Softw Syst Model
DOI 10.1007/s10270-014-0451-z

THEME SECTION PAPER

Knowledge-based construction of distributed constrained systems

Susanne Graf · Sophie Quinton

Received: 22 November 2013 / Revised: 19 September 2014 / Accepted: 9 December 2014

Abstract The problem of deriving distributed implementations from global specifications has been extensively studied for a number of application domains. We explore it here from the knowledge perspective: A process may decide to take a local action when it has enough *knowledge* to do so. Such knowledge may be acquired by communication through primitives available on the platform or by static analysis. In this paper, we want to combine control and distribution, that is, we need to impose some global control constraint on a system executed in a distributed fashion. To reach that goal, we compare two approaches: either build a centralized controlled system, distribute its controller and then implement this controlled system on a distributed platform; or alternatively, directly enforce the control constraint while implementing the distributed system on the platform. We show how to achieve a solution following the second approach and explain why this is a pragmatic and more efficient strategy than the other, previously proposed one.

Keywords Distributed implementations · Knowledge · Controlled system · Correct-by-construction · Implementation relation · Knowledge preservation

Communicated by Prof. Einar Broch Johnsen and Luigia Petre.

S. Graf (✉)
VERIMAG/CNRS, Université Joseph Fourier, Grenoble, France
e-mail: Susanne.Graf@imag.fr

S. Quinton
INRIA Rhône-Alpes, Grenoble, France

1 Introduction

Building correct distributed systems is a challenge because they are too complex for global verification. An interesting solution consists in starting from a centralized specification of the system under construction, verifying all properties of interest on this centralized specification—which has a much lower complexity than verifying the corresponding distributed system—and finally deriving a distributed solution using some correct-by-construction approach. In addition, we want here to *control* the distributed system so as to guarantee some global property Ψ.

This leaves us with two problems which must be addressed together:

- *Control* Given a system S and a global safety constraint Ψ to be enforced, how to implement a controlled system S^{Ψ} behaving like S but guaranteeing Ψ? Note that if S is a global system (or specification), this problem can be solved easily.
- *Distribution* Given a system S, how to implement it in a distributed fashion? This means decomposing S into k processes $\pi_1, \ldots, \pi_k$ executing on a distributed platform, either totally agnostic of each other or communicating—in a limited way—through the communication primitives provided by the platform.

In this paper, we discuss two strategies for handling control and distribution jointly. The first strategy [11] consists in solving first the control problem on the centralized specification, then distributing the obtained controller and finally distributing the resulting constrained system forgetting about the original control problem. We propose here a second, integrated solution which solves the control and

Published online: 26 February 2015

distribution problems jointly. We show that this approach may avoid unnecessary synchronizations while allowing use of *knowledge* [9], as in the layered approach.

More specifically, we formulate the problem of achieving a distributed implementation of a centralized specification as a control problem which can trivially be extended to take into account also the original constraint Ψ. We define a knowledge-based approach to derive a *distributed* controller achieving that goal. We also discuss how to achieve the knowledge needed by the controllers of individual localities by communication and/or statically available knowledge that may be derived from platform or application domain induced conditions or obtained by static analysis of the centralized specification—under the condition that this knowledge is preserved in the distributed context.

One interesting side effect of this work is to make explicit some underlying assumptions made by previous knowledge-based work on distributed control. Indeed, the appropriate notion of what is a *correct* implementation solving both the control and the distribution problems depends on the (global) properties that we want to see *preserved*. In particular, one must define the meaning in a distributed setting of a constraint Ψ defined on the centralized specification, or of a property φ that it should satisfy; the reason is that the distributed implementation and the centralized specification are not defined on the same set of states, and transitions of the distributed implementation are transitions *local to a process* (which are part of a global transition of the original specification).

Finally, another contribution of this paper is that we show how to prove correctness of a distributed implementation in a compositional manner by separating the proof into a framework- and an application-dependent part: (1) We formulate several possible implementation relations $\prec$ and show how they can be formulated as a control problem; (2) we propose a distributed solution to this problem in the form of a set of knowledge properties and a communication strategy for achieving the required knowledge that we prove correct for some implementation relation $\prec$. For the proof of any concrete algorithm—that we do not provide here—it is then sufficient to (3) prove that it guarantees these knowledge properties.

We use here safe and elementary Petri nets as a convenient formalism to represent global system specifications, and local Petri nets extended with data and transition guards—one per process—to represent controlled and distributed systems at an abstract level. Petri nets have concurrency and synchronization as *first class citizens*, and these are also the important concepts for our purpose. The applicability of the approach presented here is, however, in no way restricted to Petri nets.

This paper, which is an extended version of [13], is structured as follows. In Sect. 2, we give an overview on approaches that have been proposed to derive distributed implementations from global (controlled) specifications. In Sect. 3, we recall some standard notions and notations for Petri nets [26,28], and we formulate the centralized controller synthesis problem in terms of a simple class of extended Petri nets which allow enforcing a given control invariant. In Sect. 4, we introduce a notion of distributed Petri net and define how it relates to global Petri nets. We then formulate the distributed implementation problem as a (centralized) control problem. How to distribute this centralized controller is shown in Sect. 5, using the notion of knowledge and defining how communication can be integrated into the formal framework presented. Finally, in Sect. 6, we put into use all previous results for solving our initial problem of imposing control constraints on distributed computations. More specifically, we show the benefits of an integrated approach dealing with control and distribution jointly.

2 Related work

The problems of deriving a distributed implementation from a global specification and that of deriving a controller enforcing a global (safety) property have been studied intensively in the past. Here, we give a short overview of some important results in these domains, organized around three topics: The distributed implementation of synchronous languages, the derivation of protocols from specifications, and distributed control, with an emphasis on knowledge-based approaches. Other closely related areas are test and analysis of distributed implementations which we do not discuss here.

2.1 Distributed implementation of synchronous languages

In synchronous languages [5], global specifications are given as a set of concurrent interacting components with local data, similar to what is done in hardware description languages. However, classical compilers of synchronous languages do not generate a distributed implementation, but a sequential one. The need to distribute such a specification stems from the need to execute it on a distributed platform, where different specification level components run on different hardware units. In this context, the target system is typically a real-time controller with rather tight synchronization constraints.

In this context, the control flow is driven by (local) clocks, and the data exchanged between locations are continuous flows. Most synchronous languages define semantically Kahn networks [17], that is, deterministic specifications where each variable is written at most once in each computation step, and no circular dependencies exist among them. Therefore, according to [17], achieving a correct distributed execution is straightforward on a platform with communication through unbounded FIFO buffers. The Esterel language [6] allows also fixpoint computations within a step which imposes additional constraints. The so obtained distributed implementations are reliable, but generally uninteresting in a context imposing (hard) real-time constraints to be met. Communication and computation time need to be bounded, such that bounded buffers are sufficient and real-time constraints are guaranteed [7].

2.2 Protocol derivation

In the domain of telecommunications, automatic protocol generation from a global service specification has been a hot topic in the eighties. An *action* of a global service specification may represent an (oriented) data transfer or a genuine synchronization belonging to more than one physical location. Besides, specifications often feature some nondeterminism which represents both, abstraction of how decisions are taken as well as some degree of openness of the design to be resolved at a later stage.

There has been a huge amount of work in the eighties on specifications given by communicating finite state machines [8,27,35] or formal specification languages such as LOTOS [15,18,30,34]. There, *joint transitions* represent message passing with a well-identified source and target. All these approaches are more or less *pattern based*, where for certain specification patterns a choice of corresponding implementation patterns is proposed. As an example, the simplest pattern transforms an assignment $x := y$ where x and y reside on different locations into a message exchange, transferring the value of y to the right location. [18] treats a quite complete subset of LOTOS and proposes *model transformations* to be applied to the abstract syntax tree. The paper [27] is more conceptual but includes patterns for achieving reliable communication by means of timeouts and retransmissions. This paper proposes also a parallelization of local activities. Few papers consider timing constraints to be satisfied [34].

Some works propose methods for Petri nets with data transfer (through registers). For example, [32] presents an algorithm for generating, starting from a Petri net, a message passing protocol by means of a set of message synthesis rules. This line of work supposes that (1) the control over interactions (determining who initiates it) is solved a priori, derived from the direction of data flow and that (2) conflicts can always be solved locally. A more general method, dealing also with conflicts and multi-party synchronizations, is proposed by Bagrodia [1], taken up in [25] for defining the α-*core* protocol; note that these protocols are not automatically derived.

