Synthesizing Distributed Scheduling Implementation for Probabilistic Component-based Systems

Saddek Bensalem, Axel Legay, Ayoub Nouri, Doron Peled

Synthesizing Distributed Scheduling Implementation for Probabilistic Component-based Systems Technical Report

Saddek Bensalem*, Axel Legay†, Ayoub Nouri*, and Doron Peled‡
*UJF-Grenoble 1 / CNRS VERIMAG UMR 5104, Grenoble, F-38041, France
†INRIA/IRISA, Rennes, France
‡Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

***Abstract*—Developing concurrent systems typically involves a lengthy debugging period, due to the huge number of possible intricate behaviors. Using a high level description formalism at the intermediate level between the specification and the code can vastly help reduce the cost of this process, and the existence of remaining bugs in the deployed code. Verification is much more affordable at this level. An automatic translation of component based systems into running code, which preserves the temporal properties of the design, helps synthesizing reliable code. We provide here a transformation from a high level description formalism of component based system with probabilistic choices into running code. This transformation involves synchronization using shared variables. This synchronization is component-based rather than interaction-based, because of the need to guarantee a stable view for a component that performs probabilistic choice. We provide the synchronization algorithm and report on the implementation.**

I. Introduction

Computer systems are initially developed as single entities working on a single core architecture. The advances in hardware and system design have drastically changed this situation. Indeed, to increase performances and offer new services, systems are now fully exploiting new technologies such as multi-core or grid and cloud computing. This exacerbate the challenge of coordinating for the multiple subsystems running concurrently to achieve a series of local/global objectives. Moreover, economical constraints and separation of concerns lead to a new way of developing huge systems. The software design process can also consists of breaking the original design into small entities, each of them being potentially developed by a separate party that may not want to completely reveal the entire implementation.

This paper presents an automatic transformation from a high level description formalism for a component based systems, which include concurrency and probabilistic choices, into a physical implementation. This is a continuation to the models proposed in, e.g., [12], [13], [18], [21], [23], [24], [32]. Our solution uses a limited amount of information from the interface of the components [2], [8]. Our transformation is a direct progress from the recent modeling of concurrency and probabilities in Petri nets in [21], that suggests a principal algorithm that uses the Petri net objects, specifically, the places, as semaphores to achieve a correct synchronization. In contrast to that theoretical algorithm, we suggest here a realistic transformation based on shared variables for synchronization. We assume no additional scheduling mechanism or algorithm, nor we assume large atomic operations besides standard ones (read, write and swap-and-compare). Our transformation preserves the correctness of the high level design.

Research on implementing and verifying concurrent component-based systems extends back to Milner's CCS [28] and Hoare's CSP [19]. The I/O automata [25] and latter interface theories [14] permits reasoning on subsystems interacting through input/output mechanisms. Work on BIP [8] proposes very flexible coordination languages that classify sets of processes that may interact together [11]. By identifying independent interactions, one can eventually map sets of subsystems to different processes. The implementation reveals the difficulty in achieving concurrent implementation. Using efficient synchronizing algorithms such as α-core [29], one can then obtain an efficient mechanism to handle concurrency with interactions. As observed in [15], the latter is difficult for the case of probabilistic systems.

In the design phase, one often has to model subcomponents by pure probabilistic systems rather than by transition systems [34]. This view allows capturing faults and uncertainties in the hardware design. Mixing the non-determinism of process selection with the probabilistic information of individual components leads to very expressive but complex models, among which are Markov Decision Processes, Probabilistic Petri Nets, and Probabilistic automata. Clearly, adding probabilities in the presence of nondeterminism is challenging, especially if one wants to preserve full concurrency [1], [5], [6], [10]. One solution that was proposed is to parameterize the design to bias the choice of a given process [13]. A second approach suggests to exploits real-time delays and properties of exponential distributions rather than parameters [36].

Another, more robust approach, which is particularly em-

The fourth author is supported by ISF grant 126/12 "Efficient Synthesis of Control for Concurrent Systems"

phasized in the probabilistic I/O automata work, allow both nondeterministic and probabilistic choice [32]. Switched I/O automata [12] propose a solution to combination of concurrency and probabilistic choices in two steps. The first step establishes a competition between the components on making the probabilistic decision. In the second step, a probabilistic selection is made by the selected component. This approach suffers from the fact that it limits the concurrency of the system by allowing only one component to be active at each state.

The difficulty in deciding which component will make the probabilistic choice is the one of giving to each of the components a stable and coherent view of the entire design. This needs to be done without while preserving as much as possible the concurrent behavior of the system, and while revealing too much information about the global design. Recently, Peled and Katoen proposed a new scheduling algorithm to avoid the confusion problem for Probabilistic Petri Nets [21]. This algorithm, which is based on shared locks, is fully concurrent. It is related to the idea of distributed schedulers [18], but does not force probabilistic selection of processes, hence maintaining probabilistic choices at the level of the process.

In this paper, we make an additional step forward by proposing an algorithm that deals with concurrent probabilistic systems that makes *realistic* local assumptions on the individual components. The algorithm is intended for a direct implementation on concurrent hardware, provided standard shared variables operations; no need for special synchronization or scheduling services from the operating system.

The contributions of this paper are the following:

1) An automatic transformation from a high level description of component based system that include both *concurrency* and *probabilistic choices* into an implementation based on shared locks. The implementation does not assume or use any high level scheduling mechanism, e.g., the α-core algorithm [29], [22] used to provide synchronization for BIP components.
2) We refute the *theoretical principal implementation* presented in [21] into a completely *realistic and practical* one. Our transformation does not assume atomicity on observing the local view of a component, nor on executing a transition. In developing the algorithm, we realized where the algorithm in [21] needs to be refined, and further refuted and optimized in order to obtain the efficient implementation reported here.
3) We report on a concrete and realistic implementation of the algorithm and provide a case study.

II. Components Based Systems

We first propose a component-based design formalism.

