Knowledge-based Distributed Conflict Resolution for Multiparty Interactions and Priorities

Saddek Bensalem, Marius Bozga, Jean Quilbeuf, Joseph Sifakis

Knowledge-based Distributed Conflict Resolution for Multiparty Interactions and Priorities[⋆]

Saddek Bensalem, Marius Bozga, Jean Quilbeuf, and Joseph Sifakis

UJF-Grenoble 1 / CNRS VERIMAG UMR 5104, Grenoble, F-38041, France

Abstract. Distributed decentralized implementation of systems of communicating processes raises non-trivial problems. Correct execution of multiparty interactions, subject to priority rules, requires sophisticated mechanisms for runtime conflict detection and resolution. We propose a method for detection of false conflicts which combines partial observation of the system's state and apriori knowledge extracted from invariants. We propose heuristics for determining optimal sets of observations leading to implementations with particular guarantees. We provide preliminary experimental results on an implementation of the method in the BIP framework.

Keywords: Distributed System, Priorities, Knowledge, Partial Observation, Multiparty Interactions

1 Introduction

Systems of communicating processes are a very common model for concurrent systems. Processes have their own data space and can interact by executing interactions, which are atomic synchronization operations involving a simultaneous state change of the set of the processes involved. The meaning of interactions can be specified compositionally by using operational semantics. Specifications are given in the form of rules. The premises include facts about the capabilities of individual processes to execute an action. The conclusion describes interactions, that is, transitions of the system obtained as the composition of actions executed by individual processes. In addition to interactions, operational semantics rules can be used to express priorities between interactions. These are parameterized by a priority order between interactions. They express the fact that some interaction may be executed only if interactions of higher priority are disabled. Priorities are instrumental for specifying scheduling policies [1].

The distributed implementation of systems of communicating processes raises several non trivial problems. It should be consistent with the operational semantics of multiparty interaction which assumes knowledge of the global system

[⋆] The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 248776 (PRO3D) and no 257414 (ASCENS) and from ARTEMIS JU grant agreement ARTEMIS-2009-1-100230 (SMECY) states. Furthermore, multiparty interactions should be replaced in distributed implementations by protocols based on asynchronous message passing.

The BIP component framework [2] allows the construction of composite components from a set of atomic components by using layered parameterized composition of two types of operators: 1) operators parameterized by a set of interactions which are sets of ports of the atomic components that must synchronize; 2) operators parameterized by priorities between interactions. We proposed a distributed implementation method involving a set of transformations from the initial global state model with multiparty interactions to a distributed model that can be directly implemented [3, 4]. This method has been extended to handle priorities [5]. The target model consists of components representing processes and interactions representing asynchronous message passing. Correct coordination is achieved through additional components implementing conflict resolution protocols that resolve two types of potential conflicts:

1. The first type of conflicts is symmetric conflicts between interactions. Such conflicts arise when two interactions a and b involve a common component. Since execution of interactions is atomic, execution of interaction a requires ensuring that b will not take place concurrently. Thus execution of a requires permission from some conflict resolution protocol.
2. The second type of conflicts is asymmetric conflicts between interactions related by priorities. Execution of interaction a dominated by interaction b is allowed only if some conflict resolution protocol ensures that b is not enabled.

Conflict resolution protocols are solicited for all potential statically computed conflicts according to a structural analysis of a BIP composite component. This may lead to huge implementation overhead for systems with large numbers of potentially conflicting interactions. Is it possible to reduce this overhead based on a priori knowledge of the system's dynamics and decide that some potential conflicts are not real conflicts?

We denote by $a\#b$ the fact that there is a potential conflict between interactions a and b. A false conflict state for interaction a is a state where a is enabled and all other interactions b such that $a\#b$ are disabled. In such a state, interaction a can be executed independently without arbitration by the conflict resolution protocol. States where a is in a false conflict are characterized by the predicate $FC_a = EN_a \wedge \bigwedge_{a\#b} \neg EN_b$ where EN_x is the state predicate characterizing all the states from which interaction x can be executed.

The aim of the paper is to study whether partial knowledge of the system's state is sufficient for deciding when a potential conflict is a false conflict. In that case, execution of interactions can directly take place, without arbitration and thus, reduce communication with the conflict resolution protocol for a more efficient implementation. The concept of knowledge [6] has been extensively studied for distributed systems in particular with respect to their ability to execute actions [7]. Distributed Knowledge [8] allows a set of components to "know" that an interaction is in a false conflict. We assume that each interaction a observes the states of a set of components L_a. The knowledge predicate denoted

$K_{L_a} FC_a$ characterizes the states where observing only components in L_a is sufficient to ensure that FC_a holds. In other words, it characterizes states where the distributed knowledge of the set of components L_a allows detection of false conflicts for a. We propose conditions for basic and complete implementation, respectively. In a basic implementation, it is possible to detect for each state at least one amongst the false conflicts of the global state model. In a complete implementation all false conflicts of the global state model are detected.

An interesting problem is to minimize the number of observed components, while achieving either basic or complete detection of false conflicts. To this end, we propose heuristics based on simulated annealing strategy [9].

A predicate φ is known to be true for a partial state observed on components L, that is $K_L\varphi$, if it holds in all reachable global states extending this partial state. However, computing the reachable states of a model is not always tractable. Therefore, we use invariants that over-approximate the set of reachable states. Depending on the invariant used, we obtain different results for the minimization heuristics and different performance for the implementation.

The paper is structured as follows: Section 2 provides a formal definition of the BIP global state semantics. In Section 3, we propose a definition of knowledge in the BIP context and we use it to formalize false conflict detection, and detection levels. We provide in Section 4, heuristics based on a simulated annealing strategy to minimize the number of observed components while ensuring a given detection level. In Section 5, we apply false conflict detection to implementation of priorities. We show results about both heuristics from Section 4 and an actual distributed implementation based on [3, 4] that uses false conflicts to implement priority resolution. Finally, we present related work in Section 6, concluding remarks and future work in Section 7.

2 The BIP Framework

In this section, we present BIP[2], a component framework for building systems from a set of atomic components by using two types of composition operators: Interaction and Priority.

Atomic Components. An *atomic component* B is a labelled transition system represented by a tuple (Q, P, T) where Q is a set of *control locations* or *states*, P is a set of *communication ports* and $T \subseteq Q \times P \times Q$ is a set of *transitions*.

