 4 finally assures that the new system is safe. Note that some states were omitted from error sequences in order to keep the presentation simple.

B. Beyond Disjunctive Systems

Furthermore, we have extended Algorithm 2 to other systems that can be framed as WSTS, in particular rendezvous systems [32] and systems based on broadcasts or other global synchronizations [24], [34]. We summarize our results here, more details can be found in Appendix D.

Both types of systems are known to be WSTS, and there are two remaining challenges:

1) how to find suitable constraints to determine a restriction δ', and
2) how to exclude deadlocks.

The first is relatively easy, but the constraints become more complicated because we now have synchronous transitions of multiple processes. Deadlock detection is decidable for rendezvous systems, but the best known method is by reduction to reachability in VASS [32], which has recently been shown to be TOWER-hard [17]. For broadcast protocols we can show that the situation is even worse:

Theorem 5: Deadlock detection in broadcast protocols is undecidable.

a) Proof idea.: Reachability in affine VASS has recently been shown to be undecidable in almost all cases, including the case where all transitions use broadcast matrices [11]. We can reduce this undecidable problem to deadlock detection in broadcast protocols by modifying the construction of German and Sistla [32] for reducing reachability in (non-affine) VASS to deadlock detection in rendezvous systems. A full proof with the modified construction is given in App. D-B

Approximate Methods for Deadlock Detection. Since solving the problem exactly is impractical or impossible in general, we propose to use approximate methods. For rendezvous systems, the 01-counter system introduced as a precise abstraction for disjunctive systems in Sect. III-C can also be used, but in this case it is not precise, i.e., it may produce spurious deadlocked runs. Another possible overapproximation is a system that simulates pairwise transitions by a pair of disjunctive transitions. For broadcast protocols we can use lossy broadcast systems, for which the problem is decidable [18].[6] Another alternative is to add initial constraints that restrict the repair algorithm and imply deadlock-freedom.

VI. Implementation & Evaluation

We have implemented a prototype of our parameterized repair algorithm that supports the three types of systems (disjunctive, pairwise and broadcast), and safety and reachability properties. For disjunctive and pairwise systems, we have evaluated it on different variants of reader-writer-protocols, based on to the ones given in Sect. I,II, where we replicated some of the states and transitions to test the performance of our algorithm on bigger benchmarks. For disjunctive systems, all variants have been repaired successfully in less than 2s. For pairwise systems, these benchmarks are denoted "RWi (PR)" in Table I, and a detailed treatment of one version, including an explanation of the whole repair process is given in Appendix E.

can we compare to related work in the literature?

For broadcast protocols, we have evaluated our algorithm on a range of more complex benchmarks taken from the parameterized verification literature [35]: a distributed **Lock Service** (DLS) inspired by the Chubby protocol [12], a distributed **Robot Flocking** protocol (RF) [13], a distributed **Smoke Detector** (SD) [34], a sensor network implementing a **Two-Object Tracker** (2OT) [14], and the cache coherence protocol **MESI** [21] in different variants (Appendix F includes details of this benchmark and its repair process).

Typical desired safety properties are mutual exclusion and similar properties. Since deadlock detection is undecidable for broadcast protocols, the absence of deadlocks needs to be ensured with additional initial constraints.

one reviewer asked to compare different encodings of error paths (s.t. formula is in CNF)

On all benchmarks, we compare the performance of our algorithm based on the valuations of two flags: SEP and EPT. The SEP ("single error path") flag indicates that, instead of encoding all the model checker's computed error paths, only one path is picked and encoded for SAT solving. When the EPT ("error path transitions") flag is raised the SAT formula is constructed so that only transitions on the extracted error paths may be suggested for removal. Note that in the default case, even transitions that are unrelated to the error may be removed. Table I summarizes the experimental results we obtained.

We note that the algorithm deletes fewer transitions when the EPT flag is raised (EPT=T). This is because we tell the SAT solver explicitly not to delete transitions that are not on the error paths. Removing fewer transitions is desirable in applications where we do not want to restrict the repaired protocol more than necessary. We observe the best performance when the SEP flag is set to true (SEP=T) and the EPT flag is false. This is because the constructed SAT formulas are much simpler and the SAT solver has more freedom in deleting transitions, resulting in a small number of iterations.

[6] Note that in the terminology of Delzanno et al., deadlock detection is a special case of the TARGET problem.

TABLE I: Running time, number of iterations, and number of deleted transitions (#D.T.) for the different configurations. Each benchmark is listed with its number of local states, and edges. We evaluated the algorithms on different sets of errors with $P_1 \cup P_2 = C$. Smallest number of iterations, runtime per benchmark, deleted transitions are highlighted in boldface.

Benchmark	**Size**		**Errors**	**[SEP=F & EPT=F]**			**[SEP=T & EPT=F]**			**[SEP=F & EPT=T]**			**[SEP=T & EPT=T]**		
	States	Edges		#Iter	Time	#D.T.	#Iter	Time	#D.T.	#Iter	Time	#D.T.	#Iter	Time	#D.T.
RW1 (PW)	5	12	C	3	2.5	4	3	2.9	4	**2**	**1.7**	**2**	**2**	**1.7**	**2**
RW2 (PW)	15	42	C	3	3.8	14	3	4.8	14	**2**	**3.2**	**7**	7	8.4	**7**
RW3 (PW)	35	102	C	3	820.7	34	3	**7.6**	34	**2**	552.3	**17**	17	40.3	**17**
RW4 (PW)	45	132	C	TO	TO	TO	**3**	**11.8**	44	TO	TO	TO	22	99.2	**22**
DLS	10	95	P1	**1**	**0.8**	13	**1**	**0.8**	13	3	2.4	**5**	5	5.6	**5**
DLS	10	95	P2	**1**	**0.8**	13	2	1.7	13	3	2.6	**9**	7	5.5	**9**
DLS	10	95	C	**2**	4.2	13	**2**	**1.5**	13	3	3	**9**	9	8.1	**9**
RF	10	147	P1	**1**	2.5	32	**1**	**1.2**	32	TO	TO	TO	8	12.4	**13**
RF	10	147	P2	**1**	**1.2**	32	**1**	1.3	32	TO	TO	TO	8	11.3	**14**
RF	10	147	C	**1**	7.8	32	**1**	**1.4**	32	TO	TO	TO	8	12.5	**12**
SD	6	39	C	**1**	**1**	**4**	**1**	**1**	**4**	3	2.4	**4**	3	3	**4**
2OT	12	128	P1	12	18.8	26	**6**	**8.3**	26	16	73.8	**17**	16	34	**17**
2OT	12	128	P2	**1**	**1.8**	26	**1**	**1.8**	26	4	2958	**11**	8	16.5	12
2OT	12	128	C	11	17.2	Unreal.	**6**	**11.7**	Unreal.	TO	TO	TO	11	48.6	Unreal.
MESI1	4	26	C	**1**	2.4	6	**1**	**0.9**	6	2	1.8	**5**	4	3.5	**5**
MESI2	9	71	C	**1**	**1.1**	26	**1**	**1.1**	26	3	56.4	20	6	6.8	**15**
MESI3	14	116	C	**1**	109.4	46	**1**	**108.1**	46	TO	TO	TO	6	289.9	**15**