Definition 1: A *component based system* $\mathcal{A}$ contains a set of components $\mathcal{A}_i = \langle \Sigma_i, S_i, \delta_i, s_i^0 \rangle$ for $i \in [1..n]$ such that:

- Σ_i is a finite set of *transitions*,
- S_i is a finite set of *local* states.
- $\delta_i \subseteq S_i \times \Sigma_i \times S_i$ is a deterministic partial transition system, i.e., if $\delta_i(s, \alpha, s')$ and $\delta_i(s, \alpha, s'')$ then $s' = s''$. Also, we assume that if $\delta_i(s, \alpha, s')$ and $\delta_i(s, \beta, s'')$, then $s' \neq s''$.
- $s_i^0 \in S_i$ is the initial state of $\mathcal{A}_i$.

A transition α is called *shared* if $\alpha \in \Sigma_i \cap \Sigma_j$ for $i \neq j$.

Intuitively, a component is a finite automaton, acting as a concurrent process, where it can execute a transition if it owns it exclusively, and needs to coordinate with other components in order to execute transitions that are shared with other components.

Definition 2: The *global representation* of $\mathcal{A}$, also as a finite state automaton $\langle \Sigma, S, \delta, g^0 \rangle$ as follows:

- $\Sigma = \cup_{i=1..n} \Sigma_i$ is the set of transitions.
- $G = \Pi_{i=1..n} S_i$ is the set of *global* states. Denote a state $g \in S$ also as a tuple of components $\langle s_1, s_2, \ldots, s_n \rangle$ with $s_i \in S_i$.
- $\delta \subseteq G \times \Sigma \times G$ such that $\delta(\langle s_1, s_2, \ldots, s_n \rangle, \alpha, \langle s'_1, s'_2, \ldots s'_n \rangle)$, where
  1) For at least one component A_i, $\delta_i(s_i, \alpha, s'_i)$,
  2) For each component $\mathcal{A}_i$, if $\alpha \notin \Sigma_i$ then $s_i = s'_i$, otherwise, $\delta_i(s_i, \alpha, s'_i)$.
- $g^0 = \langle s_1^0, s_2^0, \ldots s_n^0 \rangle$.

The global representation is basically the synchronization of the component automata on shared transitions. A (finite or infinite) execution for $\mathcal{A}$ is a maximal alternating sequence $g_0 \alpha_0 g_1 \alpha_1 \ldots$ of global states from G and transitions from Σ such that g_0 is the initial state g^0 of $\mathcal{A}$, and $\delta(g_i, \alpha_i, g_{i+1})$. The sequence is finite if from its last state, no transition is enabled.

For $s \in S_i$, let ${}^\bullet s = \{\alpha | \exists s' \in S_i, \delta_i(s', \alpha, s)\}$ be the set of input transitions to s, and $s^\bullet = \{\alpha | \exists s' \in S_i, \delta_i(s, \alpha, s')\}$ be the set of output transitions from s. A transition α is *enabled* from a global state $\langle s_1, s_2, \ldots, s_n \rangle \in S$ of a components system if $\alpha \in s_i{}^\bullet$ for each i such that $\alpha \in \Sigma_i$.

Definition 3: Any two transitions $\{\alpha, \beta\} \subseteq {}^\bullet s \cup s^\bullet$ for some local state s of some component are said to be *dependent*. Let $D \subseteq \Sigma \times \Sigma$ be the reflexive *dependence* relation. Then its complement $I = (\Sigma \times \Sigma) \setminus D$ is the *independence* relation. We further identify a pair of interdependent transitions $(\alpha, \beta) \in D$ to be *in conflict* (or conflicting) if $\{\alpha, \beta\} \subseteq s^\bullet$ for some local state s for some component $\mathcal{A}_i$, and *subsequent* if $\alpha \in {}^\bullet s$ and $\beta \in s^\bullet$.

The notion of a *confusion*, formally defined below, come from Petri nets [16]. Intuitively, it describes a situation where the execution of a transition β, independently of another transition α, would change the alternative choices to executing β.

Definition 4: A pair of independent transitions $(\alpha, \beta) \in I$ is a *confusion* if there exists $\gamma \in \Sigma$, two different components $\mathcal{A}_i$, $\mathcal{A}_j$ such that $\alpha \in \Sigma_i$, $\beta \in \Sigma_j$, $s \in S_i$ be a local

state of $\mathcal{A}_i$, and $s' \in S_j$ be a local state of $\mathcal{A}_j$, such that either

- $\{\alpha, \gamma\} \subseteq s^\bullet$ and $\{\gamma, \beta\} \subseteq s'^\bullet$, then the confusion (α, β) is *symmetric*, see Figure 1, or
- $\{\alpha, \gamma\} \subseteq s^\bullet$, $\gamma \in s'^\bullet$, and $\beta \in {}^\bullet s'$. then the confusion (α, β) is *asymmetric*, see Figure 2.

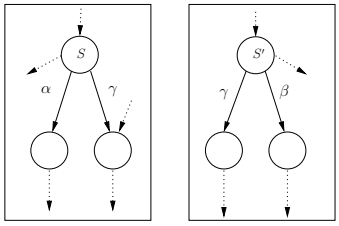

Figure 1: Symmetric confusion (α, β)

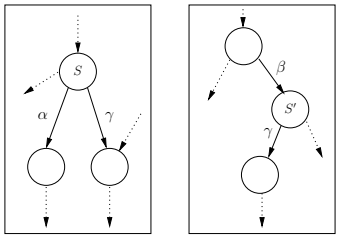

Figure 2: Asymmetric confusion (α, β)

III. Component Based Systems with Probabilistic Choice

A. Probabilistic Component based systems

In this section, we define component based systems that can make probabilistic choices.

Definition 5: A *view* of a component $\mathcal{A}_i$ is a pair (s, Γ), where $s \in S_i$ is a local state and $\Gamma \subseteq \Sigma_i$ is a set of transitions of $\mathcal{A}_i$. The view $view_i(g)$ of $\mathcal{A}_i$ in a global state g, includes the component s of $\mathcal{A}_i$ in g and the set of transitions $\Gamma \subseteq \Sigma_i$ that are enabled in g.

The view of component $\mathcal{A}_i$ is then dependent not only on the local state of this component but also on the local states of other components to enabled joint transitions in Σ_i. In essence, the view of a component in a given state summarizes the information it needs to select a transition and move to the next state. As we will see later, our implementation is required to stabilize the view of $\mathcal{A}_i$ before it can make a (probabilistic) decision about firing one of its enabled transitions.