Interactions. In order to compose a set of n atomic components $\{B_i = (Q_i, P_i, T_i)\}_{i=1}^n$, we assume that their respective sets of control locations and ports are pairwise disjoint; i.e., for any two $i \neq j$ in $\{1..n\}$, we require that $Q_i \cap Q_j = \emptyset$ and $P_i \cap P_j = \emptyset$. We define the global set $P \overset{def}{=} \bigcup_{i=1}^n P_i$ of ports. An *interaction* a is a set of ports such that a contains at most one port from each atomic component. We denote $a = \{p_i\}_{i \in I}$ with $I \subseteq \{1..n\}$ and $p_i \in P_i$. If a is an interaction, we denote by *support*(a) the set of atomic components that participate in a. This notation is extended to sets of interactions γ, that is, *support*$(\gamma) \overset{def}{=} \bigcup_{a\in\gamma}$ *support*(a).

Priorities. Given a set γ of interactions, we define a priority as a strict partial order $\pi \subseteq \gamma \times \gamma$. We write $a \pi b$ for $(a, b) \in \pi$, to express the fact that a has lower priority than b.

Composite Components. A *composite component* $\pi\gamma(B_1, \ldots, B_n)$ (or simply *component*) is defined by a set of atomic components $\{B_i = (Q_i, P_i, T_i)\}_{i=1}^n$ composed by a set of interactions γ and a priority $\pi \subseteq \gamma \times \gamma$. If π is the empty relation, then we may omit π and simply write $\gamma(B_1, \cdots, B_n)$. A global state q of $\pi\gamma(B_1, \cdots, B_n)$ is defined by a tuple of control locations $q = (q_1, \cdots, q_n)$. The behavior of $\pi\gamma(B_1, \cdots, B_n)$ is a labelled transition system $(Q, \gamma, \rightarrow_{\pi\gamma})$, where $Q = \bigotimes_{i=1}^n Q_i$ and $\rightarrow_\gamma, \rightarrow_{\pi\gamma}$ are the least set of transitions satisfying the rules:

$$\frac{\begin{array}{c} a = \{p_i\}_{i \in I} \in \gamma \\ \forall i \in I.\ (q_i, p_i, q_i') \in T_i \\ \forall i \notin I.\ q_i = q_i' \end{array}}{(q_1, \ldots, q_n) \xrightarrow{a}_\gamma (q_1', \ldots, q_n')} \ [\textsc{inter}] \qquad \frac{\begin{array}{c} q \xrightarrow{a}_\gamma q' \\ \forall a' \in \gamma.\ a \pi a' \implies q \overset{a'}{\nrightarrow}_\gamma \end{array}}{q \xrightarrow{a}_{\pi\gamma} q'} \ [\textsc{prio}]$$

Intuitively, transitions $\rightarrow_\gamma$ defined by rule [INTER] express the behaviour of the component without considering priorities. A component can execute an interaction $a \in \gamma$ iff for each port $p_i \in a$, the corresponding atomic component B_i can execute a transition labelled by p_i. If this happens, a is said to be *enabled*. Execution of a modifies atomically the state of all interacting atomic components whereas all others stay unchanged. The behavior of the component is defined by transitions $\rightarrow_{\pi\gamma}$ defined by rule [PRIO]. This rule restricts execution to interactions which are maximal with respect to the priority order. An enabled interaction a can execute only if no other one a' with higher priority is enabled.

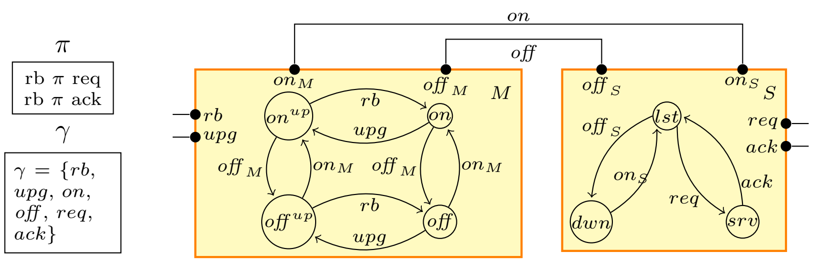

Fig. 1. An example of BIP component. Initial state is $(\mathit{off}, \mathit{dwn})$.

Example 1. A BIP component is depicted in Figure 1 using a graphical notation. It consists of two atomic components named M and S. Component S is a server, that may receive requests (req) and acknowledge them (ack). Component M is a manager that may perform upgrades (upg) and needs to reboot (rb) the server for the upgrade to be done. Interactions are represented using lines connecting the interacting ports. There are 4 unary interactions and 2 binary interactions. The component goes up through the interaction *on* and down through *off*, which are both binary interactions. Priorities $\mathit{rb}\ \pi\ \mathit{req}$ and $\mathit{rb}\ \pi\ \mathit{ack}$ are used to prevent a reboot whenever a request or an acknowledgement are possible.

Invariants and reachable states. Let $B = \pi\gamma(B_1, \cdots, B_n)$ be a component. We say that the state q is reachable from a fixed, initial state q_0 if there exist a sequence of interactions $a_1, \cdots, a_k \in \gamma$ and states $q_1, \cdots, q_k$ such that $q_0 \xrightarrow{a_1}_{\pi\gamma} q_1 \xrightarrow{a_2}_{\pi\gamma} \cdots \xrightarrow{a_k}_{\pi\gamma} q_k = q$. We denote by $\mathfrak{R}(B)$ the set of reachable states of B.

An invariant of B is a state predicate $\mathfrak{I}(q)$ satisfied by all its reachable states, that is the characteristic set of $\mathfrak{I}$ contains the set of the reachable states. For a control location $q_i \in Q_i$, we define the predicate $at(q_i)$ which is true (or equal to 1) when the atomic component B_i is at control location q_i. We are interested in two types of invariants that can be generated automatically [10], respectively:

– A *boolean invariant* is a conjunction of boolean constraints of the form $\bigvee_{j \in J} at(q_j)$. For the example of Figure 1, $at(on^{up}) \vee at(on) \vee at(dwn)$ is a boolean invariant. It characterizes a set of control locations such that at each global state, at least one location of the set is active. Such constraints are obtained using methods described in [10].
– A *linear invariant* is a conjunction of linear constraints of the form $\sum_{j \in J} k_j \, at(q_j) = k_0$, where each k_j and k_0 are integer constants. For the example of Figure 1, $at(on^{up}) + at(on) + at(dwn) = 1$ is a linear invariant. Linear invariants are obtained using algebraic methods as described in [11].