VII. Related Work

Many automatic repair approaches have been considered in the literature, most of them restricted to monolithic systems [5], [19], [30], [33], [37], [39]. Additionally, there are several approaches for synchronization synthesis and repair of *concurrent systems*. Some of them differ from ours in the underlying approach, e.g., being based on automata-theoretic synthesis [7], [28]. Others are based on a similar underlying counterexample-guided synthesis/repair principle, but differ in other aspects from ours. For instance, there are approaches that repair the program by adding atomic sections, which forbid the interruption of a sequence of program statements by other processes [9], [42]. *Assume-Guarantee-Repair* [31] combines verification and repair, and uses a learning-based algorithm to find counterexamples and restrict transition guards to avoid errors. In contrast to ours, this algorithm is not guaranteed to terminate. From *lazy synthesis* [27] we borrow the idea to construct the set of *all* error paths of a given length instead of a single concrete error path, but this approach only supports systems with a fixed number of components. Some of these existing approaches are more general than ours in that they support certain infinite-state processes [9], [31], [42], or more expressive specifications and other features like partial information [7], [28].

The most important difference between our approach and all of the existing repair approaches is that, to the best of our knowledge, none of them provide correctness guarantees for systems with a parametric number of components. This includes also the approach of McClurg et al. [38] for the synthesis of synchronizations in a software-defined network. Although they use a variant of Petri nets as a system model, which would be suitable to express parameterized systems, their restrictions are such that the approach is restricted to a fixed number of components. In contrast, we include a parameterized model checker in our repair algorithm, and can therefore provide parameterized correctness guarantees. There exists a wealth of results on parameterized model checking, collected in several good surveys recently [10], [16], [25].

VIII. Conclusion and Future Work

We have investigated the parameterized repair problem for systems of the form $A \| B^n$ with an arbitrary $n \in \mathbb{N}$. We introduced a general parameterized repair algorithm, based on interleaving the generation of candidate repairs with parameterized model checking and deadlock detection, and instantiated this approach to different classes of systems that can be modeled as WSTS: disjunctive systems, pairwise rendezvous systems, and broadcast protocols.

Since deadlock detection is an important part of our method, we investigated this problem in detail for these classes of systems, and found that the problem can be decided in EXPTIME for disjunctive systems, and is undecidable for broadcast protocols.

Besides reachability properties and the absence of deadlocks, our algorithm can guarantee general safety properties, based on the automata-theoretic approach to model checking. On a prototype implementation of our algorithm, we have shown that it can effectively repair non-deterministic overapproximations of many examples from the literature. Moreover, we have evaluated the impact of different heuristics or design choices on the performance of our algorithm in terms of repair time, number of iterations, and number of deleted transitions.

A limitation of the current algorithm is that it cannot guarantee any *liveness properties*, like termination or the absence of undesired loops. Also, it cannot automatically *add behavior* (states, transitions, or synchronization options) to the system, in case the repair for the given input is unrealizable. We consider these as important avenues for future work. Moreover, in order to improve the practicality of our approach we want to examine the inclusion of symbolic techniques for counter abstraction [8], and advanced parameterized model checking techniques, e.g., *cutoff* results for disjunctive systems [6], [22], [36], or recent *pruning* results for immediate observation Petri nets, which model exactly the class of disjunctive systems [26].

References

[1] Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: Proceedings 11th Annual IEEE Symposium on Logic in Computer Science. pp. 313–321. IEEE (1996)

[2] Aminof, B., Jacobs, S., Khalimov, A., Rubin, S.: Parameterized model checking of token-passing systems. In: VMCAI. LNCS, vol. 8318, pp. 262–281. Springer (2014)

[3] Aminof, B., Kotek, T., Rubin, S., Spegni, F., Veith, H.: Parameterized model checking of rendezvous systems. In: CONCUR. LNCS, vol. 8704, pp. 109–124. Springer (2014)

[4] Aminof, B., Rubin, S.: Model checking parameterised multi-token systems via the composition method. In: IJCAR. LNCS, vol. 9706, pp. 499–515. Springer (2016)

[5] Attie, P.C., Bab, K.D.A., Sakr, M.: Model and program repair via sat solving. ACM Transactions on Embedded Computing Systems (TECS) **17**(2), 1–25 (2017)

[6] Außerlechner, S., Jacobs, S., Khalimov, A.: Tight cutoffs for guarded protocols with fairness. In: VMCAI. LNCS, vol. 9583, pp. 476–494. Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_23

[7] Bansal, S., Namjoshi, K.S., Sa'ar, Y.: Synthesis of coordination programs from linear temporal specifications. Proc. ACM Program. Lang. **4**(POPL), 54:1–54:27 (2020). https://doi.org/10.1145/3371122, https://doi.org/10.1145/3371122

[8] Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic counter abstraction for concurrent software. In: International Conference on Computer Aided Verification. pp. 64–78. Springer (2009)

[9] Bloem, R., Hofferek, G., Könighofer, B., Könighofer, R., Außerlechner, S., Spörk, R.: Synthesis of synchronization using uninterpreted functions. In: 2014 Formal Methods in Computer-Aided Design (FMCAD). pp. 35–42. IEEE (2014)

[10] Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2015). https://doi.org/10.2200/S00658ED1V01Y201508DCT013

[11] Blondin, M., Raskin, M.A.: The complexity of reachability in affine vector addition systems with states. In: LICS. pp. 224–236. ACM (2020)

[12] Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: OSDI. pp. 335–350. USENIX Association (2006)

[13] Canepa, D., Potop-Butucaru, M.G.: Stabilizing flocking via leader election in robot networks. In: SSS. Lecture Notes in Computer Science, vol. 4838, pp. 52–66. Springer (2007)

[14] Chang, C., Tsai, J.: Distributed collaborative surveillance system based on leader election protocols. IET Wirel. Sens. Syst. **6**(6), 198–205 (2016)

[15] Clarke, E.M., Talupur, M., Touili, T., Veith, H.: Verification by network decomposition. In: CONCUR. LNCS, vol. 3170, pp. 276–291. Springer (2004)

[16] Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of model checking, vol. 10. Springer (2018)

[17] Czerwinski, W., Lasota, S., Lazic, R., Leroux, J., Mazowiecki, F.: The reachability problem for petri nets is not elementary. J. ACM **68**(1), 7:1–7:28 (2021)

[18] Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized verification of ad hoc networks. In: CONCUR. LNCS, vol. 6269, pp. 313–327. Springer (2010). https://doi.org/10.1007/978-3-642-15375-4_22