Definition 6: A probability distribution $dist$ on a set X is a function $dist : X \to [0, 1]$ with $\sum_{x \in X} dist(x) = 1$.

Definition 7: A probabilistic component based system consists of a collection of components $\mathcal{A}_i = \langle \Sigma_i, S_i, \delta_i, s_i^0, f_i \rangle$, where the components $\Sigma_i, S_i, \delta_i, s_i^0$ are as in Definition 1. Let $f_i : S_i \times 2^{\Sigma_i} \mapsto dist(\Sigma_i)$ associate with each view, i.e., a local state $s \in S_i$ and a set of transitions $\Gamma \subseteq \Sigma_i$, a distribution function over Γ.

In a probabilistic component based system, a component is selected to make a transition based on its current local view.

Definition 8: An *execution* of a probabilistic component based system is a maximal alternating sequence $g_0 k_0 \alpha_0 g_1 k_1 \alpha_1 \ldots$ such that

- $g_i \in G$,
- $k_i \in [1..n]$,
- $\alpha_i \in \Sigma_{k_i}$,
- Component $\mathcal{A}_{k_i}$ with view $view_{k_i}(g_i) = (s, \Gamma)$ selects a transition α_i with probability $f_{k_i}(view_{k_i}(g_i))$.
- $\delta(g_i, \alpha_i, g_{i+1})$ holds.

B. Modeling Probabilistic Component Based Systems as Markov Decision Processes

One can model the collection of executions of a component based system using the global state graph $\langle G, E, g_0 \rangle$, where G is the set of global states, including the initial state g_0, and $E \subseteq G \times G$ is the edge relation $(g, g') \in E$ when $\delta(g, \tau, g')$ for some $\tau \in \Sigma$. Modeling probabilistic systems, including a single component system, can be done using Markov Chains, which add to the graph structure a probability distribution for moving to the next state:

Definition 9: A Markov Chain is a triple $(Q, \mathcal{P}, q_0)$, where Q is a set of states, q_0 is the initial state, $\mathcal{P} : Q \mapsto dist(Q)$.

One can automatically compute the probability for a Markov Chain to satisfy temporal properties. This can be seen as an extension to model checking [34].

Markov chains are appropriate for modeling sequential systems with probabilistic choices. However, sometimes the system includes both probabilistic and nondeterministic choices. The reason is that often a probability on making a nondeterministic choice is not given. In this case, one can still hope to provide some minimal and maximal probability for satisfying a temporal property, over all the possible nondeterministic choices. To model a system that can use nondeterministic and probabilistic choices, we use Markov Decision Processes.

Definition 10: A Markov Decision Process (MDP) is a tuple $\mathcal{M} = (Q, Act, \mathcal{P}, q_0)$, where Q is a finite set of states with initial state q_0, Act is a finite set of action, and $\mathcal{P} : Q \times Act \mapsto dist(Q)$ is a function that associate to each state q and action α a probability distribution on the set of successors.

As MDPs interleave both probabilistic and nondeterministic information, one first has to resolve the nondeterminism, which is done using schedulers that select the next action.

Definition 11: A *scheduler* (or a *strategy*) is a function $sch : Q^* \mapsto Act$ that selects an action given a finite sequence. A scheduler is *memoryless* if it depends only on the last state, i.e., $sch(g_0, g_1, \ldots g_n, g) = sch(g'_0, g'_1, \ldots g'_n, g)$.

An execution of an MDP alternates between nondeterminism via action selection, and probabilistic choices. Each pair $(\mathcal{M}, sch)$ of an MDP and a memoryless scheduler corresponds to a Markov Chain, see e.g., [9]. The extreme (minimal and maximal) probabilistic values of some important properties of probabilistic systems depend only on memoryless schedules. Then, as a system is modeled as an MDP, one can use classical probabilistic model checking techniques [9], [14], [34] to compute minimal and maximal probability values that the system satisfies a global property.

We are now ready to give an MDP interpretation to our probabilistic component based systems. In our case, each execution step consists of a nondeterministic choice between components with enabled transitions, and then a probabilistic choice by the selected component. This nondeterministic choice reflects an abstraction of the actual scheduling mechanism that is used to implement the selection of the component that choose the next transition.

Definition 12: The MDP corresponding to a probabilistic component based system is as follows:

1) In each global state $g = (s_1, \ldots, s_i, \ldots, s_n)$ of a set of components $\mathcal{A}_i$ for $i \in [1 \ldots n]$, one of the individual components $\mathcal{A}_i$ is selected. Thus, the selection of an action of the MDP corresponds to the selection of a component $\mathcal{A}_i$, which is making the decision among its current enabled transitions.
2) The probability distribution induced by the component $\mathcal{A}_i$ in state g for moving to state g' such that $g \xrightarrow{\tau} g'$, is given by $f_i(view_i(g))(\tau)$.

Our scheduler selects one component and decides on the next move. Observe that if two components collaborates on a given transition a, then the probability of a in the composition will depend on the component that was selected by the nondeterministic step and on its view. The same transition a can have different probabilities when selected by different components or by the same component with different views. In particular, this means that the probability can differ according to the set of other transitions that are co-enabled.

IV. Implementing Stochastic Component Based Systems

Goals. We provide here a distributed algorithm for component based system, which allows true concurrency between transitions in different components.

The requirements from the implementation algorithm are as follows.

- There is no centralized scheduling.
- Provide the ability of components to execute concurrently, when there is no need to block the simultaneous execution of different components.
- Allow performing a probabilistic decision when prescribed by the component specification.
- Use only standard operations on shared variables including operations that are used for scheduling, in particular, *compare-and-swap*.
- The algorithm is self contained: it does not rely on any particular scheduling or interaction algorithm in addition to the code provided.
- Separation and information hiding. Our goal is to allow the development of separate components and their composition with the need to reveal only little information about the internal structure.
- Shared transitions are implemented non-atomically. We do not assume that the local view (consisting of the local states and enabled transitions) is collected by a component in an atomic manner.
- The implementation must avoid livelocks. Actual progress by the components must be guaranteed.