The two above categories of invariants are particularly useful for several reasons. First, they provide good approximations for the enabling/disabling conditions of interactions. This has been empirically demonstrated by the successful application of such invariants for checking deadlock-freedom of component-based systems in BIP [10, 12]. Second, the methods for computing these invariants are tractable and scale for large systems. Their computation is based on the (interaction) structure of the system and can be done incrementally [13]. In particular, it does not involve fixpoints and avoids state space exploration.

3 Knowledge-based Detection of False Conflicts

We propose a knowledge-based method for detecting *false conflicts*, that is states where an interaction is enabled and all conflicting interactions are disabled. The enabled interaction can be safely executed without any arbitration. In this section, we consider a component $B = \pi\gamma(B_1, \ldots, B_n)$ and an invariant $\mathfrak{I}$ of B.

3.1 Knowledge and Indistinguishability

The knowledge of a set of atomic components $L \subseteq \{B_1, \cdots, B_n\}$ is the set of the facts that are true by observing the states of these components. The subset L induces an equivalence relation on the global states satisfying $\mathfrak{I}$.

Definition 1 (Indistinguishability Equivalence for L). *Given L, we define the indistinguishability equivalence $\sim_L$ on global states satisfying $\mathfrak{I}$ as $q \sim_L q'$ iff $\forall B_j \in L.\ q_j = q'_j$.*

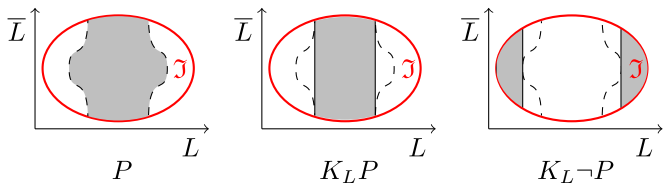

Fig. 2. Knowledge-based approximation of P for observation L, using invariant $\mathfrak{I}$.

Intuitively, two states are indistinguishable for L if their restrictions to the states of atomic components of L are identical. The equivalence classes of this relation correspond to sets of global states that can be distinguished by knowing only local states of atomic components satisfying L. Given an invariant $\mathfrak{I}$ and an arbitrary state predicate $P \implies \mathfrak{I}$, we define the predicate "L knows P" as $K_L P(q) = \mathfrak{I}(q) \wedge (\forall q' \; \mathfrak{I}(q') \wedge q' \sim_L q \implies P(q'))$.

Figure 2 illustrates $K_L P$ with respect to P and $\mathfrak{I}$. Each global state within $\mathfrak{I}$ is a point characterized by two coordinates: the projections of this state on the states of L and the states of its complement $\overline{L} = \{B_1, \ldots B_n\} \setminus L$. On the left, the gray region represents the characteristic set of P. In the middle, the gray region represents the characteristic set of "L knows P" that is the set of the global states where observation of their projection on the state space of L suffices to assert "P is true". On the right, the gray region represents the set of the states where "L knows not P" that is the set of the global states where observation on L suffices to assert "P is false".

3.2 Conflict-free Semantics

For an interaction a, we denote by EN_a the predicate that characterizes the set of global states of $\mathfrak{I}$ where a is enabled. Formally, if $a = \{p_i\}_{i \in I}$ we define $EN_a \overset{def}{=} \bigwedge_{p_i \in a} EN^i_{p_i} \wedge \mathfrak{I}$. By $EN^i_{p_i}$ we denoted the local enabling condition of the port p_i in atomic component $B_i = (Q_i, P_i, T_i)$ that is $EN^i_{p_i} \overset{def}{=} \bigvee_{(q_i, p_i, -) \in T_i} at(q_i)$.

As pointed out in the introduction, we denote by $\#$ a conflict relation between interactions. False conflicts for a correspond to the states where the predicate $FC_a = EN_a \wedge \bigwedge_{a \# b} \neg EN_b$ holds, that is states where a is enabled and all interactions conflicting with a are disabled. We consider executions of the component where only non-conflicting interactions are allowed.

Definition 2 (Conflict-free Semantics). *Given a component $B = \pi\gamma(B_1, \ldots, B_n)$ and a conflict relation $\#$, we define the* conflict-free semantics *of B as a transition system $(Q, \gamma, \rightarrow_{FC})$, where $\rightarrow_{FC}$ is the least set of transitions satisfying:*

$$\frac{a \in \gamma \qquad FC_a(q) \qquad q \xrightarrow{a}_{\pi\gamma} q'}{q \xrightarrow{a}_{FC} q'}$$

The conflict-free semantics $\rightarrow_{FC}$ is clearly included in the original semantics $\rightarrow_{\pi\gamma}$ of the component. The interest of this semantics is that it captures the set of executions that can be realized without any conflict resolution mechanism. Note that if we consider the priority conflict relation (i.e. we take $\# = \pi$), then $FC_a(q)$ is true only when $q \xrightarrow{a}_{\pi\gamma}$ and $q \overset{b}{\nrightarrow}_{\pi\gamma}$ for all b with higher priority than a. Thus in this particular case $\rightarrow_{FC} = \rightarrow_{\pi\gamma}$. The above semantic rule assumes knowledge of the global state.

3.3 Observational Conflict-free Semantics

We now propose to restrict the execution semantics presented above by using only a partial observation of the global state. We allow for each interaction a to "observe" a set of atomic components L_a including the atomic components involved in a.

Definition 3 (Observation). *Given an interaction a, an observation is a set of atomic components L_a such that support$(a) \subseteq L_a$.*

Knowledge defines a natural way to describe the false conflicts that can be detected based on an observation. That is, $K_{L_a}FC_a$ characterizes the set of the states where the observation L_a detects that a is in false conflict.

Proposition 1 (Monotoncity). *The predicate $K_{L_a}FC_a$ is monotonic with respect to L_a, i.e. $L'_a \subseteq L_a$ implies $K_{L'_a}FC_a \implies K_{L_a}FC_a$.*

Proof. First, remark that if $L'_a \subseteq L_a$, then $\{q' | q' \sim_{L_a} q\} \subseteq \{q' | q' \sim_{L'_a} q\}$. Then, $K_{L'_a}FC_a(q)$ implies that $\forall q', q' \sim_{L'_a} q,\ FC_a(q')$ and by the above remark $\forall q', q' \sim_{L_a} q,\ FC_a(q')$, that is, $K_{L_a}FC_a(q)$. □

Notice that observing the whole system, that is for $L_a = \{B_1, \ldots B_n\}$, then it is possible to detect all false conflicts, *i.e.* $K_{\{B_1,\ldots B_n\}}FC_a = FC_a$.

