# Distributed Priority Synthesis

Chih-Hong Cheng[1] Rongjie Yan[2] Saddek Bensalem[3] Harald Ruess[1]

[1] Fortiss - An-Institut der TU München, Munich, Germany

[2] State Key Laboratory of Computer Science, Institute of Software, Beijing, China

[3] Verimag Laboratory, Grenoble, France

cheng@fortiss.org yrj@ios.ac.cn saddek.bensalem@imag.fr ruess@fortiss.org

Given a set of interacting components with non-deterministic variable update and given safety requirements, the goal of *priority synthesis* is to restrict, by means of priorities, the set of possible interactions in such a way as to guarantee the given safety conditions for all possible runs. In *distributed priority synthesis* we are interested in obtaining local sets of priorities, which are deployed in terms of local component controllers sharing intended next moves between components in local neighborhoods only. These possible communication paths between local controllers are specified by means of a *communication architecture*. We formally define the problem of distributed priority synthesis in terms of a multi-player safety game between players for (angelically) selecting the next transition of the components and an environment for (demonically) updating uncontrollable variables. We analyze the complexity of the problem, and propose several optimizations including a solution-space exploration based on a diagnosis method using a nested extension of the usual attractor computation in games together with a reduction to corresponding SAT problems. When diagnosis fails, the method proposes potential candidates to guide the exploration. These optimized algorithms for solving distributed priority synthesis problems have been integrated into the VissBIP framework. An experimental validation of this implementation is performed using a range of case studies including scheduling in multicore processors and modular robotics.

## 1 Introduction

Distributed computing assemblies are usually built from interacting components with each component realizing a specific, well defined capability or service. Such a constituent component can be understood as a platform-independent computational entity that is described by means of its interface, which is published and advertised in the intended hosting habitat.

In effect, computing assemblies constrain the behavior of their constituent components to realize goal-directed behavior, and such a goal-directed orchestration of interacting components may be regarded as synthesizing winning strategies in a multi-player game, with each constituent component and the environment a player. The game is won by the component players if the intended goals are achieved, otherwise the environment wins. The orchestration itself may be centralized in one or several specialized controller components or the control may be distributed among the constituent components. Unfortunately, distributed controller synthesis is known to be undecidable [20] in theory even for reachability or simple safety conditions [14]. A number of decidable subproblems have been proposed either by restricting the communication structures between components, such as pipelined, or by restricting the set of properties under consideration [18, 17, 19, 12].

In this paper we describe a solution to the distributed synthesis problem for automatically synthesizing local controllers which are distributed among the constituent components. More precisely, given a set of interacting components with non-deterministic variable update and given a safety requirement on the overall system, the goal of distributed priority synthesis is to restrict, by means of priorities on interactions, the set of possible interactions in such a way as to guarantee the given safety conditions. The structure of these priorities is restricted in order to deploy the corresponding controllers in a distributed way, and communication between these local controllers is restricted based on a given communication architecture.

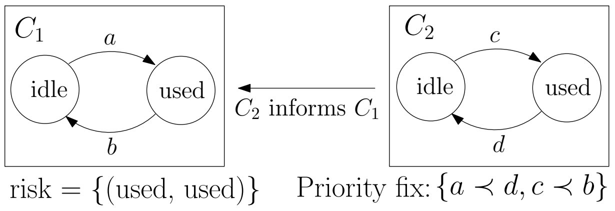

$C_1$
$a$
idle
used
$b$
$C_2$ informs $C_1$
$C_2$
$c$
idle
used
$d$
risk $= \{(\text{used}, \text{used})\}$
Priority fix: $\{a \prec d, c \prec b\}$

Figure 1: A sample example.

For example, Figure 1 depicts two interacting components $C_1$ and $C_2$ with states *idle* and *used* and transitions $a$ through $d$ with no further synchronization between the components. The goal is to never simultaneously be in the risk state *used*. This goal is achieved by placing certain priorities on possible interactions. The priority $a \prec d$, for example, inhibits transitions of $C_1$ from state *idle* to *used*, whenever $C_2$ is ready to leave the state *used*. This constraint might be used as the basis of a local controller for $C_1$ as it is informed by $C_2$ about its intended move using the given communication channel. Since many well-known scheduling strategies can be encoded by means of priorities on interactions [13], priority synthesis is closely related to solving scheduling problems. In this way, the result of distributed priority synthesis may also be viewed as a distributed scheduler.

The rest of the paper is structured as follows. Section 2 contains background information on a simplified variant of the Behavior-Interaction-Priority (BIP) modeling framework [1]. The corresponding priority synthesis problem corresponds to synthesizing a state-less winning strategy in a two-player safety game, where the control player (angelically) selects the next transition of the components and the environment player (demonically) updates uncontrollable variables. In Section 3 we introduce the notion of deployable communication architectures and formally state the distributed priority synthesis problem. Whereas the general distributed controller synthesis problem is undecidable [20] we show that distributed priority synthesis is NP-complete. Overall, distributed priority synthesis is decidable over all communication architectures, as the methodology essentially searches for a strategy of a certain "shape", where the shape is defined in terms of priorities. Section 4 contains a solution to the distributed synthesis problem, which is guaranteed to be deployable on a given communication architecture. This algorithm is a generalization of the solution to the priority synthesis problem in [10, 9]. It integrates essential optimizations based on symbolic game encodings including visibility constraints, followed by a nested attractor computation, and lastly, solving a corresponding (Boolean) satisfiability problem by extracting fix candidates while considering architectural constraints. Section 5 describes some details and optimization of our implementation, which is validated in Section 6 against a set of selected case studies including scheduling in multicore processors and modular robotics. Section 7 contains related work and we conclude in Section 8. Due to space limits, we leave proofs of propositions to our technical report [8].

## 2 Background

Our notion of *interacting components* is heavily influenced by the Behavior-Interaction-Priority (BIP) framework [1] which consists of a set of automata (extended with data) that synchronize on joint labels; it is designed to model systems with combinations of synchronous and asynchronous composition. For simplicity, we omit many syntactic features of BIP such as hierarchies of interactions and we restrict ourselves to Boolean data types only. Furthermore, uncontrollability is restricted to non-deterministic update of variables, and data transfer among joint interaction among components is also omitted.

Let $\Sigma$ be a nonempty alphabet of *interactions*. A *component* $C_i$ of the form $(L_i, V_i, \Sigma_i, T_i, l_i^0, e_i^0)$ is a *transition system* extended with data, where $L_i$ is a nonempty, finite set of *control locations*, $\Sigma_i \subseteq \Sigma$ is a nonempty subset of interaction labels used in $C_i$, and $V_i$ is a finite set of *(local) variables* of Boolean domain $\mathbb{B} = \{\texttt{True}, \texttt{False}\}$. The set $\mathcal{E}(V_i)$ consists of all evaluations $e : V_i \to \mathbb{B}$ over the variables $V_i$, and $\mathcal{B}(V_i)$ denotes the set of propositional formulas over variables in $V_i$; variable evaluations are extended to propositional formulas in the obvious way. $T_i$ is the set of *transitions* of the form $(l, g, \sigma, f, l')$, where $l, l' \in L_i$ respectively are the source and target locations, the guard $g \in \mathcal{B}(V_i)$ is a Boolean formula over the variables $V_i$, $\sigma \in \Sigma_i$ is an interaction label (specifying the event triggering the transition), and $f : V_i \to (2^{\mathbb{B}} \setminus \emptyset)$ is the *update relation* mapping every variable to a set of allowed Boolean values. Finally, $l_i^0 \in L_i$ is the *initial location* and $e_i^0 \in \mathcal{E}(V_i)$ is the initial evaluation of the variables.

A system $\mathcal{S}$ of *interacting components* is of the form $(C, \Sigma, \mathcal{P})$, where $C = \{C_i\}_{1 \le i \le m}$ is a set of components, the set of *priorities* $\mathcal{P} \subseteq 2^{\Sigma \times \Sigma}$ is irreflexive and transitive [13]. The notation $\sigma_1 \prec \sigma_2$ is usually used instead of $(\sigma_1, \sigma_2) \in \mathcal{P}$, and we say that $\sigma_2$ has higher priority than $\sigma_1$. A *configuration (or state)* $c$ of a system $\mathcal{S}$ is of the form $(l_1, e_1, \ldots, l_m, e_m)$ with $l_i \in L_i$ and $e_i \in \mathcal{E}(V_i)$ for all $i \in \{1, \ldots, m\}$. The *initial configuration* $c_0$ of $\mathcal{S}$ is of the form $(l_1^0, e_1^0, \ldots, l_m^0, e_m^0)$. An interaction $\sigma \in \Sigma$ is *(globally) enabled* in a configuration $c$ if, first, joint participation holds for $\sigma$, that is, for all $i \in \{1, \ldots, m\}$, if $\sigma \in \Sigma_i$, then there exists a transition $(l_i, g_i, \sigma, f_i, l_i') \in T_i$ with $e_i(g_i) = \texttt{True}$, and, second, there is no other interaction of higher priority for which joint participation holds. $\Sigma_c$ denotes the set of (globally) enabled interactions in a configuration $c$. For $\sigma \in \Sigma_c$, a configuration $c'$ of the form $(l_1', e_1', \ldots, l_m', e_m')$ is a *$\sigma$-successor* of $c$, denoted by $c \xrightarrow{\sigma} c'$, if, for all $i$ in $\{1, \ldots, m\}$: if $\sigma \notin \Sigma_i$, then $l_i' = l_i$ and $e_i' = e_i$; if $\sigma \in \Sigma_i$ and (for some) transition of the form $(l_i, g_i, \sigma, f_i, l_i') \in T_i$ with $e_i(g_i) = \texttt{True}$, $e_i' = e_i[v_i/d_i]$ with $d_i \in f(v_i)$.