Correctness proofs are rarely given. When they exist, they provide more or less formal arguments establishing that the protocol guarantees atomicity. An explicit knowledge-based formulation of the protocol under study and the property to be proven would allow for simpler proofs, and we hope, also more efficient protocols.

2.3 Distributed control and its knowledge-based formulations

The problem of achieving distributed control of a plant with respect to a global specification is closely related to the distribution problem. Here, for a given set of possible next actions supported by the plant, the aim is to allow in a distributed fashion one or more of them to be executed, using a set of controllers with some partial vision of the global situation. This requires—as before—to find some enabled actions, and, when there is more than one, to detect whether there is a conflict and to choose among enabled actions if needed. In this specific context, instead of initiating the local part of a global action, local controllers provide a judgment on whether or not they propose an action for execution, and these local judgments are then somehow fused into a global decision (see e.g., [24,29,31] and [33] for a generalized decision architecture). The problem we address here is slightly different as we do not suppose the existence of supervisor fusing local views.

We are particularly interested in the methods presented in [16,22,23] where a knowledge-based presentation of the distributed control problem is proposed for systems, although without the possibility of actual conflict situations. In [22], only *negative knowledge* is used: A local controller *knows* that an action a cannot be executed if this is due to its local protocol, and in order to forbid the execution of a, at least one local protocol must do so. This means that action a is allowed when no individual controller knows the contrary. In [16], the notion of *knowledge-based protocol* is proposed as a means for representing protocol specifications abstractly: The local action a_i of P_i is *enabled* if P_i *knows* this fact in its present state. Obviously, knowledge depends on the global system and not just on the local state. Constructing a distributed protocol consists therefore in transforming this *external knowledge* into an *acquired knowledge* which can be locally exploited. We take up upon this work.

2.4 Solving control problems for subsequent distribution

The previously discussed knowledge-based approach has been taken up and generalized in [3,4,11,12,20] by suggesting the use of model checking—or more generally, static analysis—for calculating knowledge properties of local states. This is done for global service specifications given in terms of Petri net-like formalisms. The objective in the above-cited work is not maximal progress, but preservation of deadlock freedom and minimization of need for communication. A problem is that there may not exist enough knowledge to take any local decision. In [2], it is suggested to merge local processes into fewer, larger ones when the existing knowledge cannot avoid deadlock situations, and process fusion is stopped when in S there is sufficient knowledge to allow these larger processes to take local decisions. In [3,11], it is proposed to enrich the specification with some additional actions representing *temporary synchronizations* in order to preserve more of the initial concurrency. This work relies on some protocol such as α-core for achieving a distributed implementation, in particular, for resolving conflicts. In [4], knowledge computation is used to avoid actual conflicts at execution time by eliminating them statically.

Our goal here is to apply such knowledge-based approaches also to the underlying distribution protocol. For example, we would like to optimize the extension of the α-core protocol for dealing with priorities defined in [14]. To do so, we aim at a knowledge-based formulation of this algorithm, such that a given protocol step is only executed if the knowledge to be acquired in this step is not yet available. The approach defined in this paper should provide the basic tools for achieving this goal, but actually doing it is beyond its scope.

3 Centralized controlled specifications

In this section, we recall some standard definitions and notations used throughout the paper. We use safe *Petri nets* [26,28] to represent centralized specifications and define a standard global semantics for Petri nets as the set of traces defined by their possible executions. We also formulate the centralized controller synthesis problem, and we represent controlled specifications by means of a simple class of *extended Petri nets*.

3.1 Petri nets

Definition 1 A *Petri net* N is a tuple (P, T, E, s_0) where:

- P is a finite set of *places*. The set of *states* (markings) is defined as $S = 2^P$.
- T is a finite set of *transitions*.
- $E \subseteq (P \times T) \cup (T \times P)$ is a bipartite relation between places and transitions.
- $s_0 \subseteq 2^P$ is an *initial state* (initial marking).

For a transition $t \in T$, we define the set of *input places* $^\bullet t$ as $\{p \in P | (p, t) \in E\}$, and the set of *output places* $t^\bullet$ as $\{p \in P | (t, p) \in E\}$.

A transition t is called *enabled* in a state s if $^\bullet t \subseteq s$ and $(t^\bullet \backslash {}^\bullet t) \cap s = \emptyset$. We denote the fact that t is enabled in s by $s[t\rangle$.

An *event*, corresponding to the firing of t, leads from state s to state s', which is denoted by $s[t\rangle s'$, when t is enabled in s and $s' = (s \backslash {}^\bullet t) \cup t^\bullet$.

Throughout the paper, we use the Petri net of Fig. 1 as a running example. As usual, transitions are represented as segments, places as circles, and the relation E as a set of arrows from transitions to places and from places to transitions. The Petri net has places named p_i and transitions named $a, b, \ldots, g$. We represent a state s by *tokens* inside some places of s. In the example, the depicted (initial) state s_0 is $\{p_1, p_2, p_5\}$. The transitions enabled in s_0 are a and b. Firing a from s_0 means removing the token from p_1 and adding one to p_3.

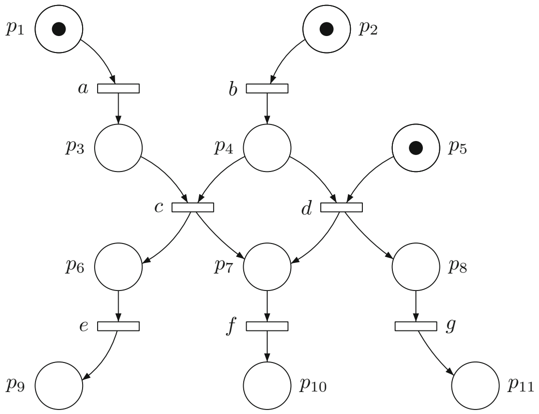

Fig. 1 A Petri net with initial state $\{p_1, p_2, p_5\}$

Note that there cannot be more than one token in any place. Indeed, a transition t is enabled in a state s only if (after removing the tokens from the input places of t) there is no token in any of the output places of t. That is, using standard vocabulary for Petri nets, we consider here Petri nets that are *one-safe* by construction. A state s is in *deadlock* if there is no enabled transition from it.

Definition 2 Two transitions t_1 and t_2 are *concurrent* if $(^\bullet t_1 \cup t_1^\bullet) \cap (^\bullet t_2 \cup t_2^\bullet) = \emptyset$. Two transitions t_1 and t_2 are *in conflict* if $^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$.

For example, transitions a and b are concurrent whereas transitions c and d are in conflict.

As usual, we define *event traces* as maximal (non-extendable) sequences of events.

Definition 3 An *event trace* of a Petri net N is a maximal sequence of events $s_0[t_1\rangle s_1 \cdot s_1[t_2\rangle s_2 \cdot \ldots$ where s_0 is the initial state of N, and any two consecutive events share their final, respectively, initial state in the obvious manner.

We denote the set of event traces of N by *traces*(N). A state is *reachable* in N if it appears in at least one event trace of N. The running example has 16 reachable states, for instance $\{p_3, p_7, p_{11}\}$. We denote the set of reachable states of N by *reach*(N).

3.2 Centralized control and extended Petri nets.

On top of the Petri net specification, we want to enforce some global (safety) constraint. We consider here only a restricted type of constraint.

Definition 4 Given a Petri net $N = (P, T, E, s_0)$ with set of states S, a *control invariant* $\Psi \subseteq S \times T$ defines for each state s the set of transitions allowed in s.

A control invariant Ψ has the potential to forbid the execution of some events allowed by N. Notice that any enforceable safety property can be transformed into a control invariant, possibly after extending the set of states S.

Definition 5 We call a Petri net constrained by a control invariant Ψ a *constrained Petri net* which we denote simply as the pair (N, Ψ). An event trace of (N, Ψ) is defined as the maximal prefix $s_0[t_1\rangle s_1 \cdot s_1[t_2\rangle s_2 \cdot \ldots$ of an event trace of N such that for each event $s[t\rangle s'$ in the sequence, (s, t) is in Ψ.

We denote the set of event traces of (N, Ψ) by *traces* (N, Ψ) and its set of reachable states, that is, states that appear in at least one event trace of (N, Ψ), as $reach(N, \Psi)$.