We define a new semantics that allows only interactions detected to be in a false conflict. In such a semantics, observation is used to decide whether a conflict-free move is allowed or not.

Definition 4 (Observational Conflict-free Semantics). *Given a component $B = \pi\gamma(B_1, \cdots, B_n)$, a conflict relation $\#$, and a set of observations $\{L_a\}_{a\in\gamma}$, we define* observational conflict-free semantics *of B as a transition system $(Q, \gamma, \rightsquigarrow)$, where $\rightsquigarrow$ is the least set of transitions satisfying:*

$$\frac{a \in \gamma \qquad K_{L_a}FC_a(q) \qquad q \xrightarrow{a}_{\pi\gamma} q'}{q \overset{a}{\rightsquigarrow} q'}$$

Again, we clearly have $\rightsquigarrow$ included in $\rightarrow_{\pi\gamma}$. However, depending on the observed atomic components, this semantics may not completely implement $\rightarrow_{FC}$. We define two criteria characterizing different levels of false conflict detection, namely basic and complete.

Definition 5 (Detection level). *A set of observations* $\{L_a\}_{a\in\gamma}$ *is* basic *iff* $\bigvee_{a\in\gamma} K_{L_a}\ FC_a = \bigvee_{a\in\gamma} FC_a$. *A set of observations is* complete *iff for each interaction* $a \in \gamma$*:* $K_{L_a} FC_a = FC_a$

Theorem 1 below, relates the detection levels and observational conflict-free semantics. Baseness ensures that observational conflicts-free semantics does not introduce deadlocks. Completeness ensures that observational conflict-free semantics corresponds exactly to the conflict-free semantics. This is particularly interesting for the case of priority conflict where the conflict-free semantics is the same as the original semantics of the component.

Theorem 1. *Let* $\pi\gamma(B_1, \cdots, B_n)$ *be a component,* $\#$ *a conflict relation and* $\{L_a\}_{a\in\gamma}$ *a set of observations. Then,* $\rightsquigarrow \subseteq \rightarrow_{FC}$ *and:*

1. *If* $\{L_a\}_{a\in\gamma}$ *is basic, then* $q \in Q$ *is a deadlock for* $\rightsquigarrow$ *only if* q *is a deadlock for* $\rightarrow_{FC}$*.*
2. *If* $\{L_a\}_{a\in\gamma}$ *is complete, then* $\rightsquigarrow = \rightarrow_{FC}$*.*

Proof. Since $K_{L_a} FC_a \implies K_{\{B_1, \cdots, B_n\}} FC_a$ (proposition 1) we have $\rightsquigarrow \subseteq \rightarrow_{FC}$.

1. By contraposition, let $q \in Q$ be a deadlock-free state for $\rightarrow_{FC}$, *i.e.* such that $\exists a \in \gamma$, $FC_a(q)$. Baseness ensures that $\bigvee_{a\in\gamma} K_{L_a} FC_a$ holds and thus $\exists b \in \gamma$ such that $K_{L_b} FC_b(q)$. Thus $q \overset{b}{\rightsquigarrow}$ and q is a deadlock-free state for $\rightsquigarrow$.

2. Assume that $q \xrightarrow{a}_{FC} q'$. Then $FC_a(q)$ and completeness ensures $K_{L_a} FC_a(q)$. Thus $q \overset{a}{\rightsquigarrow} q'$. □

These results characterize to what extent conflict-free semantics, can be captured through partial observation. They can be used in a distributed implementation where the process responsible for executing interaction a can only see the states of atomic components in L_a. However, adding observation increases communication between processes, which may slow down execution. Therefore, we propose in the next section heuristics to minimize the number of observed atomic components, yet ensuring the required detection level.

4 Heuristics for Minimizing Observation

Given a BIP component and a conflict relation, we want to minimize the number of observed atomic components while ensuring either baseness or completeness. For practical reasons, we consider that a single process may coordinate the execution of several interactions. We call such a process an *engine*. All interactions managed by the same engine share a common set of observed atomic components. We consider that the mapping of interactions γ into engines is defined by an arbitrary, fixed partition of γ as $\bigcup_{1\leq j\leq m} \gamma_j$. An engine E_j is used to execute interactions in γ_j while observing a set of atomic components L_j. With these notations, minimizing observation means minimizing the sum $\sum_{j=1}^{m} |L_j|$.

We propose a solution to the minimizing observation problem based on simulated annealing [9]. A pseudo-code for the heuristic is shown in Algorithm 1. This heuristic allows on to search for optimal solutions to arbitrary cost optimization

Algorithm 1 Pseudo-code of Simulated Annealing

```
Input: An initial solution $init$, a cost function, an alter function.
Output: A solution with a minimized cost.
 1: $sol := init$
 2: $T := T_{max}$
 3: while $T > T_{min}$ do
 4:     $sol' :=$ alter($sol$)
 5:     $\Delta :=$ cost($sol'$) - cost($sol$)
 6:     if $\Delta < 0$ or random() $< e^{\frac{-\Delta}{T}}$ then
 7:         $sol := sol'$
 8:     end if
 9:     $T := 0.99 \times T$
10: end while
11: return $sol$
```

problems. The search through the solution space is controlled by a *temperature* parameter T. At every iteration, temperature decreases slowly (line 9) and the current solution moves into a new, nearby solution still ensuring either baseness or completeness (line 4). If the new solution is better (i.e. observes fewer components), then it becomes the current solution. Otherwise, it may be accepted with a probability that decreases when (1) the temperature decreases or (2) the extra cost of the new solution increases (line 6). The idea is to temporarily allow a bad solution whose neighbors may be better than the current one. By the end of the process, the temperature is low, which prevents bad solutions from being accepted. Now, we provide initial solutions *init* as well as `alter` and `cost` functions that are used to ensure either completeness or baseness.

Ensuring Completeness. According to Definition 5, checking for completeness is performed interaction by interaction, Therefore, minimizing observation can be carried out independently for each engine. Given the set of interactions γ_j we are seeking for a minimal set of atomic components L_j, whose observation ensures complete detection of false conflicts for all interactions in γ_j.