A *run* is of the form $c_0, \ldots, c_k$ with $c_0$ the initial configuration and $c_j \xrightarrow{\sigma_{j+1}} c_{j+1}$ for all $j : 0 \le j < k$. In this case, $c_k$ is reachable, and $\mathcal{R}_{\mathcal{S}}$ denotes the set of all reachable configurations from $c_0$. Notice that such a sequence of configurations can be viewed as an execution of a two-player game played alternatively between the control Ctrl and the environment Env. In every position, player Ctrl selects one of the enabled interactions and Env non-deterministically chooses new values for the variables before moving to the next position. The game is won by Env if Ctrl is unable to select an enabled interaction, i.e., the system is deadlocked, or if Env is able to drive the run into a bad configuration from some given set $C_{risk} \subseteq C_{\mathcal{S}}$. More formally, the system is *deadlocked* in configuration $c$ if there is no $c' \in \mathcal{R}_{\mathcal{S}}$ and no $\sigma \in \Sigma_c$ such that $c \xrightarrow{\sigma} c'$, and the set of deadlocked states is denoted by $C_{dead}$. A configuration $c$ is *safe* if $c \notin C_{dead} \cup C_{risk}$, and a system is safe if no reachable configuration is unsafe.

**Definition 1 (Priority Synthesis)** *Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ together with a set $C_{risk} \subseteq C_{\mathcal{S}}$ of risk configurations, $\mathcal{P}_+ \subseteq \Sigma \times \Sigma$ is a solution to the* priority synthesis problem *if the extended system $(C, \Sigma, \mathcal{P} \cup \mathcal{P}_+)$ is safe, and the defined relation of $\mathcal{P} \cup \mathcal{P}_+$ is also irreflexive and transitive.*

For the product graph induced by system $\mathcal{S}$, let $Q$ be the set of vertices and $\delta$ be the set of transitions. In a single player game, where Env is restricted to deterministic updates, finding a solution to the priority synthesis problem is NP-complete in the size of $(|Q| + |\delta| + |\Sigma|)$ [11].

**(Example in Fig. 1)** The system $\mathcal{S}$ has two components $C_1, C_2$ (each component does not use any variable), uses interactions $\Sigma = \{a, b, c, d\}$, and has no predefined priorities. The initial configuration is $(idle, idle)$. Define the set of risk states to be $\{(used, used)\}$, then priority synthesis introduces $\{a \prec d, c \prec b\}$ as the set of priorities to avoid deadlock and risk states. Such a set ensures that whenever one component uses the resource, the other component shall wait until the resource is released. E.g., when $C_2$ is at *used* and $C_1$ is at *idle*, priority $a \prec d$ can force $a$ to be disabled.

Examples of using non-controllable environment updates can be found in our extended report [8].

## 3 Distributed Execution

We introduce the notion of (deployable) communication architecture for defining distributed execution for a system $\mathcal{S}$ of interacting components. Intuitively, a communication architecture specifies which components exchange information about their next intended move.

**Definition 2** *A communication architecture $Com$ for a system $\mathcal{S}$ of interacting components is a set of ordered pairs of components of the form $(C_i, C_j)$ for $C_i, C_j \in C$. In this case we say that $C_i$* informs *$C_j$ and we use the notation $C_i \rightsquigarrow C_j$. Such a communication architecture $Com$ is* deployable *if the following conditions hold for all $\sigma, \tau \in \Sigma$ and $i, j \in \{1, \dots, m\}$:*

- (Self-transmission) $\forall i \in \{1, \dots, m\}$, $C_i \rightsquigarrow C_i \in Com$.
- (Group transmission) *If $\sigma \in \Sigma_i \cap \Sigma_j$ then $C_j \rightsquigarrow C_i$, $C_i \rightsquigarrow C_j \in Com$.*
- (Existing priority transmission) *If $\sigma \prec \tau \in \mathcal{P}$, $\sigma \in \Sigma_j$, and $\tau \in \Sigma_i$ then $C_i \rightsquigarrow C_j \in Com$.*

Therefore, components that possibly participate in a joint interaction exchange information about next intended moves (group transmission), and components with a high priority interaction $\tau$ need to inform all components with an interaction of lower priority than $\tau$ (existing priority transmission)[1]. We make the following assumption.

**Assumption 1 (Compatibility Assumption)** *A system under synthesis has a deployable communication architecture.*

**(Example in Fig. 1)** The communication architecture $Com$ in Fig. 1 is $\{C_1 \rightsquigarrow C_1, C_2 \rightsquigarrow C_2, C_2 \rightsquigarrow C_1\}$. The original system $\mathcal{S}$ under $Com$ is deployable, but the modified system which includes the synthesized priorities $\{a \prec d, c \prec b\}$ is not, as it requires $C_1 \rightsquigarrow C_2$ to support the use of priority $c \prec b$ (when $C_2$ wants to execute $c$, it needs to know whether $C_1$ wants to execute $b$).

Next we define distributed notions of enabled interactions and behaviors, where all the necessary information is communicated along the defined communication architecture.

**Definition 3** *Given a communication architecture $Com$ for a system $\mathcal{S}$, an interaction $\sigma$ is* visible *by $C_j$ if $C_i \rightsquigarrow C_j$ for all $i \in \{1, \dots, m\}$ such that $\sigma \in \Sigma_i$. Then for configuration $c = (l_1, e_1, \dots, l_m, e_m)$, an interaction $\sigma \in \Sigma$ is* distributively-enabled (at *$c$*) *if:*

- *(Joint participation: distributed version) for all $i$ with $\sigma \in \Sigma_i$: $\sigma$ is visible by $C_i$, and there exists $(l_i, g_i, \sigma, \_, \_) \in T_i$ with $e_i(g_i) = \texttt{True}$.*
- *(No higher priorities enabled: distributed version) for all $\tau \in \Sigma$ with $\sigma \prec \tau$, and $\tau$ is visible by $C_i$: there is a $j \in \{1, \dots, m\}$ such that $\tau \in \Sigma_j$ and either $(l_j, g_j, \tau, \_, \_) \notin T_j$ or for every $(l_j, g_j, \tau, \_, \_) \in T_j$, $e_j(g_j) = \texttt{False}$.*

A configuration $c' = (l'_1, e'_1, \dots, l'_m, e'_m)$ is a *distributed $\sigma$-successor* of $c$ if $\sigma$ is distributively-enabled and $c'$ is a $\sigma$-successor of $c$. *D*istributed runs are runs of system $\mathcal{S}$ under communication architecture $Com$.

Any move from a configuration to a successor configuration in the distributed semantics can be understood as a multi-player game with $(|C| + 1)$ players between controllers $\mathsf{Ctrl}_i$ for each component and the external environment $\mathsf{Env}$. In contrast to the two-player game for the global semantics, $\mathsf{Ctrl}_i$ now is only informed on the intended next moves of the components in the visible region as defined by the communication architecture, and the control players play against the environment player. First, based

[1]For the example in the introduction, to increase readability, we omit listing the communication structure for self-transmission and group transmission.

on the visibility, the control players agree (cmp. Assumption 2 below) on an interaction $\sigma \in \Sigma_c$, and, second, the environment chooses a $\sigma$-enabled transition for each component $C_i$ with $\sigma \in \Sigma_i$. Now the successor state is obtained by local updates to the local configurations for each component and variables are non-deterministically toggled by the environment.

**Proposition 1** *Consider a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ under a deployable communication architecture Com.* (a) *If $\sigma \in \Sigma$ is globally enabled at configuration c, then $\sigma$ is distributively-enabled at c.* (b) *The set of distributively-enabled interactions at configuration c equals $\Sigma_c$.* (c) *If configuration c has no distributively-enabled interaction, it has no globally enabled interaction.*

From the above proposition (part c) we can conclude that if configuration $c$ has no distributively-enabled interaction, then $c$ is deadlocked ($c \in C_{dead}$). However we are looking for an explicit guarantee for the claim that the system at configuration $c$ is never deadlocked whenever there exists one distributively-enabled interaction in $c$. This means that whenever a race condition over a shared resource happens, it will be resolved (e.g., via the resource itself) rather than halting permanently and disabling the progress. Such an assumption can be fulfilled by variants of distributed consensus algorithms such as majority voting (MJRTY) [7].