One example of such control invariant are priority orders as in [2,11] used to arbitrate between simultaneously enabled transitions in N, where we not require the priorities to be defined among conflicting transitions only.

Example 1 A *priority order* $\ll$ is a partial order on the transitions T of N. In a state s, transition t is said to be *maximally enabled* if it is enabled, and there is no transition t' with higher priority (i.e., such that $t \ll t'$) that is enabled in s.

The control invariant induced by $\ll$ is $\Psi_{\ll} = \{(s, t) \in S \times T \mid t \text{ maximally enabled in } s\}$.

Consider Petri net N of Fig. 1 and priority order $\ll$ defined by $\{a \ll b,\ e \ll f,\ f \ll g\}$. The state $\{p_2, p_3, p_5\}$ is in $reach(N)$ but not in $reach(N, \Psi_{\ll})$, because in the initial state, a may not be fired as long as b is enabled.

Here, we use *extended Petri nets* [10] to implement constrained Petri nets. An extended Petri net is obtained by extending N with variables and enabling conditions on transitions, as well as data transformations associated with transitions.

Definition 6 An *extended* Petri net N_{ext} consists of

- a Petri net $N = (P, T, E, s_0)$ with states S, called the *underlying Petri net* of N_{ext};
- a finite set of variables V with value domain $\mathcal{V}$ and given initial values v_0;
- for each transition $t \in T$:
  - an *enabling condition* en_t, i.e., a predicate on V and S defining in which states and for which values of the variables in V transition t is allowed. We note $(s, v) \models en_t$ when en_t holds in s for values $v \in \mathcal{V}$;
  - an *update predicate* or *update function* f_t associating new values to variables in V.

An *execution* of an extended Petri net is a maximal sequence of the form $(s_0, v_0) \cdot t_1 \cdot (s_1, v_1) \cdot t_2 \cdot (s_2, v_2) \ldots$ such that for all $i \geq 0$ we have: $s_i[t_{i+1}\rangle s_{i+1}$, $(s_i, v_i) \models en_{t_{i+1}}$ and $v_{i+1} = f_{t_{i+1}}(s_i, v_i)$. The corresponding *event trace* is obtained by projecting out variables and representing the sequence as a sequence of events of N. We denote the set of event traces of N_{ext} by $traces(N_{ext})$. We call reachable states of N_{ext} the states that appear in at least one event trace of N_{ext} and extended reachable states of N_{ext} the extended states that appear in at least one execution of N_{ext}.

Note that N_{ext} can only restrict the event traces of its underlying Petri net N, not generate new ones. It may, however, introduce deadlocks, and more generally, affect the progress properties of N, meaning that the traces of N_{ext} may be just prefixes of traces of N.

Coming back to the control problem to be solved, we now show how a Petri net N can be extended to enforce a control invariant Ψ. One of the challenges in the remainder of this paper is to distribute this controller. But first, we need to define what a *correct implementation* of a constrained Petri net is.

Definition 7 An *extended* Petri net N_{ext} *implements* a Petri net N constrained by Ψ if $traces(N_{ext}) \subseteq traces(N, \Psi)$.

Note that this is just one possible definition of correct implementation. First, it forces the use of an identical state structure, which could be easily generalized by defining a homomorphism or equivalence relation between states. This definition also forbids N_{ext} to introduce new deadlocks compared to N constrained by Ψ. It does not require any stronger progress, meaning that the only properties which are guaranteed to be preserved by this definition are safety and deadlock freedom. Quite clearly, other definitions are needed for stronger preservation guarantees.

Theorem 1 *For a Petri net N and a control invariant Ψ on N, consider the extended Petri net $N_{\Psi} = (N, V, \{en_t\}_{t\in T}, \{f_t\}_{t\in T})$ where*

- *the set of variables is empty (enabling conditions depend only on the current state of N);*
- *for each transition $t \in T$, en_t holds in state $s \in S$ if and only if $(s, t) \in \Psi$.*

Then, N_Ψ implements N constrained by Ψ, that is, traces$(N_\Psi) \subseteq$ traces(N, Ψ).

We say that $(V, \{en_t\}_{t\in T}, \{f_t\}_{t\in T})$ *is a* controller *for* (N, Ψ).

Proof [11] Any event trace of N_Ψ is also an event trace of (N, Ψ). Indeed, in N_Ψ, for any transition t and any state s, t can be fired in s if and only if: (1) t is enabled in s, as defined by N; and (2) en_t holds in s, that is, $(s, t) \in \Psi$. Thus, N_Ψ allows exactly all events of N allowed by Ψ. Therefore, we even have the stronger property that $\mathit{traces}(N_\Psi) = \mathit{traces}(N, \Psi)$, i.e., this controller is the most permissive one for N enforcing Ψ. □

Note that to enforce a control invariant, as above, a centralized controller of a Petri net does not require any variable and is defined using enabling conditions depending only on the current global state. In Sect. 4, however, for distributing Petri nets even without data, the local controllers of the distributed system need to be extended with variables.

4 Distributed specifications

In practice, the systems we consider consist of a set of processes that are executed in a concurrent fashion and which may communicate with others using mechanisms provided by the chosen platform. This is why we now consider *distributed* specifications obtained by partitioning a Petri net into a set of (independent) processes: transitions shared by processes then correspond in practice to a set of local transitions, and similarly for events.

In this section, we first define distributed Petri nets and discuss several possible semantics for them. Then, we show how to formulate the problem of building a correct implementation of a distributed Petri net by means of a control invariant to imposed.

4.1 Distributed Petri nets and their semantics

We define a distributed Petri net as a system of sequential processes.[1]

Definition 8 A *process* π of a Petri net N is a subset of the places of N (i.e., $\pi \subseteq P$) such that there is always exactly one token in π. A *distributed Petri net* is a pair (N, Π) where N is a Petri net as in Definition 1 and Π a set of processes of N defining a partition of P.

In the remainder of this section, we assume a distributed Petri net (N, Π) as above. A transition $t \in T$ is now seen as a set of *local transitions* t_π, one for each process π involved in t. Formally, for each transition $t \in T$, we denote $\mathit{proc}(t)$ the set of processes which have at least one place in ${}^\bullet t$ and we say that these processes are *involved* in t. Because we consider sequential processes, this set is exactly the set of processes which have at least one place in $t^\bullet$, and furthermore, the processes in $\mathit{proc}(t)$ have exactly one place in ${}^\bullet t$ and $t^\bullet$. As an example, Fig. 2 illustrates a possible distribution of the Petri net of Fig. 1. To improve readability, we write t_i to denote t_{π_i} for $t \in \{c, d\}$.

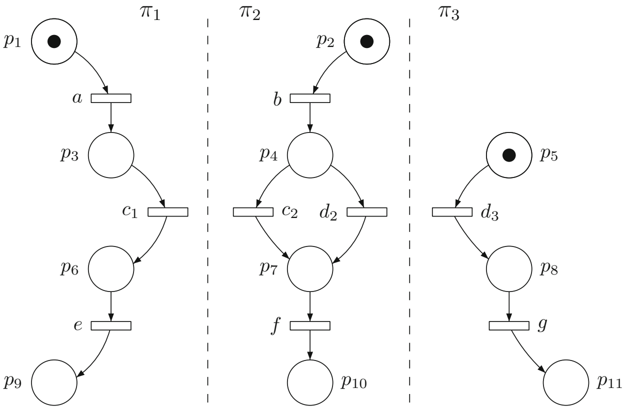

Fig. 2 Graphical representation of a distributed Petri net (for $t \in \{c, d\}$, t_i denotes t_{π_i})

At this point, note that a distributed Petri net, as the one of Fig. 2, trivially defines the following:

- one Petri net per process, where each process π represents a Petri net in isolation with its own places, transitions, states, event traces and so on, called the *local* states of π, etc.
- one global Petri net denoted N^Π where local transitions are considered as standard Petri net transitions, and processes are then forgotten. In a distributed setting, this Petri net can be implemented very easily by considering all processes as being independent of each other. But the interaction through shared transitions is missing: The key to proper distribution of a Petri net is a mechanism to guarantee a *correct* implementation of these shared transitions.

The projections of a global state onto a local state, a global transition onto a local transition, and a global event trace onto a local event trace are defined in the obvious manner.