Algorithm 2 Function `alter` for ensuring complete detection of false conflicts

```
Input: A BIP component $B$, a subset of interactions $\gamma_j$, a conflict relation $\#$ and a solution $L_j$.
Output: A solution $L'_j$ that is a neighbor of $L_j$.
 1: $L'_j := L_j$
 2: choose $B_i$ in $L'_j \setminus support(\gamma_j)$
 3: $L'_j := L'_j \setminus \{B_i\}$                                   //perturbation
 4: while not complete($L'_j, \gamma_j$) do
 5:     choose $B_i$ in $\{B_1, \ldots, B_n\} \setminus L'_j$
 6:     $L'_j := L'_j \cup \{B_i\}$                                   //completion
 7: end while
 8: choose $B'_i$ in $L'_j \setminus support(\gamma_j)$
 9: while complete($L'_j \setminus \{B'_i\}, \gamma_j$) do
10:     $L'_j := L'_j \setminus \{B'_i\}$                              //reduction
11:     choose $B'_i$ in $L'_j \setminus support(\gamma_j)$
12: end while
13: return $L'_j$
```

The initial solution is obtained by taking the set of atomic components involved in interactions conflicting with those of γ_j, that is $init_j = \bigcup_{a \in \gamma_j} (support(a) \cup \bigcup_{a \# b} support(b))$. At each iteration of the simulated annealing, a new solution is computed using the `alter` function shown in Algorithm 2. First, one atomic component is removed from the solution (perturbation), possibly breaking completeness. Then, new atomic components are added randomly until the solution ensures complete detection again (completion). Finally, atomic components are removed randomly (reduction).

After completion and during reduction steps, completeness is ensured by checking the condition $\texttt{complete}(L_j, \gamma_j) \equiv \bigwedge_{a \in \gamma_j} \left(FC_a = K_{L_j} FC_a\right)$. On termination, this ensures that the solution returned by the heuristic is complete.

The cost of the solution is obtained by counting the number of atomic components additionally observed by each engine. That is, for an engine E_j the cost of solution L_j is $\texttt{cost}(L_j) = |L_j \setminus support(\gamma_j)|$.

Ensuring Baseness. Baseness is achieved if for every state which contains false conflicts, at least one engine detects one of them. Baseness is a global property that can be ensured by cooperation between engines. On one hand, allowing an engine E_j to observe additional atomic components may extend the set of false conflicts detected by E_j. On the other hand, reducing observation of E_j, while restricting the set of false conflicts detected, might not necessarily break the baseness. Therefore, the solution $L_1, \cdots, L_m$ to the minimizing observation ensuring baseness cannot be built independently for each engine. Given a partition $\gamma_1, \cdots, \gamma_m$ of the interactions, we build a tuple of sets of atomic components $L = (L_1, \cdots, L_m)$ ensuring baseness.

Algorithm 3 Function `alter` for ensuring basic detection of false conflicts

```
Input: A BIP component $B$, a partition of interactions $\gamma = \bigcup_{1 \leq j \leq m} \gamma_j$, a conflict relation $\#$ and
    a solution $L = (L_1, \ldots, L_m)$,
Output: A solution $L'$ that is a neighbor of $L$.
 1: $L' := L$
 2: choose $k$ in $[\![1, m]\!]$ and $B_i$ in $L'_k \setminus support(\gamma_k)$
 3: $L'_k := L'_k \setminus \{B_i\}$                                        //perturbation
 4: while not basic$(L', \{\gamma_1, \ldots, \gamma_m\})$ do
 5:     choose $k$ in $[\![1, m]\!]$ and $B_i$ in $\{B_1, \ldots, B_n\} \setminus L'_k$
 6:     $L'_k := L'_k \cup \{B_i\}$                                        //completion
 7: end while
 8: choose $k$ in $[\![1, m]\!]$ and $B'_i$ in $L'_k \setminus support(\gamma_k)$
 9: while basic$((L'_1, \ldots, L'_k \setminus \{B'_i\}, \ldots, L'_m), \{\gamma_1, \ldots, \gamma_m\})$ do
10:     $L'_k := L'_k \setminus \{B'_i\}$                                      //reduction
11:     choose $k$ in $[\![1, m]\!]$ and $B'_i$ in $L'_k \setminus support(\gamma_k)$
12: end while
13: return $L'$
```

The initial solution assumes that each engine E_j observes all atomic components involved in interactions conflicting with those of γ_j, that is $init = (init_1, \ldots, init_m)$. As for completeness, the `alter` function for baseness presented in Algorithm 3 computes a new solution based on the same three steps (perturbation,completion,reduction) being performed on a tuple of sets of observed atomic components, instead of a single set.

After completion and during reduction steps, baseness is ensured by the condition $\texttt{basic}\,(L, \{\gamma_1, \ldots, \gamma_m\}) \equiv \left(\bigvee_{a\in\gamma} FC_a = \bigvee_{j=1}^{m} \bigvee_{a\in\gamma_j} K_{L_j} FC_a\right)$. This guarantees that the returned solution is basic.

Here the cost of the solution is the sum of the number of additional atomic components observed by each engine. Thus, we define the `cost` function as $\texttt{cost}(L) = \sum_{j=1}^{m} |L_j \setminus \mathit{support}(\gamma_j)|$.

5 Implementation and Experiments

In this section, we provide experiments related to detection of false priority conflicts. We apply our simulated annealing heuristics to compute minimal basic or complete solutions for two examples. Then we provide performance gains for the corresponding distributed implementations.

5.1 Dining Philosophers

We consider a variation of the dining philosophers problem, denoted by Philo$N$ where N is the number of philosophers. A fragment of this composite component is presented in Figure 3. In this component, an "eat" interaction eat_i involves a philosopher and the two adjacent forks. After eating, philosopher P_i cleans the forks one by one ($\mathit{cleanleft}_i$ then $\mathit{cleanright}_i$). We consider that each eat_i interaction has higher priority than any $\mathit{cleanleft}_j$ or $\mathit{cleanright}_j$ interaction. We evaluate two different partitions. In Partition 1, there is one engine E_i for every eat_i interaction and one engine C_i for every pair $\mathit{cleanright}_{i-1}$, $\mathit{cleanleft}_i$. Only the latter deals with low priority interactions and therefore may need to observe additional atomic components. Partition 2 is coarser. One engine E_i manages all interactions involving philosopher P_{2i} or P_{2i+1}, for $0 \leq i < \lceil N/2 \rceil$. Thus, for N philosophers, there are $\lceil N/2 \rceil$ engines.