**Assumption 2 (Runtime Assumption)** *For a configuration c with $|\Sigma_c| > 0$, the distributed controllers $\mathsf{Ctrl}_i$ agree on a distributively-enabled interaction $\sigma \in \Sigma_c$ for execution.*

The assumption assumes that the distributed semantics of a system can be implemented as the global semantics [4]. With the above assumption, we then define, given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ under a communication architecture *Com*, the set of deadlock states of $\mathcal{S}$ in distributed execution to be $C_{dist.dead} = \{c\}$ where $c$ has no distributively-enabled interaction. We immediately derive $C_{dist.dead} = C_{dead}$, as the left inclusion ($C_{dist.dead} \subseteq C_{dead}$) is the consequence of Proposition 1, and the right inclusion is trivially true. With such an equality, given a risk configuration $C_{risk}$ and global deadlock states $C_{dead}$, we say that system $S$ under the distributed semantics is *distributively-safe* if there is no distributed run $c_0, \ldots, c_k$ such that $c_k \in C_{dead} \cup C_{risk}$; a system that is not safe is called *distributively-unsafe*.

**Definition 4** *Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ together with a deployable communication architecture Com, the set of risk configurations $C_{risk} \subseteq C_{\mathcal{S}}$, a set of priorities $\mathcal{P}_{d+}$ is a solution to the* distributed priority synthesis problem *if the following holds: 1) $\mathcal{P} \cup \mathcal{P}_{d+}$ is transitive and irreflexive. 2) $(C, \Sigma, \mathcal{P} \cup \mathcal{P}_{d+})$ is distributively-safe. 3) For all $i, j \in \{1, \ldots, m\}$ s.t. $\sigma \in \Sigma_i$, $\tau \in \Sigma_j$, if $\sigma \prec \tau \in \mathcal{P} \cup \mathcal{P}_{d+}$ then $C_j \rightsquigarrow C_i \in Com$.*

The 3rd condition states that newly introduced priorities are indeed deployable. Notice that for system $\mathcal{S}$ with a deployable communication architecture *Com*, and any risk configurations $C_{risk}$ and global deadlock states $C_{dead}$, a solution to the distributed priority synthesis problem is distributively-safe iff it is (globally) safe. Moreover, for a fully connected communication architecture, the problem of distributed priority synthesis reduces to (global) priority synthesis.

**Theorem 1** *Given system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ under a deployable communication architecture Com, the problem of distributed priority synthesis is NP-complete to $|Q| + |\delta| + |\Sigma|$, where $|Q|$ and $|\delta|$ are the size of vertices and transitions in the product graph induced by $\mathcal{S}$, provided that $|C|^2 < |Q| + |\delta| + |\Sigma|$.*

(Sketch; see technical report [8] for full proof) First select a set of priorities (including $\mathcal{P}$) and check if they satisfy transitivity, irreflexivity, architectural constraints. Then check, in polynomial time, if the system under this set of priorities can reach deadlock states; hardness follows from hardness of global priority synthesis.
**(Example in Fig. 1)** The priority set $\{a \prec c, a \prec d\}$ is a feasible solution of distributed priority synthesis, as these priorities can be supported by the communication $C_2 \rightsquigarrow C_1$.

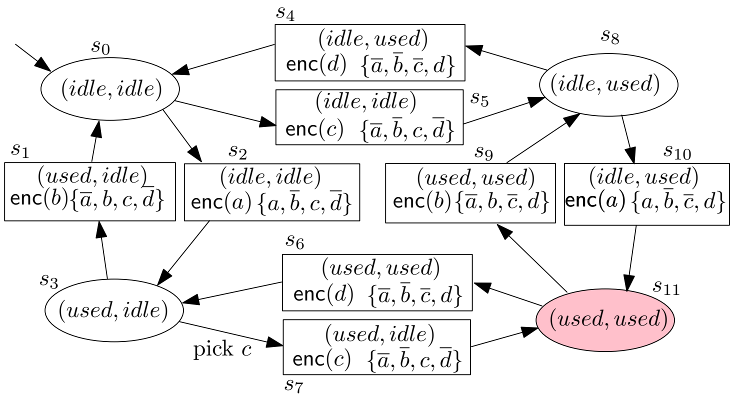

$s_0$
$(idle, idle)$
$s_4$
$(idle, used)$
$\mathsf{enc}(d)$ $\{\overline{a}, \overline{b}, \overline{c}, d\}$
$s_8$
$(idle, used)$
$(idle, idle)$
$\mathsf{enc}(c)$ $\{\overline{a}, \overline{b}, c, \overline{d}\}$
$s_5$
$s_1$
$(used, idle)$
$\mathsf{enc}(b)\{\overline{a}, b, c, \overline{d}\}$
$s_2$
$(idle, idle)$
$\mathsf{enc}(a)$ $\{a, \overline{b}, c, \overline{d}\}$
$s_9$
$(used, used)$
$\mathsf{enc}(b)$ $\{\overline{a}, b, \overline{c}, d\}$
$s_{10}$
$(idle, used)$
$\mathsf{enc}(a)$ $\{a, \overline{b}, \overline{c}, d\}$
$s_6$
$(used, used)$
$\mathsf{enc}(d)$ $\{\overline{a}, \overline{b}, \overline{c}, d\}$
$s_3$
$(used, idle)$
$s_{11}$
$(used, used)$
pick $c$
$(used, idle)$
$\mathsf{enc}(c)$ $\{\overline{a}, \overline{b}, c, \overline{d}\}$
$s_7$

Figure 2: The symbolic encoding for the system in Fig. 1.

## 4 Algorithmic Issues

It is not difficult to derive from the NP-completeness result (Section 3) a DPLL-like search algorithm (see technical report [8] for such an algorithm), where each possible priority $\sigma \prec \tau$ is represented as a Boolean variable $\underline{\sigma \prec \tau}$. If $\underline{\sigma \prec \tau}$ is evaluated to `True`, then it is introduced in the priority. Then the algorithm checks if such an introduced set is sufficient to avoid entering the risk. Notice, however, that checking whether a risk state is reachable is expensive[2]. As an optimization we therefore extend the basic search algorithm above with a diagnosis-based fixing process. In particular, whenever the system is unsafe under the current set of priorities, the algorithm diagnoses the reason for unsafety and introduces additional priorities for preventing immediate entry into states leading to unsafe states. If it is possible for the current scenario to be fixed, the algorithm immediately stops and returns the fix. Otherwise, the algorithm selects a set of priorities (from reasoning the inability of fix) and uses them to guide the introduction of new priorities in the search algorithm.

The diagnosis-based fixing process proceeds in two steps:

**(Step 1: Deriving fix candidates)** Game solving is used to derive potential fix candidates represented as a set of priorities. In the distributed case, we need to encode visibility constraints: they specify for each interaction $\sigma$, the set of other interactions $\Sigma_\sigma \subseteq \Sigma$ visible to the components executing $\sigma$ (Section 4.1). With visibility constraints, our game solving process results into a *nested attractor computation* (Section 4.2).

**(Step 2: Fault-fixing)** Then create from fix candidates one feasible fix via solving a corresponding SAT problem, which encodes properties of priorities and architectural restrictions (Section 4.3). If this propositional formula is unsatisfiable, then an unsatisfiable core is used to extract potentially useful candidate priorities.

### 4.1 Game Construction

Symbolic encodings of interacting components form the basis of reachability checks, the diagnosis process, and the algorithm for priority fixing (here we use $\mathcal{P}$ for $\mathcal{P}_{tran}$). In particular, symbolic encodings of system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ use the following propositional variables:

- $p0$ indicates whether it is the controller's or the environment's turn.

[2]This suffers from the state-explosion problem. Therefore, if the size of the input is defined not to be the set of all reachable states but rather the number of components together with the size of each component, the problem is in PSPACE.