[19] Demsky, B., Rinard, M.: Automatic detection and repair of errors in data structures. In: Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03). pp. 78–95 (2003)

[20] Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: CADE. LNCS, vol. 1831, pp. 236–254. Springer (2000)

[21] Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: CHARME. LNCS, vol. 2860, pp. 247–262. Springer (2003). https://doi.org/10.1007/978-3-540-39724-3_22

[22] Emerson, E.A., Kahlon, V.: Model checking guarded protocols. In: LICS. pp. 361–370. IEEE Computer Society (2003)

[23] Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. Foundations of Computer Science **14**(4), 527–549 (2003)

[24] Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS. pp. 352–359. IEEE Computer Society (1999)

[25] Esparza, J.: Keeping a crowd safe: On the complexity of parameterized verification (invited talk). In: STACS. LIPIcs, vol. 25, pp. 1–10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2014). https://doi.org/10.4230/LIPIcs.STACS.2014.1

[26] Esparza, J., Raskin, M.A., Weil-Kennedy, C.: Parameterized analysis of immediate observation petri nets. In: Petri Nets. Lecture Notes in Computer Science, vol. 11522, pp. 365–385. Springer (2019)

[27] Finkbeiner, B., Jacobs, S.: Lazy synthesis. In: VMCAI. LNCS, vol. 7148, pp. 219–234. Springer (2012)

[28] Finkbeiner, B., Schewe, S.: Bounded synthesis. STTT **15**(5-6), 519–539 (2013). https://doi.org/10.1007/s10009-012-0228-z

[29] Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science **256**(1-2), 63–92 (2001)

[30] Forrest, S., Nguyen, T., Weimer, W., Goues, C.L.: A genetic programming approach to automated software repair. In: Genetic and Evolutionary Computation Conference (GECCO'09). pp. 947–954. ACM (2009)

[31] Frenkel, H., Grumberg, O., Pasareanu, C., Sheinvald, S.: Assume, guarantee or repair. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 211–227. Springer (2020)

[32] German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (1992)

[33] Griesmayer, A., Bloem, R., Cook, B.: Repair of Boolean programs with an application to C. In: 18th Conference on Computer Aided Verification (CAV'06). pp. 358–371 (2006), LNCS 4144

[34] Jaber, N., Jacobs, S., Wagner, C., Kulkarni, M., Samanta, R.: Parameterized verification of systems with global synchronization and guards. In: CAV (1). Lecture Notes in Computer Science, vol. 12224, pp. 299–323. Springer (2020)

[35] Jaber, N., Wagner, C., Jacobs, S., Kulkarni, M., Samanta, R.: Quicksilver: modeling and parameterized verification for distributed agreement-based systems. Proc. ACM Program. Lang. **5**(OOPSLA), 1–31 (2021)

[36] Jacobs, S., Sakr, M.: Analyzing guarded protocols: Better cutoffs, more systems, more expressivity. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 247–268. Springer (2018)

[37] Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: 17th Conference on Computer Aided Verification (CAV'05). pp. 226–238. Springer (2005), lNCS 3576

[38] McClurg, J., Hojjat, H., Černỳ, P.: Synchronization synthesis for network programs. In: International Conference on Computer Aided Verification. pp. 301–321. Springer (2017)

[39] Monperrus, M.: Automatic software repair: A bibliography. ACM Comput. Surv. **51**(1), 17:1–17:24 (2018)

[40] Pnueli, A., Xu, J., Zuck, L.D.: Liveness with (0, 1, infty)-counter abstraction. In: CAV. Lecture Notes in Computer Science, vol. 2404, pp. 107–122. Springer (2002)

[41] Suzuki, I.: Proving properties of a ring of finite state machines. Inf. Process. Lett. **28**(4), 213–214 (1988)

[42] Vechev, M., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 327–338 (2010)

Appendix A
Full Proofs of Lemmas from Section III

Lemma 1: Let $M = (S, S_0, \Delta)$ be a counter system for process templates A, B, and let $\lessapprox \subseteq S \times S$ be the binary relation defined by:

$$(q_A, \mathbf{c}) \lessapprox (q'_A, \mathbf{d}) \;\Leftrightarrow\; (q_A = q'_A \wedge \mathbf{c} \lesssim \mathbf{d}),$$

where $\lesssim$ is the component-wise ordering of vectors. Then $(M, \lessapprox)$ is a WSTS.

Proof: The partial order $\lessapprox$ is a wqo due to the fact that $\lesssim$ is a wqo. Moreover, we show that $\lessapprox$ is strongly compatible with Δ. Let $s = (q_A, \mathbf{c}), s' = (q'_A, \mathbf{c}'), r = (q_A, \mathbf{d}) \in S$ such that $s \xrightarrow{t_U} s' \in \Delta$ and $s \lessapprox r$. Since the transition t_U is enabled in s, it is also enabled in r and $\exists r' = (q'_A, \mathbf{d}') \in S$ with $r \xrightarrow{t_U} r' \in \Delta$. Then it is easy to see that $s' \lessapprox r'$: either t_U is a transition of A, then we have $\mathbf{c} = \mathbf{c}'$ and $\mathbf{d} = \mathbf{d}'$, or t_U is a transition of B with $t_U = (q_i, g, q_j)$, then $q_A = q'_A$ and $\mathbf{c}' = \mathbf{c} - \mathbf{c}_i + \mathbf{c}_j \lesssim \mathbf{d} - \mathbf{c}_i + \mathbf{c}_j = \mathbf{d}'$. ∎

Lemma 2: Let $M = (S, S_0, \Delta)$ be a counter system for guarded process templates A, B. Then $(M, \lessapprox)$ has effective *pred*-basis.

Proof: Let $R \subseteq S$ be finite. Since $pred(\uparrow R)$ will be upward-closed with respect to $\lessapprox$, it is sufficient to prove that a *basis* of $pred(\uparrow R)$ can be computed from R. Let $g = \{q_t\}$, $f = ((t = j \wedge \mathbf{c}'(j) = 1) \vee (\mathbf{c}'(t) \geq 1 \wedge \mathbf{c}'(j) = 0))$. Consider the following set of states:

SJ: can we give an intuition what f is needed for?