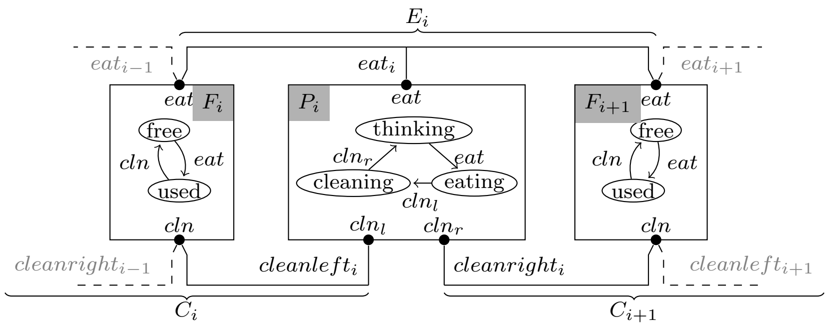

Fig. 3. Fragment of the dining philosopher component. Braces illustrates Partition 1.

Computing complete solutions is done independently for each engine. Table 1 shows results of engine C_0 for Partition 1 and engine E_0 for Partition 2. The total

Eng.	**Comp./Part.**	*Size*	*true*	*BI*	*LI*	*optimal*
C_0	Philo3 / 1	6	3	3	1	1
	Philo4 / 1	8	5	5	2	2
	Philo5 / 1	10	7	7	3	3
	Philo10 / 1	20	17	17	8	8
	Philo20 / 1	40	37	37	18	18
	Philo100 / 1	200	197	197	108	98
E_0	Philo3 / 2	6	1	1	0	0
	Philo4 / 2	8	3	3	1	1
	Philo5 / 2	10	5	5	2	2
	Philo10 / 2	20	15	15	7	7
	Philo20 / 2	40	35	35	18	17
	Philo100 / 2	200	195	195	106	97

Table 1. Minimal observation for completeness.

Comp./Part.	*Size*	*true*	*BI*	*LI*
Philo3 / 1	6	9	9	0
Philo4 / 1	8	20	20	4
Philo5 / 1	10	35	35	6
Philo10 / 1	20	170	170	23
Philo3 / 2	6	4	4	0
Philo4 / 2	8	6	6	1
Philo5 / 2	10	17	17	3
Philo10 / 2	20	75	75	14

Table 2. Minimal observation for baseness.

number of atomic components in the composite component is indicated in Column *Size*. Columns *true*, *BI* and *LI* provide the cost of the solutions obtained when using respectively *true*, the boolean invariant and the linear invariant as invariant $\mathfrak{I}$. The column *optimal* indicates the cost of an optimal solution.

$$\forall i \in \{0,1,2\}\ (at(F_i.free) \vee at(F_i.used)) \tag{1}$$
$$\wedge \quad \forall i \in \{0,1,2\}\ (at(P_i.thinking) \vee at(P_i.eating) \vee at(P_i.cleaning)) \tag{2}$$
$$\wedge \quad (at(P_1.eating) \vee at(P_0.eating) \vee at(P_0.cleaning) \vee at(F_1.free)) \tag{3}$$
$$\wedge \quad (at(P_2.eating) \vee at(P_1.eating) \vee at(P_1.cleaning) \vee at(F_2.free)) \tag{4}$$
$$\wedge \quad (at(P_0.thinking) \vee at(F_0.used) \vee at(P_0.cleaning) \vee at(P_2.thinking)) \tag{5}$$
$$\wedge \quad (at(P_0.thinking) \vee at(F_1.used) \vee at(P_1.cleaning) \vee at(P_1.thinking)) \tag{6}$$
$$\wedge \quad (at(P_2.cleaning) \vee at(F_0.free) \vee at(P_2.eating) \vee at(P_0.eating)) \tag{7}$$
$$\wedge \quad (at(F_1.free) \vee at(F_2.free) \vee at(F_0.free) \vee at(P_1.eating) \vee at(P_2.eating) \vee at(P_0.eating)) \tag{8}$$
$$\wedge \quad (at(F_2.used) \vee at(P_2.cleaning) \vee at(P_1.thinking) \vee at(P_2.thinking)) \tag{9}$$
$$\wedge \quad (at(F_2.used) \vee at(P_2.cleaning) \vee at(P_1.thinking) \vee at(F_0.free) \vee at(P_0.eating)) \tag{10}$$
$$\wedge \quad (at(F_1.free) \vee at(P_1.eating) \vee at(F_0.used) \vee at(P_0.cleaning) \vee at(P_2.thinking)) \tag{11}$$
$$\wedge \quad (at(P_0.thinking) \vee at(F_2.free) \vee at(F_1.used) \vee at(P_2.eating) \vee at(P_1.cleaning)) \tag{12}$$

Fig. 4. Boolean invariant for the Dining Philosophers example with $N = 3$.

Here, the linear invariant gives better results than the boolean invariant, which does not give enough information about the system to reduce observation comparatively to the *true* invariant. For $N = 3$, we provide the boolean and linear invariants respectively in Figures 4 and 5. In this case, the linear constraint (15) in linear invariant ensures that interaction $cleanleft_0$ and interaction eat_1 cannot be enabled concurrently, otherwise, control locations $P_0.eating$ and $F_1.free$ would be active and the sum in constraint (15) would be equal to 2.

$$\begin{array}{rlr}
 & (at(P_0.thinking) + at(P_0.eating) + at(P_0.cleaning) = 1) & (13)\\
\wedge & \forall i \in \{0,1,2\} \quad (at(F_i.free) + at(F_i.used) = 1) & (14)\\
\wedge & (at(P_1.eating) + at(P_0.eating) + at(P_0.cleaning) + at(F_1.free) = 1) & (15)\\
\wedge & (at(P_1.thinking) + at(P_0.thinking) + at(F_1.used) + at(P_1.cleaning) = 2) & (16)\\
\wedge & (at(P_2.eating) + at(P_0.thinking) + at(F_1.used) + at(P_1.cleaning) + at(F_2.free) = 1) & (17)\\
\wedge & (at(P_2.cleaning) + 2 * at(P_0.eating) + at(P_0.cleaning) - at(F_1.used) & \\
 & - at(P_1.cleaning) + at(F_2.used) - at(F_0.used) = 0) & (18)\\
\wedge & (at(P_2.thinking) - at(P_0.eating) + at(F_0.used) = 1) & (19)
\end{array}$$

Fig. 5. Linear invariant for the Dining Philosophers example with $N = 3$.