**Algorithm 1:** Generate controllable transitions and the set of deadlock states

```
input  : System $\mathcal{S} = (C = (C_1,\dots,C_m),\Sigma,\mathcal{P})$, visibility constraint $\mathsf{Vis}^{\sigma_1}_{\sigma_2}$ where $\sigma_1,\sigma_2 \in \Sigma$
output: Transition predicate $\mathcal{T}_{ctrl}$ for control and the set of deadlock states $C_{dead}$
begin
       let predicate $\mathcal{T}_{ctrl} = \mathtt{False}$, $C_{dead} := \mathtt{True}$
       for $\sigma \in \Sigma$ do
           let predicate $P_\sigma := \mathtt{True}$
       for $\sigma \in \Sigma$ do
           for $i = \{1,\dots,m\}$ do
1              if $\sigma \in \Sigma_i$ then $P_\sigma := P_\sigma \wedge \bigvee_{(l,g,\sigma,f,l')\in T_i}(enc(l) \wedge g)$
2          $C_{dead} := C_{dead} \wedge \neg P_\sigma$
       for $\sigma_1 \in \Sigma$ do
3          let predicate $\mathcal{T}_{\sigma_1} := p0 \wedge \neg p0' \wedge P_{\sigma_1} \wedge \mathsf{enc}'(\sigma_1) \wedge \sigma'_1$
           for $\sigma_2 \in \Sigma, \sigma_2 \neq \sigma_1$ do
4              if $\mathit{Vis}^{\sigma_2}_{\sigma_1} = \mathtt{True}$ then $\mathcal{T}_{\sigma_1} := \mathcal{T}_{\sigma_1} \wedge (P_{\sigma_2} \leftrightarrow \sigma'_2)$
5              else $\mathcal{T}_{\sigma_1} := \mathcal{T}_{\sigma_1} \wedge \neg\sigma'_2$
           for $i = \{1,\dots,m\}$ do
6              $\mathcal{T}_{\sigma_1} := \mathcal{T}_{\sigma_1} \wedge \bigwedge_{y\in Y_i} y \leftrightarrow y' \wedge \bigwedge_{v\in V_i} v \leftrightarrow v'$
7          $\mathcal{T}_{ctrl} := \mathcal{T}_{ctrl} \vee \mathcal{T}_{\sigma_1}$
       for $\sigma_1 \prec \sigma_2 \in \mathcal{P}$ do
8          $\mathcal{T}_{ctrl} := \mathcal{T}_{ctrl} \wedge ((\sigma'_1 \wedge \sigma'_2) \rightarrow \neg\mathsf{enc}'(\sigma_1))$;
9          $\mathcal{T}_{12} = \mathcal{T}_{ctrl} \wedge (\sigma'_1 \wedge \sigma'_2)$; $\mathcal{T}_{ctrl} := \mathcal{T}_{ctrl} \setminus \mathcal{T}_{12}$;
10         $\mathcal{T}_{12,fix} := (\exists\sigma'_1 : \mathcal{T}_{12}) \wedge (\neg\sigma'_1)$;
11         $\mathcal{T}_{ctrl} := \mathcal{T}_{ctrl} \vee \mathcal{T}_{12,fix}$
       return $\mathcal{T}_{ctrl}$, $C_{dead}$
```

- $A = \{a_1,\dots,a_{\lceil\log_2|\Sigma|\rceil}\}$ for the binary encoding $\mathsf{enc}(\sigma)$ of the *chosen interaction* $\sigma$ (which is agreed by distributed controllers for execution, see Assumption 2).
- $\bigcup_{\sigma\in\Sigma}\{\sigma\}$ are the variables representing interactions to encode *visibility*. Notice that the same letter is used for an interaction and its corresponding encoding variable.
- $\bigcup_{i=1}^{m} Y_i$, where $Y_i = \{y_{i1},\dots,y_{ik}\}$ for the binary encoding $enc(l)$ of locations $l \in L_i$.
- $\bigcup_{i=1}^{m}\bigcup_{v\in V_i}\{v\}$ are the encoding of the component variables.

Primed variables are used for encoding successor configurations and transition relations. Visibility constraints $\mathsf{Vis}^{\tau}_{\sigma} \in \{\mathtt{True}, \mathtt{False}\}$ denote the visibility of interaction $\tau$ over another interaction $\sigma$. It is computed statically: such a constraint $\mathsf{Vis}^{\tau}_{\sigma}$ holds iff for $C_i, C_j \in C$ where $\tau \in \Sigma_i$ and $\sigma \in \Sigma_j$, $C_i \rightsquigarrow C_j \in Com$.
**(Example in Fig. 1)** We illustrate the symbolic game encoding using Fig. 2 to offer an insight. Circles are states which is the turn of the controller ($p0 = \mathtt{True}$) and squares are those of the environment's turn ($p0 = \mathtt{False}$). The system starts with state $s_0$ and proceeds by selecting an interaction that is distributively enabled. E.g., $C_1$ may decide to execute $a$, and the state then goes to $s_2$. In $s_2$, we have $\mathsf{enc}(a)$ to represent that $a$ is under execution. We also have $\{a,\overline{b},c,\overline{d}\}$ as the encoded visibility, as $C_2 \rightsquigarrow C_1$ and the availability of $c$ and $d$ can be passed. This is used to represent that when $a$ is selected, $C_1$ is *sure* that $c$ is enabled at $C_2$. Then as no non-deterministic update is in the system, the environment just moves to the successor by updating the local state to the destination of the interaction $a$. Notice that if $C_2$ decides to execute $c$, the play enters state $s_5$, which has visibility $\{\overline{a},\overline{b},c,\overline{d}\}$. Such a visibility reflects the fact that when $c$ executes, $C_2$ is *not aware of* the availability of $C_1$ to execute $a$, which is due to the architectural constraint.

Following the above explanation, Algorithms 1 and 2 return symbolic transitions $\mathcal{T}_{ctrl}$ and $\mathcal{T}_{env}$ for the control players $\bigcup_{i=1}^{m}\mathsf{Ctrl}_i$ and the player $\mathsf{Env}$ respectively, together with the creation of a symbolic representation $C_{dead}$ for the deadlock states of the system. Line 1 of algorithm 1 computes when an

**Algorithm 2:** Generate uncontrollable updates

```
input  : System $\mathcal{S} = (C = (C_1,\ldots,C_m),\Sigma,\mathcal{P})$
output: Transition predicate $\mathcal{T}_{env}$ for environment
begin
    let predicate $\mathcal{T}_{env}$ := False
    for $\sigma \in \Sigma$ do
        let predicate $T_\sigma := \neg p0 \wedge p0'$
        for $i = \{1,\ldots,m\}$ do
            if $\sigma \in \Sigma_i$ then
1               $T_\sigma := T_\sigma \wedge \bigvee_{(l,g,\sigma,f,l')\in T_i}(enc(l) \wedge g \wedge enc'(l') \wedge \mathsf{enc}(\sigma) \wedge \mathsf{enc}'(\sigma) \wedge \bigwedge_{v\in V_i} \cup_{e\in f(v)} v' \leftrightarrow e)$
        for $\sigma_1 \in \Sigma, \sigma_1 \neq \sigma$ do
2           $T_\sigma := T_\sigma \wedge \sigma_1' =$ False
        for $i = \{1,\ldots,m\}$ do
3           if $\sigma \notin \Sigma_i$ then $T_\sigma := T_\sigma \wedge \bigwedge_{y\in Y_i} y \leftrightarrow y' \wedge \bigwedge_{v\in V_i} v \leftrightarrow v'$
        $\mathcal{T}_{env} := \mathcal{T}_{env} \vee T_\sigma$
    return $\mathcal{T}_{env}$
```

interaction $\sigma$ is enabled. Line 2 summarizes the conditions for deadlock, where none of the interaction is enabled. The computed deadlock condition can be reused throughout the subsequent synthesis process, as introducing a set of priorities never introduces new deadlocks. In line 3, $\mathcal{T}_{\sigma_1}$ constructs the actual transition, where the conjunction with $\mathsf{enc}'(\sigma_1)$ indicates that $\sigma_1$ is the *chosen interaction* for execution. $\mathcal{T}_{\sigma_1}$ is also conjoined with $\sigma_1'$ as an indication that $\sigma_1$ is enabled (and it can see itself). Line 4 and 5 record the visibility constraint. If interaction $\sigma_2$ is visible by $\sigma_1$ ($\mathsf{Vis}_{\sigma_1}^{\sigma_2} = \mathtt{True}$), then by conjoining it with $(P_{\sigma_2} \leftrightarrow \sigma_2')$, $\mathcal{T}_{\sigma_1}$ explicitly records the set of *visible and enabled (but not chosen)* interactions. If interaction $\sigma_2$ is not visible by $\sigma_1$, then the encoding conjuncts with $\neg\sigma_2'$. In this case $\sigma_2$ is *treated as if it is not enabled* (recall state $s_5$ in Fig. 2): if $\sigma_1$ is a bad interaction leading to the attractor of deadlock states, we cannot select $\sigma_2$ as a potential escape (i.e., we cannot create fix-candidate $\sigma_1 \prec \sigma_2$), as $\sigma_1 \prec \sigma_2$ is not supported by the visibility constraints derived by the architecture. Line 6 keeps all variables and locations to be the same in the pre- and postcondition, as the actual update is done by the environment. For each priority $\sigma_1 \prec \sigma_2$, lines from 8 to 11 perform transformations on the set of transitions where both $\sigma_1$ and $\sigma_2$ are enabled. Line 8 prunes out transitions from $\mathcal{T}_{ctrl}$ where both $\sigma_1$ and $\sigma_2$ are enabled but $\sigma_1$ is chosen for execution. Then, the codes in lines 9 to 11 ensure that for remaining transitions $\mathcal{T}_{12}$, they shall change the view as if $\sigma_1$ is not enabled (line 10 performs the fix). $\mathcal{T}_{ctrl}$ is updated by removing $\mathcal{T}_{12}$ and adding $\mathcal{T}_{12,fix}$.