$$\begin{aligned}
CBasis = \{&(q_A, \mathbf{c}) \in S \mid \exists (q'_A, \mathbf{c}') \in R : \\
&[\, (q_A, g, q'_A) \in \delta_A \wedge (q_A, \mathbf{c}) \models_{q_A} g \wedge \\
&(\, (\mathbf{c} = \mathbf{c}') \vee (\mathbf{c}'(t) = 0 \wedge \mathbf{c} = \mathbf{c}' + \mathbf{u}_t)\,)\,] \\
&\vee [\, (q_i, \{q_t\}, q_j) \in \delta_B \wedge (q_A, \mathbf{c}) \models_{q_i} g \wedge q_A = q'_A \\
&\quad \wedge (\; (\mathbf{c} = \mathbf{c}' + \mathbf{u}_i - \mathbf{u}_j) \vee (\mathbf{c}'(t) = 0 \wedge \mathbf{c}'(j) \geq 1 \wedge \\
&\qquad \mathbf{c} = \mathbf{c}' + \mathbf{u}_i - \mathbf{u}_j + \mathbf{u}_t) \vee (f \wedge \mathbf{c} = \mathbf{c}' + \mathbf{u}_i) \\
&\qquad \vee (\, \mathbf{c}'(t) = 0 \wedge \mathbf{c}'(j) = 0 \wedge \mathbf{c} = \mathbf{c} + \mathbf{u}_i + \mathbf{u}_t) \;)\,]\; \}.
\end{aligned}$$

Clearly, $CBasis \subseteq pred(\uparrow R)$, and $CBasis$ is finite. We claim that also $CBasis \supseteq minBasis(pred(\uparrow R))$. For the purpose of reaching a contradiction, assume $CBasis \not\supseteq minBasis(pred(\uparrow R))$, which implies that there exists a $(q_A, \mathbf{c}) \in (minBasis(pred(\uparrow R)) \cap \neg CBasis)$. Since $(q_A, \mathbf{c}) \notin CBasis$, there exists $(q'_A, \mathbf{c}') \notin R$ with $(q_A, \mathbf{c}) \to (q'_A, \mathbf{c}')$ and since $(q_A, \mathbf{c}) \in minBasis(pred(\uparrow R))$, there is a $(q'_A, \mathbf{d}') \in R$ with $(q'_A, \mathbf{d}') \lessapprox (q'_A, \mathbf{c}')$. We differentiate between two cases:

- Case 1: Suppose $(q_A, \mathbf{c}) \xrightarrow{t_A} (q'_A, \mathbf{c}')$ with $t_A = (q_A, g, q'_A) \in \delta_A$ and $(q_A, \mathbf{c}) \models_{q_A} g$. Then $\mathbf{c} = \mathbf{c}'$, and by definition of CBasis there exists $(q_A, \mathbf{d}) \in CBasis$ with $[(q_A, \mathbf{d}) \to (q'_A, \mathbf{d}') \wedge \mathbf{d} = \mathbf{d}' \wedge \mathbf{d}'(t) \geq 1]$ or $[(q_A, \mathbf{d}) \to (q'_A, \mathbf{d}' + u_t) \wedge \mathbf{d} = \mathbf{d}' + u_t \wedge \mathbf{d}'(t) = 0]$. Furthermore, we have $\mathbf{d}' \lesssim \mathbf{c}'$, which implies $(q_A, \mathbf{d}) \lessapprox (q_A, \mathbf{c})$ with $(q'_A, \mathbf{d}') \in R$. Contradiction.
- Case 2: Suppose $(q_A, \mathbf{c}) \xrightarrow{t_B} (q'_A, \mathbf{c}')$ with $t_B = (q_i, g, q_j) \in \delta_B$ and $(q_A, \mathbf{c}) \models_{q_i} g$. Then $q_A = q'_A \wedge \mathbf{c} = \mathbf{c}' + \mathbf{u}_i - \mathbf{u}_j$. By definition of CBasis there exists $(q_A, \mathbf{d}) \in CBasis$ such that one of the following holds:
  - $(q_A, \mathbf{d}) \to (q'_A, \mathbf{d}') \wedge \mathbf{d}' = \mathbf{d} - \mathbf{u}_i + \mathbf{u}_j$
  - $\mathbf{d}'(t) = 0 \wedge \mathbf{d}'(j) \geq 1 \wedge (q_A, \mathbf{d}) \to (q'_A, \mathbf{d}' + \mathbf{u}_t) \wedge \mathbf{d}' + \mathbf{u}_t = \mathbf{d} - \mathbf{u}_i + \mathbf{u}_j$
  - $f \wedge (q_A, \mathbf{d}) \to (q'_A, \mathbf{d}' + \mathbf{u}_j) \wedge \mathbf{d}' + \mathbf{u}_j = \mathbf{d} - \mathbf{u}_i + \mathbf{u}_j$
  - $\mathbf{d}'(t) = 0 \wedge \mathbf{d}'(j) = 0 \wedge (q_A, \mathbf{d}) \to (q'_A, \mathbf{d}' + \mathbf{u}_t + \mathbf{u}_j) \wedge \mathbf{d}' + \mathbf{u}_t + \mathbf{u}_j = \mathbf{d} - \mathbf{u}_i + \mathbf{u}_j$

  Furthermore, we have $\mathbf{d}' \lesssim \mathbf{c}'$, which implies that $(q_A, \mathbf{d}) \lessapprox (q_A, \mathbf{c})$ with $(q_A, \mathbf{d}) \in minBasis(pred(\uparrow R))$. Contradiction.

∎

Theorem 1: The 01-counter system $\hat{M}$ has a deadlocked run if and only if the counter system M has a deadlocked run.

Proof idea. Suppose $x = s_1, s_2, \ldots, s_f$ is a deadlocked run of M. Note that for any $s \in S$, a transition based on local transition $t_U \in \delta_U$ is enabled if and only if a transition based on t_U is enabled in the abstract state $\alpha(s)$ of $\hat{M}$. Then it is easy to see that $\hat{x} = \alpha(s_1), \alpha(s_2), \ldots, \alpha(s_f)$ is a deadlocked run of $\hat{M}$.

Now, suppose $\hat{x} = \hat{s}_1, \hat{s}_2, \ldots, \hat{s}_f$ is a deadlocked run of $\hat{M}$. Let b be the number of transitions $(\hat{s}_k, \hat{s}_{k+1})$ based on some $t_B = (q_i, g, q_j) \in \delta_B$ with $\hat{s}_{k+1}(i) = 1$, i.e., the transitions where we keep a 1 in position i. Furthermore, let $t_1, \ldots, t_{f-1}$ be the sequence of local transitions that $\hat{x}$ is based on. Then we can construct a deadlocked run of M in the following way: We start in $s_1 = (\mathsf{init}_A, \mathbf{c}_1)$ with $\mathbf{c}_1(\mathsf{init}_B) = 2^b$ and for every t_k in the sequence do:[7]

- if $t_k \in \delta_A$, we take the same transition once,
- if $t_k = (q_i, g, q_j) \in \delta_B$ with $\hat{s}_{k+1}(i) = 0$, we take the same local transition until position i becomes empty, and
- if $t_k = (q_i, g, q_j) \in \delta_B$ with $\hat{s}_{k+1}(i) = 1$, we take the same local transition $\frac{c}{2}$ times, where c is the number of processes that are in position i before (i.e., we move half of the processes to j, and keep the other half in i).