The information required to be exported by the components is the following:

- The shared transitions. At runtime, there is a need to allow components to check the inclination (local enabledness) of other components to execute a shared transition.
- Static component information about the dependencies involving shared transitions needs to be known to the other components participating in these transitions. Specifically, if γ is a shared transition between $\mathcal{A}_j$ and $\mathcal{A}_i$, and in $\mathcal{A}_j$, β a subsequent γ, then this static information needs to be exported to $\mathcal{A}_i$ due to a possible asymmetric confusion involving β. This is the situation in Figure 2, where $\mathcal{A}_i$ is the left component and $\mathcal{A}_j$ is the right component. Note that β need not be a shared transitions. Although similar considerations could be applied to symmetric confusion, that case is avoided due to the structure of the system as explained below.

In scheduling of distributed systems with probabilistic choices [21], one models the execution as having two phases: first selecting the agent or component that makes the probabilistic choice, then firing the selected transition, which may be shared by several components. Therefore, our algorithm needs to lock a stable and consistent view for a components before it can make the probabilistic decision. We need to guarantee that the components that are not selected to make the decision cannot change the set of possibilities for the selected component. This, in turn, means that the implementation of the component based system needs to guarantee blocking the execution of transitions that are in a confusion situation with the ones that are candidates for the selection.

The algorithm alternates for each component between two phases. One phase attempts to protect its local view from changes so that it can make a probabilistic decision; then the component becomes a *master* for its selected transition and waits that the rest of the components execute their

part in the joint transition. The other phase checks whether the component needs to follow a shared transition selected by another component; then it becomes a *slave* to that component, and performs its part in the transition.

Locks. To achieve the above goals, our algorithm uses shared *locks*, behaving in a similar way as semaphores, but without blocking. In order to guarantee the situation that a component can make a selection, it first captures the following locks:

- $l_{(\alpha.\beta)}$ This is a lock for the involved asymmetric confusions: not allowing a transition that increases the set of choices for another component to fire before a selection is made. We observe that we can have both (α, β) and (β, α) independently as asymmetric confusions, but do not need two separate variables for these cases. As a result, the set of shared locks is decreased. This lock can be set up and tested by any component $\mathcal{A}_i$ that has either α or β in Σ_i. Initially, $l_{(\alpha,\beta)} \leftarrow 0$.
- l_i This is a lock for a component $\mathcal{A}_i$, captured before a transition $\alpha \in \Sigma_i$ that can appear in a probabilistic choice. This does not allow different components to make overlapping decisions about firing the same transition furthermore. This lock can be set up and test by each component $\mathcal{A}_j, j \neq i$ such that $\alpha \in \Sigma_j \cap \Sigma_i$. Initially, $l_i \leftarrow 0$.

Observe that we do not need locks for the symmetric confusion. This is because a lock on the involved components is captured, and prevents the other components from decreasing the current alternative choices. To see this, consider again Figure 1, where $\mathcal{A}_i$ is the left component and $\mathcal{A}_j$ is the right component. If $\mathcal{A}_i$ wants to make a move, it needs to capture the locks on both components, due to the fact that they are sharing the transition γ. Now, $\mathcal{A}_j$ cannot make a move and execute β, hence disabling γ; a lock on the symmetric confusion (α, β) is not needed.

Following the *two phase* locking principle of Dijkstra, we assume a total order $\ll$ between the locks. Capturing these locks is performed in ascending order, while releasing them is performed in descending order. We do not perform actual wait when we do not succeed in capturing a lock. But rather we use the compare and swap mechanism. Capturing the relevant semaphores guarantees that before a transition is selected to execution, the local view of a component is stable and consistent.

Additional shared variables. Our algorithm uses additional shared variables as follows:

- $m_{\alpha,i}$, for each shared transition α. This variable is set to 1 by component $\mathcal{A}_i$ (where $\alpha \in \Sigma_i$) when it becomes a master over α and is tested by the remaining slave components. Initially, $m_{\alpha,i} \leftarrow 0$.
- $f_{\alpha,i}$ for each shared transition α. This variable is set to 1 by a slave component $\mathcal{A}_i$ (where $\alpha \in \Sigma_i$) that participate in α and is tested by the master component upon execution α. Initially, $f_{\alpha,i} \leftarrow 0$.
- α_i, for a shared transition α is a variable that is set to 1 by $\mathcal{A}_i \in \mathcal{A}$ when its state is changed to $s \in S_i$ such that $\alpha \in s^\bullet$ and to 0 when moving away from such a state. The value of this variable can be accessed by any other agent $\mathcal{A}_j$ such that has $\alpha \in \Sigma_j$ in its set of (shared) transition. Initially, α_i.

Let V_α be the set of variables that include the locks $l_{(\alpha,\beta)}$ and let var_i be the set of shared variables and locks maintained by each component $\mathcal{A}_i$.

We use the standard atomic operations *compare-and-swap* on locks. Accordingly, we define two type of operations over the shared variables namely $capture$ and $release$. If l is a lock in var_i, then

- $capture(l)$ will try to set atomically $l \leftarrow 1$. It will fail if l is already equal to 1 and succeed otherwise.
- $release(l)$ operation will set atomically $l \leftarrow 0$.

Algorithm. The algorithm is as follows:

Algorithm 1 Distributed Scheduling Algorithm : Initialize

```
for all A_i ∈ A do
    s_i ← s_i^0;  /*initialize current state to initial*/
    var_i ← ∅;
end for
for all shared α ∈ s_i• do
    α_i ← 1;  /*mark sh. transitions from s_i enabled*/
end for
for all shared α ∈ Σ \ s_i• do
    α_i ← 0;  /*mark all other transitions as disabled*/
end for
```

Algorithm 2 Move

```
procedure MOVE(s_i, α)
    s_i ← s_i' such that δ_i(s_i, α, s_i');
    for all shared β ∈ s_i• do
        β_i ← 0;  /*disables old transitions*/
    end for
    for all shared β ∈ s_i'• do
        β_i ← 1;  /*enables new transitions*/
    end for
end procedure
```

Let the duration of executing a transition by the transformation span from the moment a master made selection until it releases all its slave (for a non shared transition, there are no salves). Then, transitions may overlap. Concurrency between independent transitions is reduced by our algorithm because of the need to stabilize the local views before making a probabilistic decision. This can happen with two transitions that are independent but form a confusion. This can also happen between two independent transitions of different components that can both make a choice on transitions shared with another component.