**Proposition 2** *Consider configuration s, where interaction $\sigma$ is (enabled and) chosen for execution. Given $\tau \in \Sigma$ at s such that the encoding $\tau' = \mathtt{True}$ in Algorithm 1, then $\mathsf{Vis}_{\sigma}^{\tau} = \mathtt{True}$ and interaction $\tau$ is also enabled at s.*

**Proposition 3** *$C_{dead}$ as returned by algorithm 1 encodes the set of deadlock states of the input system $\mathcal{S}$.*

In Algorithm 2, the environment updates the configuration using interaction $\sigma$ based on the indicator $\mathsf{enc}(\sigma)$. Its freedom of choice in variable updates is listed in line 1 (i.e., $\cup_{e\in f(v)} v' \leftrightarrow e$). Line 2 sets all interactions $\sigma_1$ not chosen for execution to be false, and line 3 sets all components not participated in $\sigma$ to be stuttered.

## 4.2 Fixing Algorithm: Game Solving with Nested Attractor Computation

The first step of fixing is to compute the *nested-risk-attractor* from the set of bad states $C_{risk} \cup C_{dead}$. Let $V_{ctrl}$ ($\mathcal{T}_{ctrl}$) and $V_{env}$ ($\mathcal{T}_{env}$) be the set of control and environment states (transitions) in the encoded game.

**Algorithm 3:** Nested-risk-attractor computation

```
input  : Initial state $c_0$, risk states $C_{risk}$, deadlock states $C_{dead}$, set of reachable states $\mathcal{R}_{\mathcal{S}}(\{c_0\})$ and symbolic transitions $\mathcal{T}_{ctrl}$, $\mathcal{T}_{env}$
         from Algorithm 1 and 2
output: (1) Nested risk attractor $\mathsf{NestAttr}_{env}(C_{risk} \cup C_{dead})$ and (2) $\mathcal{T}_f \subseteq \mathcal{T}_{ctrl}$, the set of control transitions starting outside
         $\mathsf{NestAttr}_{env}(C_{dead} \cup C_{risk})$ but entering $\mathsf{NestAttr}_{env}(C_{risk} \cup C_{dead})$.
begin
       // Create architectural non-visibility predicate
1      let Esc := False
       for $\sigma_i \in \Sigma$ do
2          let $\mathsf{Esc}_{\sigma_i} := \mathsf{enc}'(\sigma_i)$
3          for $\sigma_j \in \Sigma, \sigma_j \neq \sigma_i$ do $\mathsf{Esc}_{\sigma_i} := \mathsf{Esc}_{\sigma_i} \wedge \neg\sigma'_j$
           $\mathsf{Esc} := \mathsf{Esc} \vee (\mathsf{Esc}_{\sigma_i} \wedge \sigma'_i)$
       // Part A: Prune unreachable transitions and bad states, $\mathcal{R}_{\mathcal{S}}(\{c_0\})$ is the current set of reachable
          states
       $\mathcal{T}_{ctrl} := \mathcal{T}_{ctrl} \wedge \mathcal{R}_{\mathcal{S}}(\{c_0\})$, $\mathcal{T}_{env} := \mathcal{T}_{ctrl} \wedge \mathcal{R}_{\mathcal{S}}(\{c_0\})$; $C_{dead} := C_{dead} \wedge \mathcal{R}_{\mathcal{S}}(\{c_0\})$, $C_{risk} := C_{risk} \wedge \mathcal{R}_{\mathcal{S}}(\{c_0\})$
       // Part B: Solve nested-safety game
       let $\mathsf{NestedAttr}_{pre} := C_{dead} \vee C_{risk}$, $\mathsf{NestedAttr}_{post}$ := False
4      while True do
           // B.1 Compute risk attractor of $\mathsf{NestedAttr}_{pre}$
5          let Attr := compute_risk_attr($\mathsf{NestedAttr}_{pre}$,$\mathcal{T}_{env}$,$\mathcal{T}_{ctrl}$)
           // B.2 Generate transitions with source in ¬Attr and destination in Attr
6          PointTo := $\mathcal{T}_{ctrl} \wedge$ SUBS(($\exists\Xi'$ : Attr),$\Xi$,$\Xi'$))
7          OutsideAttr := ¬Attr ∧ ($\exists\Xi' : \mathcal{T}_{ctrl}$)
8          $\mathcal{T}$ := PointTo ∧ OutsideAttr
           // B.3 Add the source vertex of B.2 to NestedAttr
9          newBadStates := $\exists\Xi' : (\mathcal{T} \wedge \mathsf{Esc})$
10         $\mathsf{NestedAttr}_{post}$ := Attr ∨ newBadStates
           // B.4 Condition for breaking the loop
           if $\mathit{NestedAttr}_{pre} \leftrightarrow \mathit{NestedAttr}_{post}$ then break
           else $\mathsf{NestedAttr}_{pre} := \mathsf{NestedAttr}_{post}$
       // Part C: extract $\mathcal{T}_f$
11     PointToNested := $\mathcal{T}_{ctrl} \wedge$ SUBS(($\exists\Xi'$ : $\mathsf{NestedAttr}_{pre}$),$\Xi$,$\Xi'$))
12     OutsideNestedAttr := $\neg\mathsf{NestedAttr}_{pre} \wedge (\exists\Xi' : \mathcal{T}_{ctrl})$
13     $\mathcal{T}_f$ := PointToNested ∧ OutsideNestedAttr
       return $\mathsf{NestAttr}_{env}(C_{dead} \cup C_{risk}) := \mathsf{NestedAttr}_{pre}$, $\mathcal{T}_f$
```

Let *risk-attractor* $\mathsf{Attr}_{env}(X) := \bigcup_{k \in \mathbf{N}} \mathsf{attr}^k_{env}(X)$, where $\mathsf{attr}_{env}(X) := X \cup \{v \in V_{env} \mid v\mathcal{T}_{env} \cap X \neq \emptyset\} \cup \{v \in V_{ctrl} \mid \emptyset \neq v\mathcal{T}_{ctrl} \subseteq X\}$, i.e., $\mathsf{attr}_{env}(X)$ extends state sets $X$ by all those states from which either environment can move to $X$ within one step or control cannot prevent to move within the next step. ($v\mathcal{T}_{env}$ denotes the set of environment successors of $v$, and $v\mathcal{T}_{ctrl}$ denotes the set of control successors of $v$.) Then $\mathsf{Attr}_{env}(C_{risk} \cup C_{dead}) := \bigcup_{k \in \mathbf{N}} \mathsf{attr}^k_{env}(C_{risk} \cup C_{dead})$ contains all nodes from which the environment can force any play to visit the set $C_{risk} \cup C_{dead}$.

**(Example in Fig. 1)** Starting from the risk state $s_{11} = (used, used)$, in attractor computation we first add $\{s_{10}, s_7\}$ into the attractor, as they are environment states and each of them has an edge to enter the attractor. Then the attractor computation saturates, as for state $s_3$ and $s_8$, each of them has one edge to escape from entering the attractor. Thus $\mathsf{Attr}_{env}(C_{risk} \cup C_{dead}) = \{s_{10}, s_7, s_{11}\}$.

Nevertheless, nodes outside the risk-attractor are not necessarily safe due to visibility constraints. We again use Fig. 2 to illustrate such a concept. State $s_3$ is a control location, and it is outside the attractor: although it has an edge $s_3 \to s_7$ which points to the risk-attractor, it has another edge $s_3 \to s_1$, which does not lead to the attractor. We call positions like $s_3$ as *error points*. Admittedly, applying priority $c \prec b$ at $s_3$ is sufficient to avoid entering the attractor. However, as $\mathsf{Vis}^c_b = \mathtt{False}$, then for $C_2$ who tries to execute $c$, it is unaware of the enableness of $b$. So $c$ can be executed freely by $C_2$. Therefore, we should add $s_3$ explicitly to the (already saturated) attractor, and *recompute the attractor* due to the inclusion of new vertices. This leads to an extended computation of the risk-attractor (i.e., nested-risk-attractor).