By construction, after any of the transitions in $t_1, \ldots, t_{f-1}$, the same positions as in $\hat{x}$ will be occupied in our constructed run, thus the same transitions are enabled. Therefore, the constructed run ends in a deadlocked state. ∎

Appendix B
What can be done if a repair doesn't exist?

If Algorithm 2 returns "unrealizable", then there is no repair for the given input. To still obtain a repair, a designer can add more non-determinism and/or allow for more communication between processes, and then run the algorithm again on the new instance of the system. Moreover, unlike in monolithic systems, even if the result is "unrealizable", it may still be possible to obtain a solution that is good enough in practice. For instance, we can change our algorithm slightly as follows: When the SAT solver returns "UNSAT" after adding the constraints for an error sequence, instead of terminating we can continue computing the error sequence until a fixed point is reached. Then, we can determine the minimal number of processes m_e that is needed for the last candidate repair to

[7]Note that a similar, but more involved construction is also possible with $\mathbf{c}_1(\mathsf{init}_B) = b$.

reach an error, and conclude that this candidate is safe for any system up to size $m_e - 1$.

Appendix C
Local Witnesses are Upward-closed.

We show another property of our algorithm: even though for the reachable error sequence $\mathcal{RE}$ we do not consider the upward closure, the error paths we discover are in a sense upward-closed. This implies that an $\mathcal{RE}$ of length k represents *all possible error paths* of length k. We formalize this in the following.

Given a reachable error sequence $\mathcal{RE} = RE_k, \ldots, RE_0$, we denote by $\mathcal{UE}$ the sequence $\uparrow RE_k, \ldots, \uparrow RE_0$. Furthermore, let a *local witness* of $\mathcal{RE}$ be a sequence $\mathcal{T}_{\mathcal{RE}} = t_{U_k} \ldots t_{U_1}$ where for all $i \in \{1, \ldots, k\}$ there exists $s \in RE_i, s' \in RE_{i-1}$ with $s \xrightarrow{t_{U_i}} s'$. We define similarly the local witness $\mathcal{T}_{\mathcal{UE}}$ of $\mathcal{UE}$.

Lemma 3: Let $\mathcal{RE}$ be a reachable error sequence. Then every local witness $\mathcal{T}_{\mathcal{UE}}$ of $\mathcal{UE}$ is also a local witness of $\mathcal{RE}$.

Proof: Let $\mathcal{T}_{\mathcal{UE}} = t_{U_k} \ldots t_{U_1}$. Then there exist $s_k \in \uparrow E_k = \uparrow RE_k$, $s_{k-1} \in \uparrow RE_{k-1}, \ldots$, $s_0 \in \uparrow RE_0$ such that $s_k \xrightarrow{t_{U_k}} s_{k-1} \xrightarrow{t_{U_{k-1}}} \ldots\ldots \xrightarrow{t_{U_2}} s_1 \xrightarrow{t_{U_1}} s_0$. Let $s_0 = (q_A^0, \mathbf{d}^0)$, and let $t_{U_1} = (q_{U_{i_1}}, \{q_{t_1}\}, q_{U_{j_1}})$. Then, by construction of $\mathcal{E}$, there exists $(q_A^0, \mathbf{c}^0) \in E_0, (q_A^1, \mathbf{c}^1) \in E_1$ with $(q_A^0, \mathbf{c}^0) \lessapprox (q_A^0, \mathbf{d}^0)$ and $(q_A^1, \mathbf{c}^1) \xrightarrow{t_{U_1}} (q_A^0, \mathbf{c}^0)$ or $(q_A^1, \mathbf{c}^1) \xrightarrow{t_{U_1}} (q_A^0, \mathbf{c}^0 + \mathbf{u}_{t_1})$, hence t_{U_1} is enabled in $(q_A^1, \mathbf{c}^1)$. Using the same argument we can compute $(q_A^2, \mathbf{c}^2) \in E_2, (q_A^3, \mathbf{c}^3) \in E_3, \ldots$ until we reach the state $(q_A^k, \mathbf{c}^k) \in E_k$ where t_{U_k} is enabled. Therefore we have the sequence $s_k^R \xrightarrow{t_{U_k}} s_{k-1}^R \xrightarrow{t_{U_{k-1}}} \ldots\ldots \xrightarrow{t_{U_2}} s_1^R \xrightarrow{t_{U_1}} s_0^R$ with $s_k^R = (q_A^k, \mathbf{c}^k) \in RE_k = E_k$ and for all $i < k$ we have $s_i^R \in RE_i$, as they are reachable from $s_k^R \in RE_k$ and $(q_A^i, \mathbf{c}^i) \lessapprox s_i^R$ which guarantees that t_{U_i} is enabled in s_i^R. ■

Appendix D
Beyond Disjunctive Systems

Algorithm 2 is not restricted to disjunctive systems. In principle, it can be used for any system that can be modeled as a WSTS with effective *pred*-basis, as long as we can construct the transition relation constraint ($TRConstr$) for the corresponding system. In this section we show two other classes of systems that can be modeled in this framework: pairwise rendezvous (PR) and broadcast (BC) systems. We introduce transition relation constraints for these systems, as well as a procedure BuildSyncConstr that must be used instead of BuildConstr when a transition relation comprises synchronous actions.

Since these two classes of systems require processes to synchronize on certain actions, we first introduce a different notion of process templates.

Processes. A *synchronizing process template* is a transition system
$U = (Q_U, \mathsf{init}_U, \Sigma, \delta_U)$ with

- $Q_U \subseteq Q$ is a finite set of states including the initial state init_U,
- $\Sigma = \Sigma_{sync} \times \{?, !, ??, !!\} \cup \{\tau\}$ where Σ_{sync} is a set of synchronizing actions, and τ is an internal action,
- $\delta_U : Q_U \times \Sigma \times Q_U$ is a transition relation.

Synchronizing actions like $(a, !)$ or $(b, ?)$ are shortened to $a!$ and $b?$. Intuitively actions of the form $a!$ and $a?$ are PR send and receive actions, respectively, and $a!!, a??$ are BC send and receive actions, respectively.

All processes mentioned in the following are based on a synchronizing process template. We will define global systems based on either PR or BC synchronization in the following subsections.

A. Pairwise Rendezvous Systems

A PR system [32] consists of a finite number of processes running concurrently. As before, we consider systems of the form $A \| B^n$. The semantics is interleaving, except for actions where two processes synchronize. That is, at every time step, either exactly one process makes an internal transition τ, or exactly two processes synchronize on a single action $a \in \Sigma_{sync}$. For a synchronizing action $a \in \Sigma_{sync}$, the initiator process locally executes the $a!$ action and the recipient process executes the $a?$ action.