Algorithm 3 Distributed Scheduling Algorithm : Main Loop

11: **while** $\exists s' \in S_i, \exists \alpha \in \Sigma_i$ such that $\delta_i(s_i, \alpha, s_i')$ **do**
12: new_cycle: Let s_i be the current local state
13: RELEASE($v \in var_i$); /∗in descending order∗/
14: $var_i \leftarrow \emptyset$;
15: /∗collects transitions at current local state∗/
16: $now_i \leftarrow \{\alpha | \alpha \in s_i{}^\bullet\}$;
17: **for all** shared $\alpha \in now_i$ **do**
18: $f_{\alpha,i} \leftarrow 0$; /∗not master∗/
19: $m_{\alpha,i} \leftarrow 0$; /∗not slave∗/
20: **end for**
21: /∗checks if another component is master then, perform transition as a slave∗/
22: **for all** shared $\alpha \in now_i$ **do**
23: **for all** $\mathcal{A}_j \in \mathcal{A}$ such that $\alpha \in \Sigma_j$, $j \neq i$ **do**
24: **if** $m_{\alpha,j} = 1$ **then**
25: MOVE(s_i, α);
26: $f_{\alpha,i} \leftarrow 1$; /∗finishing slave part∗/
27: /∗waits master resets slave flag∗/
28: WAIT($f_{\alpha,i} = 0$);
29: GOTO(new_cycle);
30: **end if**
31: **end for**
32: **end for**
33: /∗update view w.r.t other components∗/
34: UPDATEVIEW(now_i);
35: /∗Capture locks for asymmetric confusions and involved components∗/
36: $var_i \leftarrow \cup_{\alpha \in now_i} V_\alpha \cup \{l_j \mid$ for each $\mathcal{A}_j$ s.t $\alpha \in \Sigma_j\}$;
37: **for all** $v \in var_i$, according to ascending order **do**
38: **if** $\neg capture(v)$ **then**
39: GOTO(new_cycle);
40: **end if**
41: **end for**
42: $old\_now_i \leftarrow now_i$;
43: UPDATEVIEW(now_i); /∗update view again∗/
44: **if** $old\_now_i \neq now_i$ **then**
45: GOTO(new_cycle);
46: **end if**
47: /∗probabilistic choice of a transition α in now_i∗/
48: $\alpha \leftarrow$F(s_i, now_i);
49: MOVE(s_i, α);
50: **if** α is shared **then**
51: $m_{\alpha,i} \leftarrow 1$; /∗becomes master∗/
52: **for all** $\mathcal{A}_j \in \mathcal{A}$ s.t $\alpha \in \Sigma_j$, $j \neq i$ **do**
53: WAIT($f_{\alpha,j} = 1$); /∗waits for slaves∗/
54: **end for**
55: $m_{\alpha,i} \leftarrow 0$; /∗when slaves finish reset master∗/
56: **for all** $\mathcal{A}_j \in \mathcal{A}$ s.t $\alpha \in \Sigma_j$, $j \neq i$ **do**
57: $f_{\alpha,j} \leftarrow 0$; /∗resets slave flag∗/
58: **end for**
59: **end if**
60: **end while**

Algorithm 4 UpdateView

procedure UPDATEVIEW(now_i)
for all shared $\alpha \in now_i$ **do**
for all $\mathcal{A}_j \in \mathcal{A}$ such that $\alpha \in \Sigma_j$, $j \neq i$ **do**
if $\neg \alpha_j$ **then**
$now_i \leftarrow now_i \setminus \{\alpha\}$;
end if
end for
end for
end procedure

The correctness of this algorithm follows from the following observations that can be proved using an induction on the execution of the algorithm.

- *Deadlock freedom.* The two phase locking of Dijkstra (lines 32-36 and 13) guarantees that no deadlocks can occur when different components try to block their local view. Then some component must be able to capture all of the locks it needs.
- *Livelock freedom.* In addition to the deadlock freedom though the use of Dijkstra's two phase locking, either a component is making a progress by becoming a master and choosing a transition, which the other components involved follow, or the local view has changed between the start of capturing locks (at line 31) and the end of capturing the locks at line 40. But if the view has changed, then at least one other component has made progress.
- *Consistency of local view.* The set of locks on components and confusions are designed to guarantee the following: if a set of locks for a component with some local view is captured and the view of that component has not changed since starting to capture these locks (as in the previous point), then no other component can change that view, i.e., add or remove enabled transitions or becoming a master on a transition of that component.
- *Scheduling.* The execution of conflicting or subsequent transitions never overlap. This follows from the need to capture a lock on the involved components.

Any execution of the components system can be simulated by an execution of the implementation, where overlapping transitions are ordered arbitrarily. This can be shown by scheduling the original transitions of the components and the additional care-taking steps of the algorithm in alternation. Conversely, the implementation does not allow new executions of the basic component based system to occur. Because of that, the minimums and maximums on probabilities of measurable events that do not depend on histories (such as reachability [9]) would not be affected by the implementation of the algorithm.

V. Experiments and Case Studies

We have implemented our algorithm in a prototype tool written in Java. Actually, probabilistic components (Robots) are implemented as stateful Java objects that are uniquely identified and that could be interconnected in order to communicate. Moreover, each component is a Java thread that runs independently and concurrently and which uses shared locks (*AtomicBoolean*), *volatile* variables (The value of this variable will never be cached thread-locally: all reads and writes will go straight to main memory), and non-blocking operations namely *compareAndSet()* to synchronize with other ones.

In addition, a particular component is aware of his neighbors, that is components sharing transitions with him in the following sense: only the identifiers of the neighbors and common transitions not their complete structure are shared. Transitions are also represented as Java objects that have labels and that encapsulate status information namely *enabled, shared, master, and slave*. The *enabled* (respectively *shared*) flag states if a transition is enabled (respectively *shared*) while *master* and *slave* tells if the transition owner is a master or a slave component.