**Definition 5** *The* nested-risk-attractor $\mathsf{NestAttr}_{env}(C_{risk} \cup C_{dead})$ *is the smallest superset of* $\mathsf{Attr}_{env}(C_{risk} \cup C_{dead})$ *such that the following holds.*

1. *For state $c \notin \mathsf{NestAttr}_{env}(C_{risk} \cup C_{dead})$, where there exists a (bad-entering) transition $t \in \mathcal{T}_{ctrl}$ with source $c$ and target $c' \in \mathsf{NestAttr}_{env}(C_{risk} \cup C_{dead})$:*
   - (Good control state shall have one escape) *there exists another transition $t' \in \mathcal{T}_{ctrl}$ such that its source is $c$ but its destination $c'' \notin \mathsf{NestAttr}_{env}(C_{risk} \cup C_{dead})$.*
   - (Bad-entering transition shall have another visible candidate) *for every bad-entering transition $t$ of $c$, in the encoding let $\sigma$ be the chosen interaction for execution ($\mathsf{enc}'(\sigma) = \mathtt{True}$). Then there exists another interaction $\tau$ such that, in the encoding, $\tau' = \mathtt{True}$.*
2. (Add if environment can enter) *If $v \in V_{env}$, and $v\mathcal{T}_{env} \cap \mathsf{NestAttr}_{env}(C_{risk} \cup C_{dead}) \neq \emptyset$, then $v \in \mathsf{NestAttr}_{env}(C_{risk} \cup C_{dead})$.*

Algorithm 3 uses a nested fixpoint for computing a symbolic representation of a nested risk attractor. The notation $\exists\Xi$ ($\exists\Xi'$) is used to represent existential quantification over all umprimed (primed) variables used in the system encoding. Moreover, we use the operator $\mathtt{SUBS}(X,\Xi,\Xi')$, as available in many BDD packages, for variable swap (substitution) from unprimed to primed variables in $X$. For preparation (line 1 to 3), we first create a predicate, which explicitly records when an interaction $\sigma_i$ is enabled and chosen (i.e., $\sigma'_i = \mathtt{True}$ and $\mathsf{enc}'(\sigma_i) = \mathtt{True}$). For every other interaction $\sigma_j$, the variable $\sigma'_j$ is evaluated to $\mathtt{False}$ in BDD (i.e., either it is disabled or not visible by $\sigma_i$, following Algorithm 1, line 4 and 5).

The nested computation consists of two while loops (line 4, 5): B.1 computes the familiar risk attractor (see definition stated earlier, and we refer readers to [8] for concrete algorithms), and B.2 computes the set of transitions $\mathcal{T}$ whose source is outside the attractor but the destination is inside the attractor. Notice that for every source vertex $c$ of a transition in $\mathcal{T}$: (1) It has chosen an interaction $\sigma \in \Sigma$ to execute, but it is a bad choice. (2) There exists another choice $\tau$ whose destination is outside the attractor (otherwise, $c$ shall be in the attractor). However, such $\tau$ may not be visible by $\sigma$. Therefore, $\exists\Xi' : (\mathcal{T} \wedge \mathsf{Esc})$ creates those states without any *visible escape*, i.e., without any other visible and enabled interactions under the local view of the chosen interaction. These states form the set of new bad states $\mathsf{newBadStates}$ due to architectural limitations. Finally, Part C of the algorithm extracts $\mathcal{T}_f$ (similar to extracting $\mathcal{T}$ in B.2).

Algorithm 3 terminates, since the number of states that can be added to $\mathsf{Attr}$ (by $\mathtt{compute\_risk\_attr}$) and $\mathsf{NestedAttr}_{post}$ (in the outer-loop) is finite. The following proposition is used to detect the infeasibility of distributed priority synthesis problems.

**Proposition 4** *Assume when executing the fix algorithm, only $\underline{\sigma \prec \tau}$, where $\sigma \prec \tau \in \mathcal{P}$, is evaluated to true. If the encoding of the initial state is contained in $\mathsf{NestAttr}_{env}(C_{risk} \cup C_{dead})$, then the distributed priority synthesis problem for $\mathcal{S}$ with $C_{risk}$ is infeasible.*

**(Example in Fig. 1)** We again use the encoded game in Fig. 2 to illustrate the underlying algorithm for nested-attractor computation. Line 5 computes the attractor $\{s_{10}, s_7, s_{11}\}$, and then Line 6 to 8 creates the set of transitions $\mathcal{T} = \{s_3 \rightarrow s_7, s_8 \rightarrow s_{10}\}$, which are transitions whose source is outside the attractor but the destination is inside. Remember that $\mathsf{Esc}$ in Line 2 generates a predicate which allows, when $\sigma$ is enabled, only interaction $\sigma$ itself to be visible. Then $\mathsf{newBadStates} = \exists\Xi' : \mathcal{T} \wedge \mathsf{Esc} = \exists\Xi' : \{s_3 \rightarrow s_7\} = \{s_3\}$, implying that such a state is also considered as bad. Then in Line 10 add $\{s_3\}$ to the new attractor and recompute. The computation saturates with the following set of bad states $\{s_2, s_3, s_7, s_{10}, s_{11}\}$. For state $s_0$ it has the risk edge $s_0 \rightarrow s_2$, but it can be blocked by priority $a \prec c$ (recall in $s_2$ we have visibility $\{a, \overline{b}, c, \overline{d}\}$). For state $s_8$ it has the risk edge $s_8 \rightarrow s_{10}$, but it can be blocked by priority $a \prec d$. Thus $\mathcal{T}_f = \{s_0 \rightarrow s_2, s_8 \rightarrow s_{10}\}$, and from $\mathcal{T}_f$ we can extract the candidate of the priority fix $\{a \prec c, a \prec d\}$.

Notice that a visible escape is not necessarily a "true escape" as illustrated in Figure 3. It is possible that for state $c_2$, for $g$ its visible escape is $a$, while for $a$ its visible escape is $g$. Therefore, it only suggests candidates of fixing, and in these cases, a feasible fix is derived in a SAT resolution step (Section 4.3).

## 4.3 Fixing Algorithm: SAT Problem Extraction and Conflict Resolution

The returned value $\mathcal{T}_f$ of Algorithm 3 contains not only the risk interactions but also all possible interactions which are visible and enabled (see Algorithm 1 for encoding, Proposition 2 for result). Consider, for example, the situation depicted in Figure 3 and assume that $\mathsf{Vis}_a^c$, $\mathsf{Vis}_a^b$, $\mathsf{Vis}_b^c$, $\mathsf{Vis}_g^a$, and $\mathsf{Vis}_b^a$ are the only visibility constraints which hold `True`. If $\mathcal{T}_f$ returns three transitions, one may extract fix candidates from each of these transitions in the following way.

- On $c_2$, $a$ enters the nested-risk-attractor, while $b,c$ are also visible from $a$; one obtains the candidates $\{a \prec b, a \prec c\}$.
- On $c_2$, $g$ enters the nested-risk-attractor, while $a$ is also visible from $g$; one obtains the candidate $\{g \prec a\}$.
- On $c_8$, $b$ enters the nested-risk-attractor, while $a$ is also visible; one obtains the candidate $\{b \prec a\}$.

Using these candidates, one can perform *conflict resolution* and generate a set of new priorities for preventing entry into the nested-risk-attractor region. For example, $\{a \prec c, g \prec a, b \prec a\}$ is such a set of priorities for ensuring the safety condition. Notice also that the set $\{a \prec b, g \prec b, b \prec a\}$ is circular, and therefore not a valid set of priorities.

In our implementation, conflict resolution is performed using SAT solvers. Priority $\sigma_1 \prec \sigma_2$ is presented as a Boolean variable $\underline{\sigma_1 \prec \sigma_2}$. If the generated SAT problem is satisfiable, for all variables $\underline{\sigma_1 \prec \sigma_2}$ which is evaluated to `True`, we add priority $\sigma_1 \prec \sigma_2$ to the resulting introduced priority set $\mathcal{P}_{d+}$. The constraints below correspond to the ones for global priority synthesis framework [9].

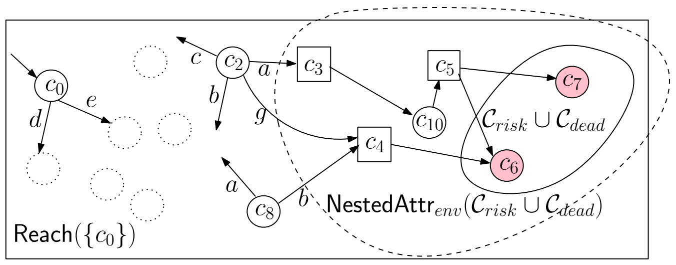
- *(1. Priority candidates)* For each edge $t \in \mathcal{T}_f$ which enters the risk attractor using $\sigma$ and having $\sigma_1,\ldots,\sigma_e$ visible escapes (excluding $\sigma$), create clause $(\bigvee_{i=1}^{e} \underline{\sigma \prec \sigma_i})$.
- *(2. Existing priorities)* For each priority $\sigma \prec \tau \in \mathcal{P}$, create clause $(\underline{\sigma \prec \tau})$.
- *(3. Irreflexive)* For each interaction $\sigma$ used in (1) and (2), create clause $(\neg \underline{\sigma \prec \sigma})$.
- *(4. Transitivity)* For any $\sigma_1,\sigma_2,\sigma_3$ used above, create a clause $((\underline{\sigma_1 \prec \sigma_2} \wedge \underline{\sigma_2 \prec \sigma_3}) \Rightarrow \underline{\sigma_1 \prec \sigma_3})$.