Thus, the priority $\mathit{cleanleft}_0 \, \pi \, \mathit{eat}_1$ never forbids execution of $\mathit{cleanleft}_0$. A related boolean constraint, that is constraint (3) of boolean invariant guarantees that at least one of these locations is active. However, this constraint is not strong enough to discard the case where two of them are active.

The results for computing basic solutions are presented in Table 2. The column *Size* contains the total number of atomic components in the composite component. The columns *true*, *BI* and *LI* contains respectively the cost of the solutions obtained when using respectively *true*, the boolean invariant and the linear invariant. For Philo3, baseness is achieved when each engine observes only the components involved in the interactions it handles (i.e. no additional atomic component), therefore the cost is 0.

Performance Evaluation. We used the tool-chain described in [3, 4] to generate automatically distributed code from the component. The generated code consists of a set of C++ programs communicating through Unix sockets. We generate one program for each atomic component, one program for each engine and one program for conflict resolution between engines (CRP). We executed these programs in a distributed setting (on a UltraSparcT1 with 24 parallel threads) during 60 seconds and counted the number of “eat” interactions.

Performance for Partition 1 (resp. Partition 2) is depicted in Figure 6 (resp. 7). We do not show performance for the boolean invariant because it falls back to observing all components, as for the *true* invariant. Since Partition 2 is coarser than Partition 1, it allows less parallelism as shown by comparing performance of execution without priority. Priority limits the number of executions as it enforces a particular scheduling policy and reduces parallelism. For both partitions, the fastest prioritized implementation is the complete one obtained by using the linear invariant. When we observe all involved atomic components (i.e. the invariant is *true*), performance is worse because the lack of knowledge about the reachable states entails more synchronization overhead. Finally, basic solutions are slow because, while restricting the communication, they also restrict the parallelism.

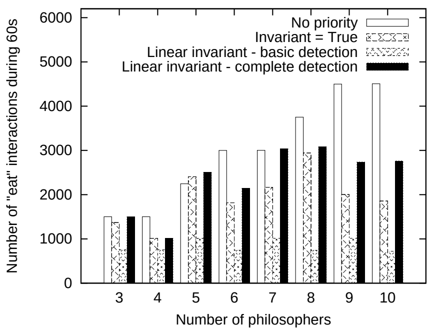

Fig. 6. Performance for different detection levels, using Partition 1.

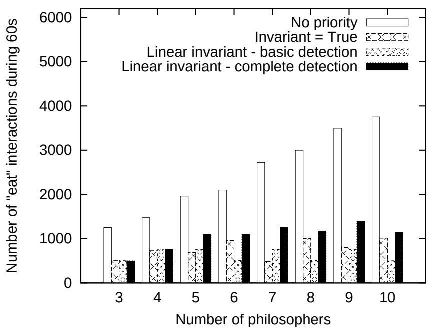

Fig. 7. Performance for different detection levels, using Partition 2.

5.2 Jukebox

The second example is a jukebox depicted in Figure 8. It represents a system, where a set of readers $R_1 \dots R_4$ access data located on disks D_1, D_2, D_3. Readers may need to access any disk. Access to disks is managed by jukeboxes J_1, J_2 that can load any disk to make it available to the connected readers. Interactions $load_{i,k}$ and $unload_{i,k}$ allows to load and unload the disk D_i in the jukebox J_k. Each reader R_j is connected to a jukebox through the $read_j$ interaction. Once a jukebox has loaded a disk, it can either take part in a "read" or "unload" interaction. Each jukebox repeatedly loads all 3 disks in a random order.

If unload interactions are always chosen immediately after a disk is loaded, then readers may never be able to read data. Therefore, we add the priority $unload_{i,k} \, \pi \, read_j$, for all i, j, k. This ensures that "read" interactions will take place before corresponding disks are unloaded. Furthermore, we assume that readers connected to J_1 need more often disk 1 and that readers connected to J_2 need more often disk 2. Therefore, loading these disks in the corresponding jukeboxes is assigned higher priority: $load_{i,1} \, \pi \, load_{1,1}$ for $i \in \{2, 3\}$ and $load_{i,2} \, \pi \, load_{2,2}$ for $i \in \{1, 3\}$.

Interaction	$true$	BI(basic)	BI(complete)	LI(basic)	LI(complete)
$unload_{i,k}$	5	3($k = 1$) or 5($k = 2$)	5	2	2
$load_{i,k}$	1	0	1	0	1

Table 3. Minimal observation cost to ensure baseness or completeness.

We use a partition assigning one engine per interaction. Results of the simulated annealing heuristic are presented in Table 3. Engines handling a "read" interaction do not need to observe additional atomic components since there is no interaction with higher priority. The boolean invariant allows removing some observed atomic components, in the basic solution. As for Philo$N$ components,

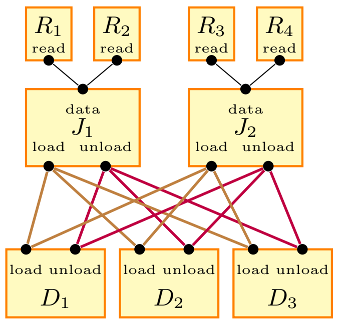

Fig. 8. Jukebox composite component.

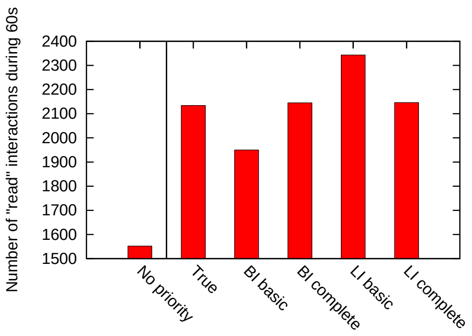

Fig. 9. Performance of the jukebox component for unprioritized and prioritized executions with different invariants/detection levels.

the linear invariant is stronger than the boolean invariant. Therefore, the same level of detection is achieved with less observed atomic components.