We now give two case studies to illustrate the concrete implementation of the algorithm.

A. The Collaborating Coauthors Example

In this example, which combines both symmetric and asymmetric confusion, there are n coauthors, sitting in a ring, each can be involved with writing up to one paper, with one of its adjacent coauthors (the conference has very high standards, hence it is impossible to finish a paper by oneself). Hence, there is a probabilistic choice between collaborating with the coauthor on the left and with the coauthor on the right (state 1 of each component in Figure 3). In addition, each coauthor starts by performing some deep meditation (state 0 of each component in Figure 3), in which it has not yet started to think about writing a paper; only when finishing the meditation, it can proceed to try to collaborate with its neighbors. The choice of collaboration give rise to symmetric confusion, while the move from meditation to choice introduces asymmetric confusion between a coauthor and its neighbors. This example can be seen as a dual to the dinning philosophers, where a philosopher needs to gain resources (forks) on both sides in order to progress (eat). Here, a coauthor needs to gain interaction in one side in order to progress (submit a paper).

In the model in Figure 3, there is for instance one asymmetric confusion between transition a of author 0 and transition e of author 1. Indeed, if author 1 is in state 1 and author 0 in state 0, then author 1 can only collaborate with author 2, but he also want to know if author 0 will eventually become available in which case his decision may change. Another asymmetric confusion is between d of author 1 and b of author 0. An example of symmetric confusion is between b of author 0 and e of author 1.

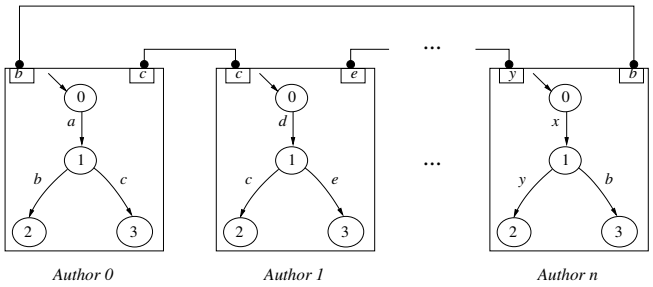

Figure 3: The Collaborating Coauthors Example.

The number of confusions in the system increases with the number of components which makes the scheduling problem quite tricky. Figure 4 shows the overhead in term of shared variables when the number of components increases. In addition, Figure 5 illustrates the evolution of the average execution time of the system with respect to the increasing number of components.

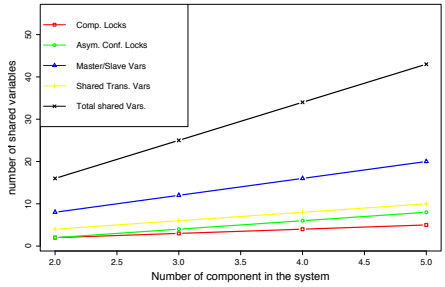

Figure 4: Number of shared variables with respect to the number of components in the system.

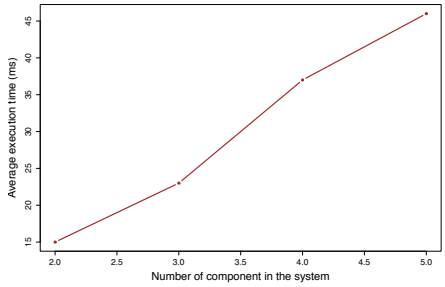

Figure 5: Average execution time with respect to the increasing number of components. The average execution time is obtained over 10 independent runs performed on a machine with 2.93 GHz Core 2 Duo processor and 4 GB of RAM.

B. Search and Rescue Operation

We demonstrate the use of the tool on a second case study that concerns the deployment of robots in a hostile

environment. Those are used to rescue motionless human victims from a toxic/nuclear disaster. The environment, consists of a safe home region from where the robots will start the rescue operation and to where they have to bring the victim. The radiation is caused by the collapse of a part of an insulating wall. To protect themselves and the victim from radiations, the robots will first have to re-build the insulating wall using building blocks spread within the environment.

Initially the robots are grouped into teams as they may not be able to perform a task alone. As an example, moving big building blocks may require team collaboration. To speed up the search operation, finding the victim is performed individually by all the deployed robots.

Each robot is equipped with sensors to detect building blocks, obstacles, and victims. In addition, they can perform the following tasks.

- *Search* The robot moves within the environment, avoid obstacles, and detects objects (building blocks or victim).
- *Move* When detecting a building block, a robot move it alone.
- *Move together* When detecting a building block, a robot can collaborate with other robots from his team to move the block.
- *Rescue* When detecting the victim, the robot will try to collaborate with other robots from his team to rescue it.

Probabilities are used to model the fact that a robot can decide to collaborate on moving a block even or move it alone. Indeed, due to its low battery status, a robot may prefer to perform less effort (collaborate) to save energy. The joint action (*move together*) depends also on the state of the other robots in the team which may be not available. This can introduce nondeterminism and confusion.

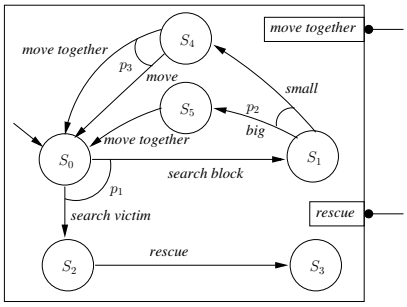

Figure 6: Robot Component Model.

Figure 6 shows a component view of the robot behavior. Initially (state S_0), two actions are possible; *search block* and *search victim*. The choice between those two actions is represented by a probability distribution, namely p_1. The use of probabilities rather than non determinism permits to configure robots so that some of them give a higher priority to one of the two tasks. When the victim is found, the robot performs the *rescue* action, that is, collaborate with teammates to bring the victim to the safe area (this needs teammates to be available). A building block is either big or small and, in our model, this distinction is made by a probability distribution p_2. When the block is small, the robot may either try to move it alone, or ask teammates to help it. The latter situation may occurs when the robot has already used a considerable amount of energy. In our setting, we do not encode energy level directly. Rather, we use a probability distribution p_3 (see Table I) to distinguish the case when the robot needs help from the one when it does not. In case the block is big, the robot will need to collaborate with teammates to move it.