Similar to what we defined for disjunctive systems, the configuration space of all systems $A \| B^n$, for fixed A, B but arbitrary $n \in \mathbb{N}$, is the counter system $M^{PR} = (S, S_0, \Delta)$, where:

- $S \subseteq Q_A \times \mathbb{N}_0^{|B|}$ is the set of states,
- $S_0 = \{(init_A, \mathbf{c}) \mid \forall q_B \in Q_B : \mathbf{c}(q_B) = 0 \text{ if } q_B \neq init_B)\}$ is the set of initial states,
- Δ is the set of transitions $((q_A, \mathbf{c}), (q'_A, \mathbf{c}'))$ such that one of the following holds:
  1) $(q_A, \tau, q'_A) \in \delta_A \wedge \mathbf{c} = \mathbf{c}'$ (internal transition A)
  2) $\exists q_i, q_j : (q_i, \tau, q_j) \in \delta_B \wedge c(i) \geq 1 \wedge \mathbf{c}' = \mathbf{c} - \mathbf{u}_i + \mathbf{u}_j \wedge q_A = q'_A$ (internal transition B)
  3) $a \in \Sigma_{sync} \wedge (q_A, a!, q'_A) \in \delta_A \wedge \exists q_i, q_j : (q_i, a?, q_j) \in \delta_B \wedge c(i) \geq 1, \mathbf{c}' = \mathbf{c} - \mathbf{u}_i + \mathbf{u}_j$ (synchronizing transition A, B)
  4) $a \in \Sigma_{sync} \wedge (q_A, a?, q'_A) \in \delta_A \wedge \exists q_i, q_j : (q_i, a!, q_j) \in \delta_B \wedge c(i) \geq 1, \mathbf{c}' = \mathbf{c} - \mathbf{u}_i + \mathbf{u}_j$ (synchronizing transition B, A)
  5) $\exists q_i, q_j : (q_i, a!, q_j) \in \delta_B \wedge \exists q_l, q_m : (q_l, a?, q_m) \in \delta_B \wedge c(i) \geq 1 \wedge c(l) \geq 1 \wedge \mathbf{c}' = \mathbf{c} - \mathbf{u}_i + \mathbf{u}_j - \mathbf{u}_l + \mathbf{u}_m$ (synchronizing transition B, B)

The following result can be considered folklore, a proof can be found in the survey by Bloem et al. [10].

Lemma 4: Let $M^{PR} = (S, S_0, \Delta)$ be a counter system for process templates A, B with PR synchronization. Then $(M^{PR}, \lessapprox)$ is a WSTS with effective *pred*-basis.

Initial Constraints. The constraint $TRConstr_{PR}$, ensuring that not all local transitions from any given local state are removed, is constructed in a similar way as $TRConstr_{Disj}$.

Furthermore, the user may want to ensure that in the returned repair, either (a) for all $a \in \Sigma_{sync}$, $t_{a!}$ is deleted if and only if all $t_{a?}$ are deleted, or (b) that synchronized actions are deterministic, i.e., for every state q_U and every synchronized action a, there is exactly one transition on $a?$ from q_U. We give *user constraints* that ensure such behavior.

Denote by $t_{a?}, t_{a!}$ synchronous local transitions based on an action a. Then, the constraint ensuring property (a) is

$$\bigwedge_{a \in \Sigma_{sync}} [(t_{a!} \wedge (\bigvee_{t_{a?} \in \delta} t_{a?})) \vee (\neg t_{a!} \wedge (\bigwedge_{t_{a?} \in \delta} \neg t_{a?}))]$$

To encode property (b), for $U \in \{A, B\}$ and $a \in \Sigma_{sync}$, let $\{t_{q_U}^{a_?^1}, \ldots, t_{q_U}^{a_?^m}\}$ be the set of all $a?$ transitions from state $q_U \in Q_U$. Additionally, let $one(t_{q_U}^{a_?}) = \bigvee_{j \in \{1,\ldots,m\}} [t_{q_U}^{a_?^j} \bigwedge_{l \neq j} \neg t_{q_U}^{a_?^l}]$. Then, (b) is ensured by

$$\bigwedge_{a \in \Sigma_{sync}} \bigwedge_{q_U \in Q} one(t_{q_U}^{a_?})$$

SJ: revise this

Deadlock Detection for PR Systems. German and Sistla [32] have shown that deadlock detection in PR systems can be reduced to reachability in VASS, and vice versa. Thus, at least a rudimentary version of repair including deadlock detection is possible, where the deadlock detection only excludes the current candidate repair, but may not be able to provide constraints on candidates that may be considered in the future. Moreover, the reachability problem in VASS has recently been shown to be TOWER-hard, so a practical solution is unlikely to be based on an exact approach.

B. Broadcast Systems

In broadcast systems, the semantics is interleaving, except for actions where all processes synchronize, with one process "broadcasting" a message to all other processes. Via such a broadcast synchronization, a special process can be selected while the system is running, so we can restrict our model to systems that only contain an arbitrary number of user processes with identical template B. Formally, at every time step either exactly one process makes an internal transition τ, or all processes synchronize on a single action $a \in \Sigma_{sync}$. For a synchronized action $a \in \Sigma_{sync}$, we say that the initiator process executes the $a!!$ action and all recipient processes execute the $a??$ action. For every action $a \in \Sigma_{sync}$ and every state $q_B \in Q_B$, there exists a state $q'_B \in Q_B$ such that $(q_B, a??, q'_B) \in \delta_B$. Like Esparza et al. [24], we assume w.l.o.g. that the transitions of recipients are deterministic for any given action, which implies that the effect of a broadcast message on the recipients can be modeled by multiplication of a *broadcast matrix*. We denote by M_a the broadcast matrix for action a.

Then, the configuration space of all broadcast systems B^n, for fixed B but arbitrary $n \in \mathbb{N}$, is the counter system $M^{BC} = (S, S_0, \Delta)$ where:

- $S \subseteq \mathbb{N}_0^{|B|}$ is the set of states,
- $S_0 = \{\mathbf{c} \mid \forall q_B \in Q_B : \mathbf{c}(q_B) = 0 \text{ iff } q_B \neq init_B)\}$ is the set of initial states,
- Δ is the set of transitions $(\mathbf{c}, \mathbf{c}')$ such that one of the following holds:
  1) $\exists q_i, q_j \in Q_B : (q_i, \tau, q_j) \in \delta_B \wedge \mathbf{c}' = \mathbf{c} - \mathbf{u}_i + \mathbf{u}_j$ (internal transition)
  2) $\exists a \in \Sigma_{sync} : \mathbf{c}' = M_a \cdot (\mathbf{c} - \mathbf{u}_i) + \mathbf{u}_j$ (broadcast)

Lemma 5: [24] Let $M^{BC} = (S, S_0, \Delta)$ be a counter system for process template B with BC synchronization. Then $(M^{BC}, \lesssim)$ is a WSTS with effective *pred*-basis.