Clauses for architectural constraints also need to be added in the case of distributed priority synthesis. For example, if $\sigma_1 \prec \sigma_2$ and $\sigma_2 \prec \sigma_3$ then due to transitivity we shall include priority $\sigma_1 \prec \sigma_3$. But if $\mathsf{Vis}_{\sigma_1}^{\sigma_3} =$ `False`, then $\sigma_1 \prec \sigma_3$ is not supported by communication. In the above example, as $\mathsf{Vis}_b^c =$ `True`, $\{a \prec c, g \prec a, b \prec a\}$ is a legal set of priority fix satisfying the architecture (because the inferred priority $b \prec c$ is supported). Therefore, we introduce the following constraints.

- *(5. Architectural Constraint)* Given $\sigma_1,\sigma_2 \in \Sigma$, if $\mathsf{Vis}_{\sigma_1}^{\sigma_2} =$ `False`, then $\underline{\sigma_1 \prec \sigma_2}$ is evaluated to `False`.
- *(6. Communication Constraint)* Given $\sigma_1,\sigma_2 \in \Sigma$, if $\mathsf{Vis}_{\sigma_1}^{\sigma_2} =$ `False`, for any interaction $\sigma_3 \in \Sigma$, if $\mathsf{Vis}_{\sigma_1}^{\sigma_3} = \mathsf{Vis}_{\sigma_3}^{\sigma_2} =$ `True`, at most one of $\underline{\sigma_1 \prec \sigma_3}$ or $\underline{\sigma_3 \prec \sigma_2}$ is evaluated to `True`.

A correctness argument of this fixing process can be found in our extended report [8].

**(Example in Fig. 1)** By the nested attractor computation we have created a priority fix candidate $\{a \prec c, a \prec d\}$. Such a fix satisfies the above 6 types of constraints and thus is a feasible solution for distributed priority synthesis.



$c_0$
$d$
$e$
$c$
$c_2$
$b$
$a$
$g$
$c_3$
$c_4$
$c_5$
$c_{10}$
$c_7$
$c_6$
$\mathcal{C}_{risk} \cup \mathcal{C}_{dead}$
$a$
$c_8$
$b$
$\mathsf{NestedAttr}_{env}(\mathcal{C}_{risk} \cup \mathcal{C}_{dead})$
$\mathsf{Reach}(\{c_0\})$

Figure 3: Locating fix candidates outside from the nested-risk-attractor.

## 5 Implementation

Our algorithm for solving the distributed priority synthesis problem has been implemented in Java on top of the open-source workbench VissBIP[3] for graphically editing and visualizing systems of interacting components. The synthesis engine itself is based on the JDD package for binary decision diagrams, and the SAT4J propositional satisfiability solver. In addition, we implemented a number of extensions and optimizations (e.g., Proposition 4) to the core algorithm in Section 4; for lack of space details needed to be omitted.

First, we also use the result of the unsatisfiable core during the fix process to guide introducing new priorities. E.g., if the fix does not succeed as both $\sigma \prec \tau$ and $\tau \prec \sigma$ are used, the engine then introduces $\underline{\sigma \prec \tau}$ for the next iteration. Then in the next diagnosis process, the engine can not propose a fix of the form $\tau \prec \sigma$ (as to give such a fix by the engine, it requires that when $\tau$ and $\sigma$ are enabled while $\tau$ is chosen for execution, $\sigma$ is also enabled; the enableness of $\sigma$ contradicts $\sigma \prec \tau$).

Second, we are over-approximating the nested risk attractor by parsimoniously adding all source states in $\mathcal{T}_f$, as returned from Algorithm 3, to the nested-risk-attractor before recomputing; thereby increasing chances of creating a new $\mathcal{T}_f$ where conflicts can be resolved.

Lastly, whenever possible the implementation tries to synthesize local controllers without any state information. If such a diagnosis-fixing fails, the algorithm can also perform a model transformation of the interacting components which is equivalent to transmitting state information in the communication. In order to minimize the amount of state information that is required to communicate, we lazily extract refinement candidates from (minimal) unsatisfiable cores of failed runs of the propositional solver, and correspondingly refine the alphabet by including new state information. Alternatively, a fully refined model transformation can eagerly be computed in VissBIP.

## 6 Evaluation

We validate our algorithm using a collection of benchmarking models including memory access problem, power allocation assurance, and working protection in industrial automation; some of these case studies are extracted from industrial case studies. Table 1 summarizes the results obtained on an Intel Machine with 3.4 GHz Intel Core i7 CPU and 8 GB RAM. Besides runtime we also list the algorithmic extensions and optimizations described in Section 5.

The experiments 1.1 through 1.16 in Table 1 refer to variations of the multiprocessor scheduling problem with increasing number of processors and memory banks. Depending on the communication architectures the engine uses refinement or extracts the UNSAT core to find a solution.

[3] `http://www6.in.tum.de/~chengch/vissbip`.

Experiments 2.1 and 2.2 refer to a multi-robot scenario with possible moves in a predefined arena, and the goal is to avoid collision by staying within a predefined protection cap. The communication architecture is restricted in that the $i$-th robot can only notify the $((i+1)\%n)$-th.

In experiments 3.1 through 3.6 we investigate the classical dining philosopher problem using various communication architectures. If the communication is clockwise, then the engine fails to synthesize priorities[4]. If the communication is counter-clockwise (i.e., a philosopher can notify its intention to his right philosopher), then the engine is also able to synthesize distributed priorities (for $n$ philosophers, $n$ rules suffice). Compared to our previous priority synthesis technique, as in distributed priority synthesis we need to separate visibility and enabled interactions, the required time for synthesis is longer.

Experiment 4 is based on a case study for increasing the reliability of data processing units (DPUs) by using multiple data sampling. The mismatch between the calculated results from different devices may yield deadlocks. The deadlocks can be avoided with the synthesized priorities from VissBIP.

Finally, in experiment 5, we are synthesizing a decentralized controller for the Dala robot [3], which is composed of 20 different components. A hand-coded version of the control indeed did not rule out deadlocks. Without any further communication constraints between the components, VissBIP locates the deadlocks and synthesizes additional priorities to avoid them.

## 7 Related Work

Distributed controller synthesis is undecidable [20] even for reachability or simple safety conditions [14]. A number of decidable subproblems have been proposed either by restricting the communication structures between components, such as pipelined, or by restricting the set of properties under consideration [18, 17, 19, 12]; these restrictions usually limit applicability to a wide range of problems. Schewe and Finkbiner's [21] bounded synthesis work on LTL specifications: when using automata-based methods, it requires that each process shall obtain the same information from the environment. The method is extended to encode locality constraints to work on arbitrary structures. Distributed priority synthesis, on one hand, its starting problem is a given distributed system, together with an additional safety requirement (together with the progress/deadlock-freedom property) to ensure. On the other hand, it is also flexible enough to specify different communication architectures between the controllers such as master-slave in the multiprocessor scheduling example. To perform distributed execution, we have also explicitly indicated how such a strategy can be executed on concrete platforms.

Starting with an arbitrary controller Katz, Peled and Schewe [16, 15] propose a knowledge-based approach for obtaining a decentralized controller by reducing the number of required communication between components. This approach assumes a fully connected communication structure, and the approach fails if the starting controller is inherently non-deployable.

Bonakdarpour, Kulkarni and Lin [6] propose methods for adding fault-recoveries for BIP components. The algorithms in [5, 6] are orthogonal in that they add additional behavior, for example new transitions, for individual components instead of determinizing possible interactions among components as in distributed priority synthesis. However, distributed synthesis described by Bonakdarpour et al. [5] is restricted to local processes without joint interactions between components.

Lately, the problem of deploying priorities on a given architecture has gained increasing recognition [4, 2]; the advantage of priority synthesis is that the set of synthesized priorities is always known to

[4]Precisely, in our model, we allow each philosopher to pass his intention over his left fork to the philosopher of his left. The engine uses Proposition 4 and diagnoses that it is impossible to synthesize priorities, as the initial state is within the nested-risk-attractor.