Another point that need to be discussed concerns Petri nets with cycles or loops. For simplicity of notation, we assume in the following that transitions (and states) occur at most once in an event trace. This is typically obtained by unfolding loops as illustrated on Fig. 3: A loop transition t is replaced by an infinite set of transitions representing the first firing of t, its

[1] Alternatively, we could allow processes with more than one token, but to keep the framework simple, we restrict ourselves to simple sequential processes.

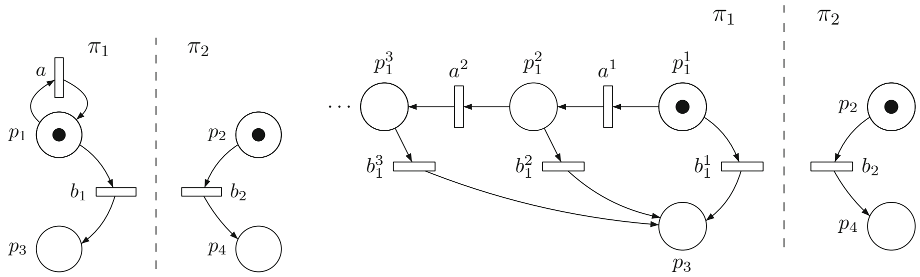

Fig. 3 A distributed Petri net with a loop *(left)* and its unfolding *(right)*

second firing, etc. This boils down to encoding in a state the prefix of the execution that led to it. In a distributed setting, we only want to encode information about *local* prefixes in *local* states, and therefore, we unfold the local Petri nets which constitute N^{Π} rather than N.

After unfolding, a transition t of the original Petri N now may correspond to several possible sets of local transitions. For example, in Fig. 3, b corresponds to $\{b_1^1, b_2\}$, $\{b_1^2, b_2\}$, etc. and a corresponds to a^1, a^2, etc. We call these sets *unfolded transitions* of b (and of N by extension). Note that local transition b_2 of process π_2 is part of several unfolded transitions which π_2 cannot locally distinguish. These transitions, however, are such that at most one of them may occur in any trace.

Notation 1 *We denote T^u the set of unfolded transitions of N.*

Given a transition $t \in T^u$, we denote by t_π the (unique) corresponding local transition in $\pi \in proc(t)$. Symmetrically, a local transition t_π may correspond to several transitions in T^u (but only one in T). We denote $T(t_\pi)$ the set of such transitions.

From now on, by (N, Π) we refer to an unfolded distributed Petri net. In addition, when discussing local transitions, local states, etc. we always refer to the local elements of the unfolded distributed Petri net. Of course, unfolding results in Petri nets with an infinite number of states and transitions. This problem, however, can trivially be solved, also in a distributed context, by adding appropriate index variables and index update functions to count (locally) the number of occurrences of local transitions—remember that our approach also applies to extended Petri nets.

As a first step toward distribution, let us now discuss how to relate global and local event traces, as is state of the art in concurrency theory. We define *concurrent event traces* to describe the behavior of the distributed Petri net based on local traces and an equivalence relation between local transitions corresponding to the same (global) transition.

Definition 9 A *concurrent event trace* for (N, Π) is a pair $(\{\sigma_\pi\}_{\pi\in\Pi}, \equiv)$ consisting of:

- a set of local event traces σ_π, and
- an equivalence $\equiv$ on local transitions of different processes

for which there exists a (global) event trace σ of N ensuring that:

- the local event traces σ_π are projections of σ, and
- two local transitions are equivalent according to $\equiv$ if and only if they are projections of the same[2] (global) transition $t \in T^u$.

Here, the notion of projection is extended to relate, after renaming, local transitions, states and traces of the unfolded Petri net to their original global counterparts in N.

Example 2 Consider again the Petri net N of Fig. 2. One possible concurrent trace of N is $(\{a, b \cdot d_2 \cdot f, d_3 \cdot g\}, \{d_2 \equiv d_3\})$. To simplify notation, we represent here event traces using transitions instead of their corresponding events. This concurrent trace may be obtained as a projection of the global trace $b \cdot a \cdot d \cdot f \cdot g$.

Notice that, for each event trace σ of N, there exists exactly one concurrent event trace for (N, Π). We denote *c-traces*(N, Π) the set of concurrent event traces of (N, Π). On the other hand, several (global) event traces may correspond to a given concurrent event trace: In our example, also the trace $b \cdot d \cdot g \cdot a \cdot f$ yields the presented concurrent trace. In fact, all global traces which project onto the same local traces also induce the same relation $\equiv$. These global traces are obtained from each other by reordering of concurrent events and form a *class*.

Definition 10 The *class* of a concurrent event trace $(\{\sigma_\pi\}_{\pi\in\Pi}, \equiv)$ is the set of (global) event traces obtained by merging the local event traces σ_π as follows: All events whose transitions are related by $\equiv$ are merged into a unique event, and all others are interleaved.

Example 3 Consider the concurrent trace of Example 2. In the initial state, a, b and d_3 are locally enabled, but d_3 cannot

[2] As already discussed, a (folded) transition of N may appear several times in σ but each occurrence can be mapped to a different $t \in T^u$ so that a local transition t_π is part of only one t occurring in σ.

be fired without d_2 because of $\equiv$. Thus, any trace σ in the class of this concurrent trace starts either with a or with b. If σ starts with b, state $\{p_1, p_4, p_5\}$ is reached, where transitions a, d_2 and d_3 are locally enabled. If d_2 and d_3 are chosen (together, as they are equivalent), then σ continues with d which corresponds to the joint execution of d_2 and d_3 (remember that it is unique).

Note that the union of all classes of concurrent traces in *c-traces*(N, Π) clearly is identical, after renaming, to the set *traces*(N) of global traces of N. First, because all global traces are in the class of their corresponding concurrent trace. Second, because all traces in the class of a concurrent traces are indeed traces of N. A class is in fact a trace in the sense of concurrency theory (see, for example, [19]).

In addition, each class defines a strict partial order on global (unfolded) transitions which we denote $\prec^{\{\sigma_\pi\}_{\pi\in\Pi}}$, namely the precedence relation that is common to all traces of the class. For our example, we have, e.g., that $b \prec d$ (which we also write b precedes d) but also $b \prec g$, intuitively because d acts as a synchronization barrier between the two transitions.

Concurrent event traces are nice because they describe exactly the set of (global) event traces of the original Petri net using only local traces. Unfortunately, *synchronizations*—requiring a set of local transitions of different processes to be executed jointly—as defined by the relation $\equiv$ can in general not be implemented as such in a distributed fashion: In a distributed execution of (N, Π) on a platform which does not provide any synchronization primitive, equivalent local transitions are necessarily executed individually, in some order that an external observer of the execution could observe. Therefore, we replace the synchronization constraint $\equiv$ by a looser constraint which only restricts the possible *orderings* of local transitions belonging to different processes. This is a constraint that can be implemented by adding communication through the primitives offered by the platform (as discussed later).

Definition 11 A *distributed event trace* for (N, Π) is a tuple $(\{\sigma_\pi\}_{\pi\in\Pi}, \prec)$ consisting of:

- a set of local event traces σ_π corresponding to a concurrent event trace $(\{\sigma_\pi\}_{\pi\in\Pi}, \equiv)$;
- a partial order relation $\prec$ defining precedence constraints between local transitions of different processes in these local event traces.

To mimic concurrent event traces, we require the existence of exactly one distributed event trace for each projection $\{\sigma_\pi\}_{\pi\in\Pi}$—that is, for each concurrent event trace. The class of a distributed trace is the set of traces obtained by merging the local traces so as to preserve the order $\prec$. The resulting traces, however, are not traces of N, because local transitions are not merged into global ones (we need $\equiv$ for this): They are in fact traces of the (unfolded) net N^Π.