Performance Evaluation. We generate C++ code, as for the previous example. We count the number of "read" interactions that take place during 60 seconds of execution, for different settings. In Figure 9, we provide results for a version of the component without priorities, as well as results for the prioritized component for the trivial invariant *true*, the boolean invariant (BI) or the linear invariant (LI). For boolean and linear invariants, we provide performance for both complete and basic implementations.

Notice that adding priority increases the number of "read" interactions executed in 60 seconds. This is due to the fact that a disk is unloaded only if no read is possible, that is only when unload is necessary to progress. Solutions obtained for the boolean invariant require more observation than the ones obtained for the linear invariant, therefore corresponding implementations are slower. More interesting, the best performance is obtained for basic solutions. In that particular case, fewer atomic components are observed which allows more parallelism in the composite component. This parallelism compensates the fact that the detection of false conflicts is not complete.

6 Related Work

Distributed resource conflict resolution boils down to solving the *committee coordination problem* [14], where a set of professors organize themselves in different committees, a meeting requires the presence of all professors to take place and two committees that have a professor in common cannot meet simultaneously. Different solutions have been provided, using managers [14–16], a circulating token [17], or a randomized algorithm without managers [18]. Solutions using managers typically rely on a conflict resolution protocol, such as a solution to the dining philosophers problem [19].

Similarly, implementation of priorities needs resolution of asymmetric conflicts. This can be achieved by direct observation as in [20] or [5] where managers observe higher priority interactions to ensure their disabledness. Knowledge is often used to drive action execution in distributed systems. Halpern and Moses [8] defined a logic to reason about the knowledge of system processes. Knowledge is used to control distributed discrete event systems [21] and build distributed controllers for executing multiparty interactions with priorities [22].

In most papers, computing knowledge requires exact computation of reachable states [21, 22, 8]. Our method overcomes this difficulty by using invariants which are over-approximations of the reachability set. Another common assumption is that the partial state observed by a manager is limited to a neighborhood determined by the architecture of the system [22]. We propose a framework where observation can be adjusted for achieving a certain detection level.

7 Conclusion

Implementing multiparty interactions scheduled by using priorities requires efficient conflict resolution techniques. Most implementations do not distinguish between real and false conflicts to reduce overhead due to conflict resolution.

Dynamic knowledge-based computation of false conflicts based on invariants allows more efficient implementations. We provided simple criteria to define the correctness of the obtained implementation. Baseness ensures preservation of deadlock-freedom and completeness ensures equivalence with the fully-observed model. Finally, the proposed heuristics allow minimization of the number of components to observe and enhanced performance. Heuristics have been applied to non-trivial examples, where the optimal is known, and gave satisfactory results. These have been used for distributed implementation. Experiments show significant performance improvement. However, depending on the model, best performance is achieved either for basic or complete observation.

Future work includes several directions. First, we plan to study in depth how choices of detection levels affect performance of the obtained implementation. We can also consider intermediate levels between basic and complete observation. Such intermediate levels could, for instance, ensure complete detection of false conflicts for some interactions and avoid introduction of deadlocks for the others.

Another improvement is to use static analysis techniques in order to take into account parallelism. These techniques allow automatically computing a partition of the interactions that does not reduce the degree of parallelism by grouping possibly concurrent interactions. The allowed degree of parallelism can also be used to measure the utility of an additional observation, i.e. how observing an additional component can increase parallelism in the obtained implementation.

References

1. Gössler, G., Sifakis, J.: Priority systems. In: Formal Methods for Components and Objects. Volume 3188 of LNCS. Springer Berlin / Heidelberg (2004) 314–329

2. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Software Engineering and Formal Methods (SEFM). (2006) 3–12
3. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: From high-level component-based models to distributed implementations. In: EMSOFT. (2010)
4. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. Distributed Computing (to appear)
5. Bonakdarpour, B., Bozga, M., Quilbeuf, J.: Automated distributed implementation of component-based models with priorities. In: EMSOFT. (2011) 59–68
6. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press (1995)
7. Halpern, J.Y., Fagin, R.: Modelling knowledge and action in distributed systems. Distributed Computing **3** (1988) 159–179
8. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. J. ACM **37** (July 1990) 549–587
9. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**(4598) (1983) 671–680
10. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.H.: Compositional verification for component-based systems and application. In: ATVA, Berlin, Heidelberg (2008)
11. Krckeberg, F., Jaxy, M.: Mathematical methods for calculating invariants in petri nets. In: Advances in Petri Nets 1987. Volume 266 of LNCS. Springer Berlin / Heidelberg (1987) 104–131
12. Bensalem, S., Bozga, M., Nguyen, T.H., Sifakis, J.: D-finder: A tool for compositional deadlock detection and verification. In: Computer Aided Verification. Volume 5643 of LNCS. (2009) 614–619
13. Bensalem, S., Bozga, M., Legay, A., Nguyen, T.H., Sifakis, J., Yan, R.: Incremental component-based construction and verification using invariants. In: Formal Methods in Computer-Aided Design (FMCAD). (oct. 2010) 257 –256
14. Chandy, K.M., Misra, J.: Parallel program design: a foundation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1988)
15. Bagrodia, R.: Process synchronization: Design and performance evaluation of distributed algorithms. IEEE Transactions on Software Engineering (TSE) **15**(9) (1989) 1053–1065
16. Pérez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurrency and Computation: Practice and Experience **16**(12) (2004) 1173–1206
17. Kumar, D.: An implementation of n-party synchronization using tokens. In: ICDCS. (1990) 320–327
18. Joung, Y.J., Smolka, S.A.: Strong interaction fairness via randomization. IEEE Trans. Parallel Distrib. Syst. **9**(2) (1998) 137–149
19. Chandy, K.M., Misra, J.: The drinking philosophers problem. ACM Transactions on Programming Languages and Systems (TOPLAS) **6**(4) (1984) 632–646
20. Ben-Hafaiedh, I., Graf, S., Quinton, S.: Building distributed controllers for systems with priorities. Journal of Logic and Algebraic Programming **80** (2011) 194 – 218
21. Ricker, S., Rudie, K.: Know means no: Incorporating knowledge into discrete-event control systems. Automatic Control, IEEE Transactions on **45**(9) (sep 2000) 1656 –1668
22. Bensalem, S., Bozga, M., Graf, S., Peled, D., Quinton, S.: Methods for knowledge based controlling of distributed systems. In: Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Proceedings. Volume 6252., Springer (September 2010) 52–66