Table 1: Experimental results on distributed priority synthesis

| Index | Testcase and communication architecture | Components | Interactions | Time (seconds) | Remark |
|---|---|---|---|---|---|
| 1.1 | 4 CPUs with broadcast A | 8 | 24 | 0.17 | x |
| 1.2 | 4 CPUs with local A, D | 8 | 24 | 0.25 | A |
| 1.3 | 4 CPUs with local communication | 8 | 24 | 1.66 | R |
| 1.4 | 6 CPUs with broadcast A | 12 | 36 | 1.46 | RP-2 |
| 1.5 | 6 CPUs with broadcast A, F | 12 | 36 | 0.26 | x |
| 1.6 | 6 CPUs with broadcast A, D, F | 12 | 36 | 1.50 | A |
| 1.7 | 6 CPUs with local communication | 12 | 36 | - | fail |
| 1.8 | 8 CPUs with broadcast A | 16 | 48 | 8.05 | RP-2 |
| 1.9 | 8 CPUs with broadcast A, H | 16 | 48 | 1.30 | x |
| 1.10 | 8 CPUs with broadcast A, D, H | 16 | 48 | 1.80 | x |
| 1.11 | 8 CPUs with broadcast A, B, G, H | 16 | 48 | 3.88 | RP-2 |
| 1.12 | 8 CPUs with local communication | 16 | 48 | 42.80 | R |
| 1.13 | 10 CPUs with broadcast A | 20 | 60 | 135.03 | RP-2 |
| 1.14 | 10 CPUs with broadcast A, J | 20 | 60 | 47.89 | RP-2 |
| 1.15 | 10 CPUs with broadcast A, E, F, J | 20 | 60 | 57.85 | RP-2 |
| 1.16 | 10 CPUs with local communication A, B, E, F, I, J | 20 | 60 | 70.87 | RP-2 |
| 2.1 | 4 Robots with 12 locations | 4 | 16 | 11.86 | RP-1 |
| 2.2 | 6 Robots with 12 locations | 6 | 24 | 71.50 | RP-1 |
| 3.1 | Dining Philosopher 10 (no communication) | 20 | 30 | 0.25 | imp |
| 3.2 | Dining Philosopher 10 (clockwise next) | 20 | 30 | 0.27 | imp |
| 3.3 | Dining Philosopher 10 (counter-clockwise next) | 20 | 30 | 0.18 | x (nor: 0.16) |
| 3.4 | Dining Philosopher 20 (counter-clockwise next) | 40 | 60 | 0.85 | x,g (nor: 0.55) |
| 3.5 | Dining Philosopher 30 (counter-clockwise next) | 60 | 90 | 4.81 | x,g (nor: 2.75) |
| 4 | DPU module (local communication) | 4 | 27 | 0.42 | x |
| 5 | Antenna module (local communication) | 20 | 64 | 17.21 | RP-1 |

[x] Satisfiable by direct fixing (without assigning any priorities)
[A] Nested-risk-attractor over-approximation
[R] State-based priority refinement
[RP-1] Using UNSAT core: start with smallest amount of newly introduced priorities
[RP-2] Using UNSAT core: start with a subset of local non-conflicting priorities extracted from the UNSAT core
[fail] Fail to synthesize priorities (time out > 150 seconds using RP-1)
[imp] Impossible to synthesize priorities from diagnosis at base-level (using Proposition 4)
[g] Initial variable ordering provided (the ordering is based on breaking the circular order to linear order)
[nor] Priority synthesis without considering architectural constraints (engine in [9])

be deployable.

## 8 Conclusion

We have presented a solution to the distributed priority synthesis problem for synthesizing deployable local controllers by extending the algorithm for synthesizing stateless winning strategies in safety games [10, 9]. We investigated several algorithmic optimizations and validated the algorithm on a wide range of synthesis problems from multiprocessor scheduling to modular robotics. Although these initial experimental results are indeed encouraging, they also suggest a number of further refinements and extensions.

The model of interacting components can be extended to include a rich set of data types by either using Boolean abstraction in a preprocessing phase or by using satisfiability modulo theory (SMT) solvers instead of a propositional satisfiability engine; in this way, one might also synthesize distributed controllers for real-time systems. Another extension is to to explicitly add the faulty or adaptive behavior by means of demonic non-determinism.

Distributed priority synthesis might not always return the most useful controller. For example, for the Dala robot, the synthesized controllers effectively shut down the antenna to obtain a deadlock-free system. Therefore, for many real-life applications we are interested in obtaining optimal, for example wrt. energy consumption, or Pareto-optimal controls.

Finally, the priority synthesis problem as presented here needs to be extended to achieve goal-oriented orchestration of interacting components. Given a set of goals in a rich temporal logic and a set of interacting components, the orchestration problem is to synthesize a controller such that the resulting assembly of interacting components exhibits goal-directed behavior. One possible way forward is to construct bounded reachability games from safety games.

Our vision for the future of programming is that, instead of painstakingly engineering sequences of program instructions as in the prevailing Turing tarpit, designers rigorously state their intentions and goals, and the orchestration techniques based on distributed priority synthesis construct corresponding goal-oriented assemblies of interacting components [22].

## 9 Acknowledgement

Chih-Hong Cheng thanks Christian Buckl and Alois Knoll for their early feedback on this work.

## References

[1] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM'06*, pages 3–12. IEEE, 2006. doi:10.1109/SEFM.2006.27

[2] S. Bensalem, M. Bozga, S. Graf, D. Peled, and S. Quinton. Methods for knowledge based controlling of distributed systems. In *ATVA'10*, volume 6252 of *LNCS*, pages 52–66. Springer, 2010. doi:10.1007/978-3-642-15643-4_6

[3] S. Bensalem, L. de Silva, F. Ingrand, and R. Yan. A verifiable and correct-by-construction controller for robot functional levels. *JOSER*, 1(2):1–19, 2011.

[4] B. Bonakdarpour, M. Bozga, and J. Quilbeuf. Automated distributed implementation of component-based models with priorities. In *EMSOFT'11*, 2011, pages 9–68. IEEE, 2011. doi:10.1145/2038642.2038654

[5] B. Bonakdarpour and S. Kulkarni. Sycraft: A tool for synthesizing distributed fault-tolerant programs. In *CONCUR'08*, volume 5201 of *LNCS*, pages 167–171. Springer, 2008. doi:10.1007/978-3-540-85361-9_16

[6] B. Bonakdarpour, Y. Lin, and S. Kulkarni. Automated addition of fault recovery to cyber-physical component-based models. In *EMSOFT'11*, pages 127–136. IEEE, 2011. doi:10.1145/2038642.2038663

[7] R. S. Boyer and J. S. Moore. Mjrty - a fast majority vote algorithm. In *Automated Reasoning*, volume 1 of *Automated Reasoning Series*, pages 105–117. Springer, 1991.

[8] C.-H. Cheng, S. Bensalem, R.-J. Yan, and H. Ruess. Distributed priority synthesis (full version). arXiv (CoRR abs/1112.1783), 2012.

[9] C.-H. Cheng, S. Bensalem, Y.-F. Chen, R.-J. Yan, B. Jobstmann, A. Knoll, C. Buckl, and H. Ruess. Algorithms for synthesizing priorities in component-based systems. In *ATVA'11*, LNCS. Springer, 2011. doi:10.1007/978-3-642-24372-1_12

[10] C.-H. Cheng, S. Bensalem, B. Jobstmann, R.-J. Yan, A. Knoll, and H. Ruess. Model construction and priority synthesis for simple interaction systems. In *NFM'11*, volume 6617 of *LNCS*, pages 466–471. Springer, 2011. doi:10.1007/978-3-642-20398-5_34

[11] C.-H. Cheng, B. Jobstmann, C. Buckl, and A. Knoll. On the hardness of priority synthesis. In *CIAA'11*, volume 6807 of *LNCS*. Springer, 2011. doi:10.1007/978-3-642-22256-6_11

[12] B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *LICS'05*, pages 321–330. IEEE, 2005. doi:10.1109/LICS.2005.53

[13] G. Gößler and J. Sifakis. Priority systems. In *FMCO'03*, volume 3188 of *LNCS*, pages 314–329. Springer, 2003. doi:10.1007/978-3-540-30101-1_15

[14] D. Janin. On the (high) undecidability of distributed synthesis problems. In *SOFSEM'07*, volume 4362 of *LNCS*, pages 320–329. Springer, 2007. doi:10.1007/978-3-540-69507-3_26

[15] G. Katz, D. Peled, and S. Schewe. The buck stops here: Order, chance, and coordination in distributed control. In *ATVA'11*, volume 6996 of *LNCS*, pages 422–431. Springer, 2011. doi:10.1007/978-3-642-24372-1_31

[16] G. Katz, D. Peled, and S. Schewe. Synthesis of distributed control through knowledge accumulation. In *CAV'11*, volume 6806 of *LNCS*, pages 510–525. Springer, 2011. doi:10.1007/978-3-642-22110-1_41

[17] P. Madhusudan and P. Thiagarajan. Distributed controller synthesis for local specifications. In *ICALP'01*, volume 2076 of *LNCS*, pages 396–407. Springer, 2001. doi:10.1007/3-540-48224-5_33

[18] P. Madhusudan and P. Thiagarajan. A decidable class of asynchronous distributed controllers. In *CONCUR'02*, volume 2421 of *LNCS*, pages 445–472. Springer, 2002. doi:10.1007/3-540-45694-5_11

[19] S. Mohalik and I. Walukiewicz. Distributed games. In *FSTTCS'03*, volume 2914 of *LNCS*, pages 338–351. Springer, 2003. doi:10.1007/978-3-540-24597-1_29

[20] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS'90*, volume 0, pages 746–757 vol.2. IEEE Computer Society, 1990. doi:10.1109/FSCS.1990.89597

[21] S. Schewe and B. Finkbeiner. Bounded synthesis. In *ATVA'07*, volume 4762 of *LNCS*, pages 474–488. Springer, 2007. doi:10.1007/978-3-540-75596-8_33

[22] P. Wegner. Why interaction is more powerful than algorithms. *CACM*, 40(5):80–91, 1997. doi:10.1145/253769.253801