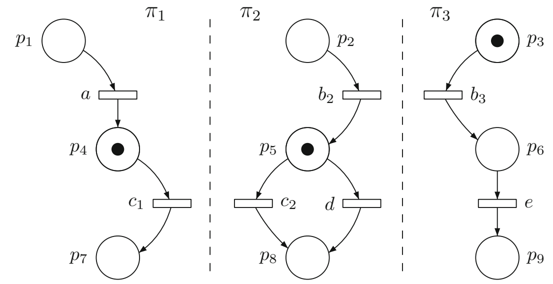

Fig. 4 In the current state, c_1 may be fired according to $\prec_{looseSync}$ and $\prec_{noSync}$ but not $\prec_{fullSync}$

A natural choice of a partial order is derived from the analogy with concurrent traces: Each distributed trace corresponds to a concurrent trace, which also defines a partial order $\prec^{\{\sigma_\pi\}_{\pi\in\Pi}}$. We denote the corresponding partial order on local transitions $\prec_{fullSync}$ because it imposes strong synchronization requirements before and after each set of transitions defining a transition of N. For example, in Fig. 4, it forbids c to start (by firing c_1 or c_2) until a and b are completely terminated.

In practice, $\prec_{fullSync}$ is frequently loosened (e.g., in α-core [25]) by eliminating the need for synchronization after a transition. This is achieved by requiring only that, whenever t precedes t' according to $\prec^{\{\sigma_\pi\}_{\pi\in\Pi}}$, all local transitions of t that precede a local transition of t' (locally) must be terminated before any local transition of t' may take place. Some other processes involved in t may not have terminated t (locally), if they are not involved in t'. For example, c_1 (or c_2) are allowed to fire in the state depicted in Fig. 4, the local transition corresponding to b (that precedes c) must be executed in π_2, but not in π_3, not involved in c. We call this relation $\prec_{looseSync}$. Note that it does not require any buffering as all local processes involved in a transition t must be ready to go before one of them may start the execution of t.

The loosest partial order, $\prec_{noSync}$, imposing no ordering constraints at all, is also a possible choice. It allows some processes to progress even infinitely faster than others, which could in a distributed implementation require unbounded buffering (allowing the slower processes to know about the decisions taken by the faster ones). This is the relation underlying Kahn networks [17].

Note that one could also use partial orders imposing stronger constraints, consisting, for example, in ordering transitions which are concurrent in N, or in ordering the set of local transitions composing a given transition of N. To remain meaningful, such an order should, however, not be allowed to contradict the order $\prec^{\{\sigma_\pi\}_{\pi\in\Pi}}$ imposed by the Petri net. We do not consider such stronger constraints here; we define now only notations for the three ordering constraints discussed above.

Definition 12 Given a distributed Petri net (N, Π) and a concurrent event trace $(\{\sigma_\pi\}_{\pi\in\Pi}, \equiv)$ for it, we define three partial order relations on transitions of (N, Π):

1. $\prec_{\mathit{fullSync}}$ is the partial order on local transitions derived from $\prec^{\{\sigma_\pi\}_{\pi\in\Pi}}$ induced by N.
2. $\prec_{\mathit{noSync}}$ is the empty order;
3. $\prec_{\mathit{looseSync}}$ loosens $\prec_{\mathit{fullSync}}$: For any local transitions t_i and t_j of processes π_i and π_j (where $i \neq j$), $t_i \prec_{\mathit{looseSync}} t_j$ if and only if: (1) $t_i \prec_{\mathit{fullSync}} t_j$ and (2) π_i is involved in the transition of which t_j is part.[3]

Note that these three relations are defined for a given concurrent trace. We use the same notation to denote the function which takes a concurrent trace and returns the corresponding partial order relation $\prec$; we then refer to them as *ordering strategies*.

For any such ordering strategy, we now define what it means for an implementation I—which is characterized by its set of event traces $\mathit{traces}(I)$—to *implement* a distributed Petri net (N, Π).

Definition 13 *I implements* (N, Π) according to an ordering strategy $\prec$ if for each trace $\sigma_I \in \mathit{traces}(I)$ there exists a prefix of a distributed trace $(\{\sigma_\pi\}_{\pi\in\Pi}, \prec)$ of (N, Π) such that σ_I is an element of the class of this distributed trace.

We do not focus on progress properties in this paper; therefore, we allow traces of an implementation to be only prefixes of global traces rather than complete traces. We do, however, discuss strategies, e.g., based on communication, to increase progress in correct implementations—but without formalizing their progress guarantees.

It is essential to note here that in a correct implementation I, one may reach global states which are *not* reachable in N. For example, even for the strongest relation $\prec_{\mathit{fullSync}}$, in a distributed implementation of the Petri net of Fig. 2 state $\{p_3, p_5, p_7\}$ is reachable; this state corresponds to an intermediate state reached during the firing of transition c. It is not reachable in N.

In order to obtain correct implementations of a distributed Petri net, we want to formulate the constraint that an implementation I must satisfy to be considered as correct according to a given ordering strategy, as a (global) control invariant to be enforced on N^Π. Definition 13 requires that to be acceptable, a trace of I must be in the class of some distributed trace for (N, Π). This means that (1) its projection on a set of local traces must be a projection that exists in (N, Π)—as enforced by Definition 11—and (2) it must obey the order constraint imposed by the implementation relation. So far, we have discussed condition (2). We now discuss informally how to break down the first requirement into a set of constraints. In Sect. 4.2, we formalize these constraints and prove that they imply (1).

1. *Transition correctness*, a local constraint: The local order of transitions is preserved. This constraint is guaranteed by any trace generated by the (possibly constrained) execution of the set of local Petri nets N^Π.
2. *Atomicity* For every conflict situation in the Petri net, all processes take the same decision. Together with *transition correctness*, *atomicity* ensures the safety part of *sequential consistency* [21] in the case that synchronizations are considered *symmetric*, i.e., the implementation of a global transition as a set of local transitions does not impose any additional order constraint on them.[4]
3. *Progress* We do not require an implementation to execute complete traces, that is, maximal executions. This would impose some progress, more precisely, deadlock freedom as any execution must be able to "run to completion." Often stronger such *coverage constraints* are used; they express some requirement on how many of the distributed event traces an implementation should be able to reproduce.

   A property related to progress is *fairness* which constrains the possibility to postpone a local transition before inserting it in the trace. The ordering relations $\prec_{\mathit{fullSync}}$ and $\prec_{\mathit{looseSync}}$ express a strong (relative) progress constraint on local transitions of the same global transitions but no constraint on independent parts of (N, Π). We choose not to discuss progress any further here, but the approach we follow also applies to them, and our definitions are general enough to be extendable to progress constraints.

4.2 Implementation relations as control invariants

For each previously defined implementation relation, we need methods to build a correct distributed implementation of a given distributed Petri net. As a starting point, remember that a distributed Petri net (N, Π) implicitly defines an unfolded Petri net N^Π where processes execute independently of each other without any further constraint, which guarantees transition correctness—as the order of local transitions is preserved—but neither atomicity nor any particular global ordering constraint. Our objective in this section is to define a control invariant $\Psi_\prec$ such that the execution of

[3] Remember that there is at most one transition of $T(t_j)$ in a trace.

[4] Note that for transitions that are not symmetric, e.g., because of a data flow, one may add precedence constraints between local transitions of the same global transition—but such precedence relations are not derived from the Petri net.

N^{Π} constrained by $\Psi_{\prec}$ defines a correct implementation of (N, Π) according to $\prec$. Such a control invariant must restrict the enabledness of transitions of N^{Π}, i.e., of local transitions, rather than of (global) transitions of N as in Sect. 3. We postpone to Sect. 5 the question of how to enforce an invariant $\Psi_{\prec}$ in a distributed manner.

Remember that we consider Petri nets without loops such that each transition, local or global, may be fired at most once in a trace. Let us first assume that the predicates defined below are available for a given event trace σ; we show later how to formally define them. They are all parameterized by some (global) transition t, respectively, its local projections t_π.

- $done_{t_\pi}$ holds in a state s if and only if the local transition t_π has been fired in σ before reaching s.
- $done_t$ holds in a state s if and only if $done_{t_\pi}$ holds in s for all processes $p \in proc(t)$, meaning that all processes involved in t have already fired their corresponding local transition.
- if $selected_t$ holds in a state s, then it remains selected forever in σ and no predicate $selected_{t'}$ ever holds for any transition t' in conflict with t in a trace that goes through s. We only allow selected transitions to be fired. This does not guarantee that transition t (all its local transitions) is fired in any trace that goes through s—but should it be the case, this guarantees that $selected_t$ holds and none of the transitions with which t is in conflict is even partially executed.
- $ready_t^\pi$ holds in a state s if and only if t_π is locally enabled in s_π and for all $\pi' \in proc(t)$, either $t_{\pi'}$ is locally enabled in $s_{\pi'}$ or $done_{t_{\pi'}}$ holds in s.
- $strong\text{-}ready_t^\pi$ holds in a state s if and only if $ready_t^\pi$ holds in s and for all global transitions $t' \in T^u$ preceding t in σ, $done_t$ holds in s.