Initial Constraints. $TRConstr_{BC}$ is defined similarly to $TRConstr_{PR}$, except that we do not have process A and can omit transitions of A. We denote by $t_{a??}, t_{a!!}$ synchronous transitions based on an action a. To ensure that in any repair and for all $a \in \Sigma_{sync}$, $t_{a!!}$ is deleted if and only if all $t_{a??}$ are deleted, the designer can use the following constraint:

SJ: To be precise, it should also preserve the assumption that in every state we have a transition on a??, for every a.

$$\bigwedge_{a \in \Sigma_{sync}} [(t_{a!!} \wedge (\bigvee_{t_{a??} \in \delta_B} t_{a??})) \vee (\neg t_{a!!} \wedge (\bigwedge_{t_{a??} \in \delta_B} \neg t_{a??}))]$$

Deadlock Detection for BC Systems.

Theorem 5: Deadlock detection in broadcast protocols is undecidable.

The main ingredient of the proof is the following lemma:

Lemma 6: There is a polynomial-time reduction from the reachability problem of affine VASS with broadcast matrices to the deadlock detection problem in broadcast protocols.

Proof: We modify the construction from the proofs of Theorems 3.17 and 3.18 from German and Sistla [32], using affine VASS instead of VASS and broadcast protocols instead of pairwise rendezvous systems.

Starting from an arbitrary affine VASS G that only uses broadcast matrices and where we want to check if configuration $(q_2, \mathbf{c}_2)$ is reachable from $(q_1, \mathbf{c}_1)$, we first transform it to an affine VASS G^* with the following properties

- each transition only changes the vector $\mathbf{c}$ in one of the following ways: (i) it adds to or subtracts from $\mathbf{c}$ a unit vector, or (ii) it multiplies $\mathbf{c}$ with a broadcast matrix M (this allows us to simulate every transition with a single transition in the broadcast system), and
- some configuration $(q'_2, 0)$ is reachable from some configuration $(q'_1, 0)$ in G^* if and only if $(q_2, \mathbf{c}_2)$ is reachable from $(q_1, \mathbf{c}_1)$ in G.

SJ: do we have notation for the 0 vector?

The transformation is straightforward by splitting more complex transitions and adding auxiliary states. Now, based on G^* we define process templates A and B such that $A \| B^n$ can reach a deadlock iff $(q'_2, 0)$ is reachable from $(q'_1, 0)$ in G^*.

The states of A are the discrete states of G^*, plus additional states q', q''. If the state vector of G^* is m-dimensional, then B has states $q_1, \ldots, q_m$, plus states init, v. Then, corresponding to every transition in G^* that changes the state from q to q' and either adds or subtracts unit vector $\mathbf{u}_i$, we have a rendezvous sending transition from q to q' in A, and a corresponding receiving transition in B from init to q_i (if $\mathbf{u}_i$ was added), or from q_i to init (if $\mathbf{u}_i$ was subtracted). For every transition that changes the state from q to q' and multiplies $\mathbf{c}$ with a matrix M, A has a broadcast sending transition from q to q', and receiving transitions between the states $q_1, \ldots, q_m$ that correspond to the effect of M.

Algorithm 3 Synchronous Constraint Computation

```
1: procedure BSC(State $s$, $\mathcal{RE}$)
2:     if $\mathcal{RE}[1:]$ is empty then
3:         return $\bigwedge_{t_U \in \Delta^{local}(s,\mathcal{RE}[0])} \neg t_U$
                  $\bigwedge_{a \in \Sigma_{sync} \wedge \Delta(s,a) \in \mathcal{RE}[0]} T(s,a)$
4:     else
5:         return $\bigwedge_{t_U \in \Delta^{local}(s,\mathcal{RE}[0])} (\neg t_U \vee$
                  $BSC(\Delta(s,t_U), \mathcal{RE}[1:]))$
              $\bigwedge_{a \in \Sigma_{sync} \wedge \Delta(s,a) \in \mathcal{RE}[0]} [T(s,a) \vee$
                  $BSC(\Delta(s,t_a), \mathcal{RE}[1:])]\}$
```

The additional states q', q'' of A are used to connect reachability of $(q_2', 0)$ to a deadlock in $A \| B^n$ in the following way: (i) there are self-loops on all states of A except on q', i.e., the system can only deadlock if A is in q', (ii) there is a broadcast sending transition from q_2' to q' in A, which sends all B-processes that are in $q_1, \dots, q_m$ to special state v, and (iii) from v there is a broadcast sending transition to init in B, and a corresponding receiving transition from q' to q'' in A. Thus, $A \| B^n$ can only deadlock in a configuration where A is in q' and there are no B-processes in v, which is only reachable through a transition from a configuration where A is in q_2 and no B-processes are in $q_1, \dots, q_m$. Letting q_1 be the initial state of A and init the initial state of B, such a configuration is reachable in $A \| B^n$ if and only if $(q_2', 0)$ is reachable from $(q_1', 0)$ in G^*.

■

C. *Synchronous Systems Constraints*

The procedure BUILDCONSTR in Algorithm 2 does not take into consideration synchronous actions. Hence, we need a new procedure that offers special treatment for synchronization. To simplify presentation we assume w.l.o.g. that each $a+$, with $+ \in \{!, !!\}$, appears on exactly one local transition. We denote by $\Delta_{sync}(s,a)$ the state obtained by executing action a in state s. Additionally, let $\Delta^{local}_{sync}(s,a) = \{(q_U, a_*, q_U') \in \delta \mid * \in \{?, !, ??, !!\},$ and a is enabled in $s\}$, and let $T(s,a) = \bigvee_{t_a \in \Delta^{local}_{sync}(s,a)} \neg t_a$. In a Broadcast system we say that an action a is enabled in a global state $\mathbf{c}$ if $\exists i, j < |B|$ s.t. $\mathbf{c}(i) > 0$ and $(q_{B_i}, a!!, q_{B_j}) \in \delta_B$. In a Pairwise rendezvous system we say that an action a is enabled in a global state $(\mathbf{c})$ if $\exists i, j, k, l < |B|$ s.t. $\mathbf{c}(i) > 0, \mathbf{c}(j) > 0)$ and $(q_{B_i}, a!, q_{B_k}), (q_{B_j}, a?, q_{B_l}), \in \delta_B$.

Given a synchronous system $M^X = (S, S_0, \Sigma, \Delta)$ with $X \in \{BR, PR\}$, a state s, and a reachable error sequence $\mathcal{RE}$, Algorithm 3 computes a propositional formula over the set of local transitions that encodes all possible ways for a state s to avoid reaching an error.