Move	*Move_together*	
0.6	0.4	In case both action are possible, the robot decides depending on his energy status (reflected trough the probabilities).
1	0	It could be that only *Move* is possible because no teammate is available for collaboration. Then, *Move* has probability 1.

Table I: Example of the p_3 probabilistic distribution.

Various configurations of the robots were experimented to evaluate the scheduling algorithm in term of shared variables overhead and execution time. For doing so, we iterate on two parameters that are (1) the number of robots in the systems (*m*) and (2) the number of robots in each single team (*n*). There are several ways for confusion to occur. As an example, consider again the robot model in Figure 6. From state S_4, a robot has two possible actions, namely *move* and *move together* which is a joint action. Those actions may introduce asymmetric confusion. This is illustrated in Figure 7: if *Robot1* is in state S_4 and *Robot2* and *Robot3* are both in state S_1, then the view of *Robot1* contains only the *move* action since *move together* (joint action) is not enabled in the other robots. In this case, if *Robot1* is going to select *move*, it needs to make sure that none of *Robot2* and *Robot3* is going to change his view. Hence, it needs to capture asymmetric confusion locks for *Robot2* and *Robot3*. The same applies to *Robot2* in state S_4 when *Robot1* and *Robot3* are in state S_1 and also to *Robot3* in state S_4 when *Robot1* and *Robot2* are in state S_1. Then, for a team of 3 robots, each one of them needs to maintain 2 asymmetric confusion locks hence 6 locks per team in total. For example, for a configuration $C = (6, 3)$ the number of asymmetric confusion locks is 12. Remark that this locks are only used in specific cases as explained below.

In Figure 8, we show the amount of shared variables in the system when the total number of robots (respectively, the number of robots in a single team) increases. Moreover, the execution time is also measured and reported in Figure 9. Table II sum up all measure. In this table, *Configuration* represents the possible couples (m, n). The *number of shared variables* is composed by components locks (l_i, in the algorithm), master/slave status variables ($m_{\alpha,i}, f_{\alpha,i}$), shared

transitions variables (α_i), and asymmetric confusions locks ($l_{(\alpha,\beta)}$) respectively.

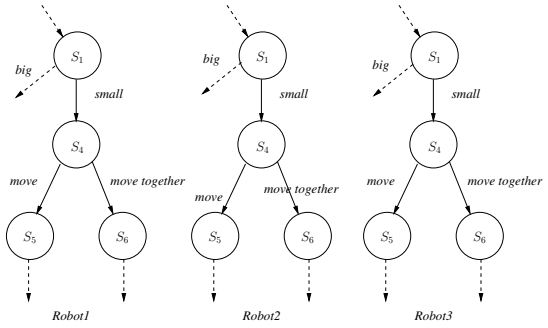

Figure 7: Part of the *Robot1*, *Robot2*, and *Robot3* models from Figure 6.

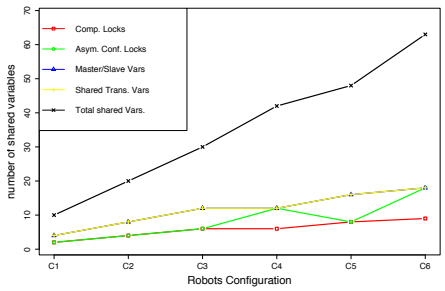

Figure 8: Number of shared variables with respect to different robot configuration.

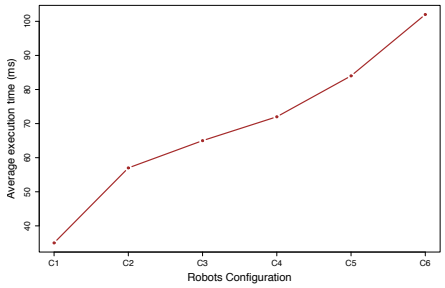

Figure 9: Average execution time with respect to different robots configuration. The average execution time is measured over 10 independent runs done on a machine with 2.4 GHz Core 2 Duo processor and 2 GB of RAM.

VI. Conclusion and Future Work

We described here an automatic transformation from a high level description formalism, similar to BIP, Petri nets and I/O automata, into a directly running shared-variables based system. Our formalism allows both concurrently executing components, with share transitions, modeling, e.g., synchronous communication, and probabilistic choices. The possibility to model and implement systems that have these two ingredients was considered problematic for many years, prompting suggestions such as restricting the generality of the systems [16]. Recently, some ideas started to emerge about allowing process-based or component-based probabilistic decisions. In the I/O automata context [24], [32], this was done by scheduling a single process to make its decision, while blocking the others. Subsequently, in [21], a model and a conceptual implementation that allow making concurrent probabilistic choices, without blocking the rest of the system, was presented.

Configuration (m,n)	*Number of Shared Variables*	*Avg. Ex. T. (ms)*
$C_1 = (2, 2)$	10 $(2 + 4 + 4 + 2)$	35
$C_2 = (4, 2)$	20 (4 + 8 + 8 + 4)	57
$C_3 = (6, 2)$	30 $(6 + 12 + 12 + 6)$	65
$C_4 = (6, 3)$	42 $(6 + 12 + 12 + 12)$	72
$C_5 = (8, 2)$	48 $(8 + 16 + 16 + 8)$	84
$C_6 = (9, 3)$	63 $(9 + 18 + 18 + 18)$	102

Table II: Results of the search and rescue scenario. *Avg. Ex. T. (ms)* = Average Execution Time (ms).

In this paper we looked at a practical transformation from a high level description of a concurrent and probabilistic system into a physical one. We provided an actual implementation using shared variables. the algorithm refines and improves that of [21]: it transfer it from the theoretical Petri-nets domain into actual implementation, with realistic assumption about non-atomicity. Then it improves on it by eliminating the number of shared locks that are needed. One of the strongest points of our transformation is that it does not assume any services or strong primitives from the underlying hardware and software. We provide a direct translation that can run on any reasonable shared variables with minimal synchronization primitives (we use the swap-and-compare primitive for non blocking locking). The complete correct-by-design synthesis of concurrent systems is shown to be undecidable [30]. On the other hand, the direct verification of actual complicated systems is often too hard for automatic verification. We believe that our approach, of allowing the designer to work with a high level description formalism, whose verification is more affordable, and translating it automatically into a directly running code, is an important step for synthesizing correct concurrent code.