Note that these predicates are defined for a given trace: $done_{t_\pi}$ provides information about the past, while $selected_t$ gives guarantees about the future. Later, we define variables encoding the required information in states of an extended Petri net.

We now formally define the constraints on which the implementation relations of Sect. 4.1 are based, namely atomicity and the ordering constraints defined by $\prec$, as control invariants. Transition correctness is already guaranteed by the local processes driving the distributed implementation.

Definition 14 We define the following control invariants.

- $\Psi_{\prec_{noSync}} = \{(s, t_\pi) \in S^{\Pi} \times T^{\Pi} \mid s \models \exists t \in T(t_\pi).\ selected_t\}$
- $\Psi_{\prec_{looseSync}} = \{(s, t_\pi) \in S^{\Pi} \times T^{\Pi} \mid s \models \exists t \in T(t_\pi).\ selected_t \wedge ready_t^\pi\}$
- $\Psi_{\prec_{fullSync}} = \{(s, t_\pi) \in S^{\Pi} \times T^{\Pi} \mid s \models \exists t \in T(t_\pi).\ selected_t \wedge strong\text{-}ready_t^\pi\}$

Theorem 2 *For an ordering strategy* $\prec \in \{\prec_{noSync}, \prec_{looseSync}, \prec_{fullSync}\}$*, the controlled Petri net* $N_{\Psi_\prec}^{\Pi}$ implements (N, Π) *according to* $\prec$.

Proof We show the correspondence between the control invariants of Definition 14 and the constraints of Definitions 12 and 13.

1. $\Psi_{\prec_{noSync}}$: The partial order $\prec_{noSync}$ is empty, and we only have to show that for every trace of $N_{\Psi_\prec}^{\Pi}$, there exists a global trace of N that projects onto the same local traces. Atomicity is guaranteed because (1) only selected transitions are executed: A local transition must be part of a selected transition to be allowed (2) the definition of the *selected* predicate guarantees that in a same trace, never any conflicting transitions may be selected jointly. This in turn guarantees that a trace of $N_{\Psi_\prec}^{\Pi}$ is a prefix of a distributed trace of (N, Π).
2. $\Psi_{\prec_{looseSync}}$: The partial order $\prec_{looseSync}$ requires that for any local transitions t_π and $t'_{\pi'}$ of distinct processes π and π', if $t'_{\pi'} \prec_{fullSync} t_\pi$ and π' is involved in the transition of which t_π is part in the current prefix of a trace, then $t'_{\pi'}$ must be fired before t_π.
The predicate $ready_t^\pi$ guarantees that for all processes π' involved in t, $t_{\pi'}$ is locally enabled or already executed in s. As a result, all local transitions of π' preceding $t_{\pi'}$ have been fired. Atomicity is guaranteed by the fact that this invariant is clearly stronger than $\Psi_{\prec_{noSync}}$ which implies atomicity (atomicity being a safety property).
3. $\Psi_{\prec_{fullSync}}$: The partial order $\prec_{fullSync}$ is the partial order induced by the current prefix so that whenever t' precedes t and π participates in both transitions, then t' must be completely terminated before any $t_{\pi'}$ may be executed.
The predicate $strong\text{-}ready_t^\pi$ guarantees that for all $\pi' \in proc(t)$, for all local transitions $t'_{\pi'}$ preceding $t_{\pi'}$, $done_{t'_{\pi'}}$ holds in s. Notice that *all* transitions preceding t, also those in which π' does not participate, must be completely terminated before t_π is allowed. It is sufficient that each process checks for transitions preceding t locally that they are terminated globally. The reason is that t must be (or have been) locally enabled in all the processes involved in t, so they have terminated all the work to be done locally before t, in particular the transitions preceding t in which π is *not* involved. In fact, only the transition *immediately* preceding t in π must be checked, earlier transitions have already been checked for executing the transition preceding t. Again, atomicity is guaranteed by the fact that $\Psi_{fullSync}$ is stronger than Ψ_{noSync}.

□

What remains to be done, is to formally define the predicates $done_{t_\pi}$ and $selected_t$.

Definition 15 Define, for each local transition $t_\pi \in T^\Pi$ and global transition $t \in T^u$, the boolean variables $done_{t_\pi}$ and $selected_t$ initialized and updated as follows.

1. All variables $done_{t_\pi}$ are initially set to F. Variables $selected_t$ are initialized so as to guarantee that the set of selected transitions contains no pair of conflicting transitions.
2. For a local transition $t_\pi \in T^\Pi$, the update predicate $f_{t_\pi}(s, v, s', v')$ is such that:
   - $(s', v') \models done_{t_\pi}$
   - for all $t'_{\pi'} \neq t_\pi$, $(s', v') \models done_{t'_{\pi'}}$ if and only if $(s, v) \models done_{t'_{\pi'}}$
   - for all $t', t'' \in T^u$:
     - if $(s', v') \models selected_{t'} \wedge t'\#t''$ then $(s', v') \models \neg selected_{t''}$
     - if $(s, v) \models selected_{t'}$ then $(s', v') \models selected_{t'}$

We do not define anywhere in this paper any particular update *function* for $selected_t$, only the constraints that such a function must satisfy. This choice is discussed further in Sect. 5.2.

Theorem 3 *The variables defined above guarantee that whenever they are set to* T *the corresponding predicates used in Definition 14 hold.*

Proof We only have to prove that the initialization and the update function have the intended effect.

- We need to show that $done_{t_\pi}$ holds in a state s if and only if in the current execution prefix the local transition t_π has been fired before reaching s. As $done_{t_\pi}$ is initially set to false, becomes true immediately after execution of t_π (only) and then holds forever, the above constraint is clearly satisfied.
- We also need to show that if $selected_t$ holds in a state s, then (1) it remains selected forever and (2) for any transition t' in conflict with t, $selected_{t'}$ never holds (1) and (2) are guaranteed by the definition of $selected_t$: Setting $selected_t$ to T is definitive, it is never reset to F; no transition in conflict with a selected one is ever selected (this holds initially and is preserved by any update function). As it is guaranteed by all invariants $\Psi_\prec$ that only selected transitions can ever be executed, the $selected_t$ predicates have the intended meaning. □

5 Knowledge and communication for distributed implementations

In the previous section, we have shown how to obtain a correct implementation (according to three possible definitions) of a distributed Petri net (N, Π) by constraining the Petri net N^Π, obtained by considering the processes in Π as if they were independent Petri nets (and unfolding them if needed), using a centralized controller $(V, \{en_{t_\pi}\}_{t_\pi \in T^\Pi}, \{f_{t_\pi}\}_{t_\pi \in T^\Pi})$. Now, our objective is the definition of a *distributed* controller achieving the same goal: Instead of a unique controller which knows the global state—including the control variables—we need a set of *local* controllers checking the enabling predicates on their *local state*.

We use the notion of *knowledge* [9] to capture the control problem in a distributed setting. Knowledge is the appropriate concept because in order to control a process with only local information, it is not sufficient that some (global) condition is satisfied: The controller has to *know* that it holds. After providing some background information on knowledge, we define an appropriate notion of distributed controller, give a knowledge-based formulation for the distributed implementation problem and discuss possible solutions.

5.1 Knowledge

The *knowledge* of a process π in a local state s_π of a Petri net N is the set of reachable states of N which project onto s_π. Intuitively, it corresponds to the set of global states in which the Petri net may be whenever π is in s_π. We use this concept to characterize which information in Definition 14 needs to be distributed.

Definition 16 For a global state $s \in S$ and a process π, we denote $S_K(\pi, s)$ the set of reachable global states $s' \in reach(N)$ such that s' and s project onto the same local state of π. $S_K(\pi, s)$ is called the *knowledge set* (or simply *knowledge*) of π in s. We also use (equivalently) the notation $S_K(s_\pi)$ and refer to it as the knowledge of π in s_π.

We also introduce a knowledge modality K to relate property satisfaction on local and global states. A process π *knows* a property φ in a state s, denoted $s \models K_\pi \varphi$, if and only if φ holds in all states in $S_K(\pi, s)$. Equivalently, we also use the notation $s_\pi \models K_\pi \varphi$.