APPENDIX E
EXAMPLE: READER-WRITER

Consider the parameterized pairwise system that consists of one scheduler (Figure 8) and a parameterized number of instances of the reader-writer process template (Figure 9). The scheduler process template has all possible receive actions from every state. In such system, the scheduler can not guarantee that, at any moment, there is at most one process in the *writing* state q_1 (Figure 9). Let $t_{U_1} = [q_0, (write!), q_1], t_{U_2} = [q_{A,0}, (write?), q_{A,1}], t_{U_3} = [q_{A,1}, (write?), q_{A,0}]$,
$t_{U_4} = [q_0, (read!), q_2], t_{U_5} = [q_{A,0}, (read?), q_{A,1}], t_{U_6} = [q_{A,1}, (read?), q_{A,0}], t_{U_7} = [q_1, (done_w!), q_0], t_{U_8} = [q_{A,1}, (done_w?), q_{A,0}], t_{U_9} = [q_{A,0}, (done_w?), q_{A,1}], t_{U_{10}} = [q_2, (done_r!), q_0], t_{U_{11}} = [q_{A,1}, (done_r?), q_{A,0}], t_{U_{12}} = [q_{A,0}, (done_r?), q_{A,1}]$.
Let $ERR = \uparrow\{(q_{A,0}, (0,2,0))(q_{A,1}, (0,2,0))\}$.
Let $UserConstr_{PR} = (t_{U_1} \wedge (t_{U_2} \vee t_{U_3})) \wedge (t_{U_4} \wedge (t_{U_5} \vee t_{U_6})) \wedge (t_{U_7} \wedge (t_{U_8} \vee t_{U_9})) \wedge (t_{U_{10}} \wedge (t_{U_{11}} \vee t_{U_{12}}))$.
Then running our repair algorithm will produce the following results:
First call to model checker returns:
$RE_0 = \{(q_{A,0}, (0,2,0))\}, RE_1 = \{(q_{A,1}, (1,1,0))\}, RE_2 = \{(q_{A,0}, (2,0,0))\}$.
Constraints for SAT: $accConstr_1 = TRConstr_{PR} \wedge UserConstr_{PR} \wedge (\neg t_{U_1} \vee \neg t_{U_2} \vee \neg t_{U_3})$.
SAT solvers solution 1:
$\neg t_{U_2} \wedge \neg t_{U_6} \wedge \neg t_{U_9} \wedge \neg t_{U_{12}}$.
Second call to model checker returns:
$RE_0 = \{(q_{A,0}, (0,2,0))\}, RE_1 = \{(q_{A,1}, (1,1,0))\}, RE_2 = \{(q_{A,0}, (2,1,0))\}, RE_3 = \{(q_{A,1}, (3,0,0))\}, RE_4 = \{(q_{A,0}, (4,0,0))\}$. Constraints for SAT:
$accConstr_2 = accConstr_1 \wedge (\neg t_{U_1} \vee \neg t_{U_3} \vee \neg t_{U_4} \vee \neg t_{U_5})$.
SAT solvers solution 2:
$\neg t_{U_3} \wedge \neg t_{U_5} \wedge \neg t_{U_9} \wedge \neg t_{U_{12}}$.
Third call to model checker returns:
$RE_0 = \{(q_{A,0}, (0,2,0))\}, RE_1 = \{(q_{A,1}, (1,1,0))\}, RE_2 = \{(q_{A,0}, (2,1,0))\}, RE_3 = \{(q_{A,1}, (3,0,0))\}, RE_4 = \{(q_{A,0}, (3,0,0))\}$. Constraints for SAT:
$accConstr_3 = accConstr_2 \wedge (\neg t_{U_1} \vee \neg t_{U_2} \vee \neg t_{U_4} \vee \neg t_{U_6})$.
SAT solvers solution 3:
$\neg t_{U_3} \wedge \neg t_{U_6} \wedge \neg t_{U_9} \wedge \neg t_{U_{12}}$.
The fourth call of the model checker returns true and we obtain the correct scheduler in Figure 10.

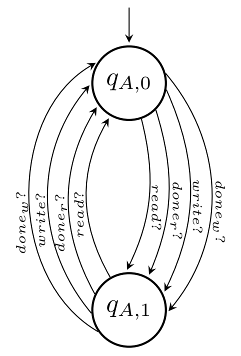

Fig. 8: Scheduler

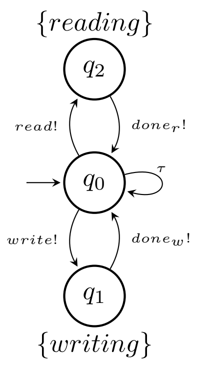

Fig. 9: Reader-Writer

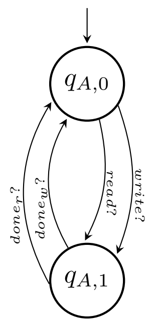

Fig. 10: Safe Scheduler

Appendix F
Example: MESI Protocol

Consider the cache coherence protocol MESI in Figure 11, where:

- M stands for *modified* and indicates that the cache has been changed.
- E stands for *exclusive* and indicates that no other process seizes this cache line.
- S stands for *shared* and indicates that more than one process hold this cache line.
- I stands for *invalid* and indicates that the cache's content is not guaranteed to be valid as it might have been changed by some process.

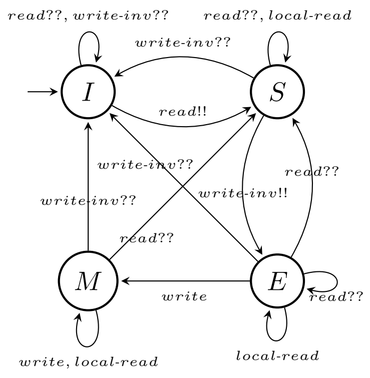

Fig. 11: MESI protocol

Initially all processes are in I and let a state vector be as follows: (M, E, S, I). An important property for MESI protocol is that a cache line should not be modified by one process (in state M) and in shared state for another process (in state S). In such case the set of error states is: $\uparrow(1,0,1,0)$. We can run Algorithm 2 on M, $\uparrow(1,0,1,0)$, $TRConstr_{BC} \wedge \bigwedge_{a\in\Sigma_{sync}} \bigwedge_{q_B\in Q_B} one(t^{a??}_{q_B})$. The model checker will return the following error sequence (nonessential states are omitted):
$E_0 = \{(1,0,1,0)\}, E_1 = \{(0,1,1,0)\}, E_2 = \{(0,1,0,1)\}, E_3 = \{(0,0,1,1)\}, E_4 = \{(0,0,0,2)\}$. Running the procedure BuildSyncConstr (Algorithm 3) in Line 9 will return the following Boolean formula $newConstr = \neg(I, read!!, S) \vee \neg(I, read??, I) \vee \neg(S, write\text{-}inv!!, E) \vee \neg(I, write\text{-}inv??, I) \vee \neg(E, read??, E) \vee \neg(I, read!!, S) \vee \neg(E, write, S)$.
Running the SAT solve in Line 12 on

$$newConstr \wedge TRConstr'_{BC} \wedge \bigwedge_{a\in\Sigma_{sync}} \bigwedge_{q_B\in Q_B} one(t^{a??}_{q_B}) \bigwedge_{t_U\in\{\delta^\tau_B\}} t_U$$

will return the only solution $\neg(E, read??, E)$ which clearly fixes the system.