Acknowledgement

We thank Gadi Taubenfeld for helpful discussion about concurrent synchronization primitives.

References

[1] S. Abbes. The (true) concurrent Markov property and some applications to Markov nets. In *Applications and Theory of Petri Nets*, LNCS 3536, pages 70–89, 2005.

[2] T. Abdellatif, S. Bensalem, J. Combaz, L. de Silva and F. Ingrand Rigorous design of robot software: A formal component-based approach In *Robotics and Autonomous Systems*, volume 60(12), pages 1563–1578, 2012

[3] M. Albanese. A constrained probabilistic Petri net framework for human activity detection in video. *IEEE Trans. on Multimedia*, 10(6):982–996, 2008.

[4] L. de. Alfaro and T. A. Henzinger. *Interface Theories for Component-Based Design*. In *EMSOFT*, volume 2211 of *LNCS*, pages 148–165, 2001.

[5] M. E. Andrés, C. Palamidessi, P. van Rossum, and A. Sokolova. Information hiding in probabilistic concurrent systems. *TCS*, 412(28):3072–3089, 2011.

[6] S. Abbes, and A. Benveniste/ Concurrency, sigma-Algebras, and Probabilistic Fairness. in *Foundations of Software Science and Computational Structures*, LNCS 5504, pages 380–394, 2008.

[7] S. Bensalem, M. Bozga, B. Delahaye, C. Jégourel, A. Legay and A. Nouri; *Statistical Model Checking QoS Properties of Systems with SBIP*, In Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, volume 7609 of *LNCS*, pages 327–341, 2012.

[8] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis. Rigorous Component-Based System Design Using the BIP Framework. In *IEEE Software*, volume 28(3), pages 41–48, 2011.

[9] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[10] M. Beccuti, G. Franceschinis, and S. Haddad. Markov decision Petri net and Markov decision well-formed net formalisms. In *Applications and Theory of Petri Nets*, volume 4546 of *LNCS*, pages 43–62, 2007.

[11] S. Bliudze and J. Sifakis. The algebra of connectors - structuring interaction in BIP. *IEEE Trans. Computers*, 57(10):1315–1330, 2008.

[12] L. Cheung, N. A. Lynch, R. Segala, and F. W. Vaandrager. Switched PIOA: Parallel composition via distributed scheduling. *TCS*, 365(1-2):83–108, 2006.

[13] P. R. D'Argenio, J.-P. Katoen, and E. Brinksma. *An algebraic approach to the specification of stochastic systems*. In *Conference on Programming Concepts and Methods*, volume 125 of *LNCS*, pages 126–147, 1998.

[14] L. de Alfaro. The verification of probabilistic systems under memoryless partial-information policies is hard. In *PROBMIV*, pages 19–32, 1999.

[15] C. Eisentraut, H. Hermanns, and L. Zhang. Concurrency and composition in a stochastic world. In *CONCUR*, volume LNCS 6269, pages 21–39, 2010.

[16] J. Esparza. Reduction and Synthesis of Live and Bounded Free Choice Petri Nets. In *Information and Computation*, 114(1): 50-87 (1994).

[17] S. Georgievska and S. Andova. Probabilistic may/must testing: retaining probabilities by restricted schedulers. *Formal Asp. Comput.*, 24(4-6):727–748, 2012.

[18] S. Giro and P. R. D'Argenio. On the expressive power of schedulers in distributed probabilistic systems. In *ENTCS*, 253(3):45–71, 2009.

[19] C. A. R. Hoare, Communicating Sequential Processes. In communication of the ACM, 21(8):666-677, 1978.

[20] J.-P. Katoen. GSPNs revisited: Simple semantics and new analysis algorithms. In *Application of Concurrency to System Design*, pages 6–11, 2012.

[21] Joost-Pieter Katoen, Doron Peled. Taming Confusion for Modeling and Implementing Probabilistic Concurrent Systems. In *ESOP*, volume 7792 of *LNCS*, pages 411-430, 2013.

[22] G. Katz, D. Peled, and S. Schewe. Synthesis of distributed control through knowledge accumulation. In *CAV*, volume 6806 of *LNCS*, pages 510–525, 2011.

[23] M. Kudlek. Probability in Petri nets. In *Fund. Informatica*, 67(1-3):121–130, 2005.

[24] N. A. Lynch, R. Segala, and F. W. Vaandrager. Observing branching structure through probabilistic contexts. *SIAM J. Comp.*, 37(4):977–1013, 2007.

[25] N. Lynch and M. R. Tuttle. *An Introduction to Input/Output Automata* In CWI-quarterly, volume 2(3), 1989

[26] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley, 1995.

[27] A. Mazurkiewicz. Introduction to trace theory. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*. World Scientific, 1995.

[28] R. Milner *A Complete Axiomatisation for Observational Congruence of Finite-State Behaviors*. In *IandC*, volume 81(2), pages 227–247, 1989.

[29] J. A. Pérez, R. Corchuelo, and M. Toro. An order-based algorithm for multiparty synchronization. *Concurrency - Practice and Experience*, 16(12):1173–1206, 2004.

[30] Amir Pnueli, Roni Rosner, Distributed Reactive Systems Are Hard to Synthesize. in FOCS 1990: Pages 746-757, 1990.

[31] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 2005.

[32] R. Segala and N. A. Lynch. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.*, 2(2):250–273, 1995.

[33] G. Taubenfeld. *Synchronization Algorithms for Concurrent Programming*. Prentice Hall, 2006.

[34] M. Y. Vardi. *Automatic Verification of Probabilistic Concurrent Finite-State Programs*. In *Annual SFCS*, IEEE Computer Society, pages 327–338, 1985.

[35] D. Varacca and M. Nielsen. Probabilistic Petri nets and Mazurkiewicz equivalence. Unpublished manuscript, 2003.

[36] S. H. Wu, S. A. Smolka and E. W. Stark. *Composition and Behaviors of Probabilistic I/O Automata*. In *CONCUR*, volume 836 of *LNCS*, pages 513–528, 1994.