Clearly, if π *knows* φ in s then $s \models \varphi$. Note that of course, knowledge sets are potentially extremely large and one does not want to manipulate them directly. We discuss this in the next sections.

Example 4 In Fig. 2, process π_1 knows in its local state p_6 that π_3 is in local state p_5. Indeed, c and d are in conflict so there is no reachable state in which π_1 is in p_6 (meaning c has been fired) and π_3 is not in p_5 (which would mean that d has also been fired).

These definitions extend naturally to extended Petri nets: the knowledge set $S_K(\pi, s, v)$ of a process π in an extended state (s, v) of an extended Petri net N_{ext} is the set of extended reachable states of N_{ext} that project onto the same local state as (s, v).

We now present a result that is essential to increase (and later distribute) knowledge in a (possibly extended) Petri net. Consider two Petri nets N and N_{ext} such that N_{ext} is an extended Petri net whose underlying Petri net is N.

Theorem 4 *For every safety property φ, if $s_\pi \models^N K_\pi \varphi$ then $(s_\pi, v) \models^{N_{ext}} K_\pi \varphi$ for any valuation v, where $\models^N$ and $\models^{N_{ext}}$ denote the satisfaction relation induced by N and N_{ext}, respectively.*

Proof [11] The reason is that N_{ext} has fewer reachable states than N so that the knowledge set of π in (s, v), after projecting out variables, is included in $S_K(s_\pi)$. This proves the property for state predicates φ. The fact that N_{ext} has fewer and shorter traces than N proves knowledge preservation for arbitrary safety properties. □

This means in particular that knowledge computed on a Petri net before unfolding is preserved in the extended Petri net obtained by unfolding. In addition, this result shows that local knowledge may be increased by constraining a system with a controller. We exploited this fact in [3] to achieve local control without communication. Note that this property also holds if N itself is an extended Petri net.

We now define what it means for a property φ to be *stable* in a local state. Stability is used for distributed control as for stable properties knowledge about φ *in the past* guarantees *knowledge of* φ.

Definition 17 A property φ is called *stable* if for every reachable state s such that $s \models \varphi$, all states reachable from s also satisfy φ.

5.2 A knowledge-based distributed controller

We put now everything together and use knowledge for defining a distributed controller. For clarity, we focus in this section on $\prec_{looseSync}$ but all definitions and results can easily be adapted to the other implementation relations that we have proposed. Let us first summarize what we have achieved so far.

Definition 18 Given a distributed Petri net (N, Π), we know how to build an extended Petri net $N_{\Psi_\prec}$ that implements (N, Π) according to $\prec_{looseSync}$, in the sense of Definition 13, namely $N_{\Psi_\prec} = (P^\Pi, T^\Pi, E^\Pi, s_0, V, \{en_{t_\pi}\}_{t_\pi \in T^\Pi}, \{f_{t_\pi}\}_{t_\pi \in T^\Pi})$ such that

- $(P^\Pi, T^\Pi, E^\Pi, s_0) = N^\Pi$—remember that N and N^Π have different places and states only because of unfolding (although s_0 also is the initial state of N^Π) and differ on their transitions (a transition $t \in T$ in N corresponds in N^Π to as many transitions t_π as there are processes involved in t, and possibly more in presence of loops) and their relation E between places and transitions (directly resulting from the splitting of transitions);
- all variables are boolean (i.e., $\mathcal{V} = \{\mathtt{T}, \mathtt{F}\}$) and initially set to $\mathtt{F}$ (except that some $selected_t$ must hold as explained in Definition 15):

  $$V = \{selected_t\}_{t \in T^u} \cup \{done_{t_\pi}\}_{t_\pi \in T^\Pi}$$

- for each transition $t_\pi \in T^\Pi$:
  - en_{t_π} holds in state $s \in S^\Pi$ for values $v \in \mathcal{V}$ if and only if $(s, t_\pi) \in \Psi_{\prec_{looseSync}}$, i.e.,

    $(s, v) \models \exists t \in T^u . (selected_t \wedge ready_t^\pi)$

    Then, expanding the definition of $ready_t^\pi$ we obtain

    $$(s, v) \models \exists t \in T^u . (selected_t \wedge \forall \pi' \in proc(t) . \\ ([t_{\pi'}\rangle \vee done_{t_{\pi'}}))$$
  - according to Definition 15, the update predicate $f_{t_\pi}(s, v, s', v')$ is such that:
    - $(s', v') \models done_{t_\pi}$
    - for all $t'_{\pi'} \neq t_\pi$, $(s', v') \models done_{t'_{\pi'}}$ if and only if $(s, v) \models done_{t'_{\pi'}}$
    - for all $t', t'' \in T^u$:
      - if $(s', v') \models selected_{t'} \wedge t' \# t''$ then $(s', v') \models \neg selected_{t''}$
      - if $(s, v) \models selected_{t'}$ then $(s', v') \models selected_{t'}$

$(V, \{en_{t_\pi}\}_{t_\pi \in T^\Pi}, \{f_{t_\pi}\}_{t_\pi \in T^\Pi})$ is called the *centralized controller* of (N, Π).

In this centralized controller, the enabling conditions to fire a local transition t_π are global conditions or at least depend on the local state of several processes (those involved in transition t). In a distributed controller, we want to define a set of local controllers such that the local controller of π can decide to fire t_π based on its local state and the local state of π.

Definition 19 Given a distributed Petri net (N, Π), a controller for N^Π (enforcing Ψ) of the form $(V, \{en_{t_\pi}\}_{t_\pi \in T^\Pi}, \{f_{t_\pi}\}_{t_\pi \in T^\Pi})$ is said to be *distributed* if it is such that

1. the set of variables V is partitioned into local variable sets V_π, one per process;
2. for each transition t_π,
   - the guard en_{t_π} is expressed as a condition $en_{t_\pi}^\pi$ depending only on local variables in V_π and the local state s_π;
   - the update f_{t_π} is similarly a local update $f_{t_\pi}^\pi$ that modifies only the corresponding local state and variables.

Note that the local controller of a process π cannot directly evaluate the condition en_{t_π} of the centralized controller in

Definition 18 because this requires global information about the execution of the Petri net. $K_\pi en_{t_\pi}$, however, is a condition that can be evaluated on the (extended) local states of π. Furthermore, we have already seen that $s \models K_\pi en_{t_\pi}$ implies $s \models en_{t_\pi}$, meaning we can guarantee[5] a correct implementation of (N, Π) if we replace en_{t_π} by $K_\pi en_{t_\pi}$. Therefore, we propose a first version of a distributed controller of (N, Π) obtained by replacing all variables by their local knowledge counterparts. To do so, we need additional variables $k_\pi [t_{\pi'}\rangle$ because the condition en_{t_π} requires to evaluate also $[t_{\pi'}\rangle$ (local enabledness) for other processes π' participating in t. The centralized controller can evaluate this predicate on the current global state, but a local controller for π must evaluate this predicate locally.

Theorem 5 *Given a distributed Petri net* (N, Π)*, define for each process* $\pi \in \Pi$*:*

1. *a set of local variables,*

   $$V^\pi = \{k_\pi selected_t\}_{t \in T^u} \cup \{k_\pi [t_{\pi'}\rangle, k_\pi done_{t_{\pi'}}\}_{t_{\pi'} \in T^\Pi}$$

2. *as in the centralized case, all variables are initialized with* F *except for some predicates* $k_\pi selected_t$ *which are supposed to be set consistently;*
3. *for each transition* $t_\pi \in T^\Pi$*,*
   - $en^\pi_{t_\pi} = \exists t \in T^u . (k_\pi selected_t \wedge \forall \pi' \in proc(t) . (k_\pi [t_{\pi'}\rangle \vee k_\pi done_{t_{\pi'}}))$*, that is, the same expression as for the centralized controller but using local knowledge variables.*
   - *the update predicate* $f^\pi_{t_\pi}(s, v, s', v')$*—where* s, s' *are local states—is exactly as in the centralized controller for locally available information.*
     The new variables are updated as follows:
     - $k_\pi [t_\pi\rangle$ *is set to* T *if* t_π *is locally enabled in* s' *and to* F *otherwise;*
     - *all other variables* $k_\pi [t'_{\pi'}\rangle$ *are set to* F.

If $k_\pi selected_t \Rightarrow K_\pi selected_t$*, then* $(N^\Pi, \bigcup_{\pi \in \Pi} V^\pi, \{en^\pi_{t_\pi}\}_{t_\pi \in T^\Pi}, \{f^\pi_{t_\pi}\}_{t_\pi \in T^\Pi})$ *implements* (N, Π) *w.r.t.* $\prec_{looseSync}$.

Proof To prove the correctness of this distributed controller we show that a transition can be fired *only* if it can also be fired by the centralized controller. This amounts to proving that $\forall t_\pi \in T^\Pi .\ en^\pi_{t_\pi} \Rightarrow en_{t_\pi}$. We do this by proving that $en^\pi_{t_\pi} \Rightarrow K_\pi en_{t_\pi}$. In fact, it is sufficient to prove this property for individual knowledge variables, which we achieve by induction using the update predicate.

We do not need any proof for $k_\pi selected_t$ as the property is assumed to hold (see below why we proceed this way). We now only have to prove that $k_\pi done_{t_{\pi'}} \Rightarrow K_\pi done_{t_{\pi'}}$ and $k_\pi [t_{\pi'}\rangle \Rightarrow K_\pi [t_{\pi'}\rangle$. In fact, only variables $k_\pi done_{t_\pi}$ and $k_\pi [t_\pi\rangle$ are ever set to T and they both correspond to properties which can entirely be determined from the local state of π, which completes the proof. □

Like for the centralized controller, the assignment of the $k_\pi selected_t$ predicates is left nondeterministic with the assumption that it guarantees that two processes cannot take contradictory decisions (by selecting conflicting transitions). The reason is that some conflict resolution mechanism is required for achieving this in an actual implementation and that we want here a specification able to capture all, or at least a large number of actual implementations without imposing any specific conflict resolution mechanism. One possible strategy is to statically decide which local process has the right to choose among a set of conflicting transitions. Another (dynamic) option is to use an agreement protocol; there exists a number of them, chosen depending on the platform properties and the needs of the application described by the Petri net.[6]

Although correct, this first distributed controller is perfectly useless because variables representing local properties of other processes are never set to T; therefore, after an initial transition, only transitions with a single participating process can ever be fired. This is why we need to enhance this controller with *communication* capabilities. We suppose that the platform offers processes some means to exchange information and formalize how communication can increase the knowledge of processes about properties of other processes and therefore increase progress of an implementation of (N, Π). At this point, it becomes clear that knowledge is an appropriate concept to characterize information exchange.

A *communication strategy* specifies which information may be communicated between local controllers of processes and when this can be done. It may require additional variables in the controllers, but it cannot alter the processes themselves.

Definition 20 A *communication strategy* com consists of:

- a set of variables V_{com} partitioned into variable sets V^π_{com} associated with processes and a variable set V^{plat}_{com} associated with the platform.
- a communication update predicate f_{com} such that in an extended global state (s, v) of the controlled Petri net $N^\Pi_{\Psi_\prec}$, two different types of updates are possible:

[5] This is true because Ψ is a safety constraint; one may, however, lose progress.

[6] Note that the problem statement as given by the centralized controller is completely symmetric; in case of circular conflicts—in which the size of the conflict set may not be reduced by a set of local arbitrations—some asymmetry must be introduced statically (for example, using priorities) to achieve conflict resolution, or the protocol must rely on some statistical decision (such as, for example, in Ethernet).

1. Communication from process π to the platform: $f_{\text{com}}(s, v, v_{\text{com}}, v', v'_{\text{com}})$ where
   - only communication variables of the platform may be updated: $v' = v$ and only variables in V_{com}^{plat} may be updated;
   - the new values of the platform communication variables depend only on local information of π (local state s_π, controller state v_π and communication state v_{com}^{π}).
2. Communication from the platform to process π: $f_{\text{com}}(s, v, v_{\text{com}}, v', v'_{\text{com}})$ where
   - only controller and communication variables of π may be updated;
   - the new values only depend on local information of π and platform variables.

A communication strategy is *correct* if it guarantees that $en_{t_\pi}^{\pi} \Rightarrow K_\pi en_{t_\pi}$ is preserved, as well as the correctness of all knowledge variables (i.e., $k_\pi \varphi \Rightarrow K_\pi \varphi$).

Of course, this definition must be adapted to reflect the exact capabilities of the platform in use. For example, the minimal communication primitive that may be offered by virtually any platform, if necessary with the help of some protocol, is reliable transmission of data with unbounded transmission delay. This would be formalized by restricting how communication variables are written and read.

Example 5 Consider a communication strategy where all processes communicate right after firing a local transition about their new extended local state, information which the platform will eventually transmit to all processes needing it. Again, we also assume that the $k_\pi selected_t$ predicates are handled correctly.

Because the predicates $done_{t_{\pi'}}$ are stable, upon receiving a message $done_{t_{\pi'}}$ from process π', a process π is able to update its own variable $k_\pi done_{t_{\pi'}}$ to $\mathtt{T}$. This, however, does not apply to $[t_{\pi'}\rangle$ which is *not* stable. We therefore introduce an additional variable, called $k_\pi aft\text{-}en_{t_{\pi'}}$, to denote that $t_{\pi'}$ has been enabled in the current execution prefix. This is useful for our purpose because $k_\pi aft\text{-}en_{t_{\pi'}}$ is stable and $selected_t \wedge aft\text{-}en_{t_{\pi'}} \Rightarrow ([t_{\pi'}\rangle \vee done_{t_{\pi'}})$: In a local state of π' in which $[t_{\pi'}\rangle \wedge selected_t$ holds, only this transition may be fired, and after that $done_{t_{\pi'}}$ holds and is stable. This in turn may be used to establish that $ready_t^{\pi}$ or $strong\text{-}ready_t^{\pi}$ holds.

Note that with a platform offering no more than the required reliable data transmission with unbounded delay, this communication strategy is sufficient to guarantee the progress allowed by the $select_t$ predicates.

Another strategy for increasing local knowledge of processes is to use statically defined knowledge that may be derived from general system settings, from a priori state space exploration, from structural information, or by a design decision. This does not necessarily avoid the corresponding communication—because the fact that π *knows* φ does not guarantee that π' *knows that* π *knows* φ, but it may allow π to progress faster as π does not need to wait for this communication to happen.

Altogether, we have now defined a nondeterministic specification of what a distributed controller should guarantee. Let us summarize now what remains to be done to obtain an actual distributed controller. One still needs:

1. a strategy for setting the $selected_t$ predicates—which may be statically predefined or obtained by means of an arbitration protocol;
2. a communication strategy adapted to each given platform;
3. a concrete representation for the required knowledge: One may use other controller variables than those defined here as long as they allow encoding the required knowledge.

Additionally, remember that we suppose a potentially infinite set of variables and transitions that must be distinguished. We made this choice to simplify the presentations and the proofs. Of course, any concrete algorithm has only a finite amount of memory for storing knowledge, and therefore, it may only distinguish a finite set of (relevant) transitions. How much information may be relevant at any point of time (and therefore needs to be stored) depends on the chosen implementation relation and on how much of the freedom allowed by this relation one wants to preserve in the implementation. For example, $\prec_{fullSync}$ forces processes to stay closely synchronized: Each process participates in at most one transition that has been started but not yet completed. Only information about such transitions *must* be available, and only information about future decisions is potentially useful. In contrast, $\prec_{noSync}$ allows some processes to run arbitrarily faster than others so that the set of transitions to store is not bounded a priori by $\prec_{noSync}$.

6 Property preservation for achieving efficient control and distribution

Our initial problem was to: (1) constrain a specification N with some control invariant Ψ; (2) validate it for some requirement φ; *and* (3) implement the constrained Petri net (N, Ψ) on a distributed platform such that the distributed implementation also satisfies Ψ and φ.

In [11], as summarized in Sect. 3, we have defined a controller that constructs for a given specification N and a control property Ψ a Petri net N_Ψ that restricts N to executions satisfying Ψ. We mention there that a distributed implementation of the constrained Petri net N_Ψ may be obtained by a *layered* *approach* using (1) model checking and some well-chosen additional synchronizations to build N_Ψ and then (2) any algorithm achieving distribution of a Petri net, for example, α-core.

On the other hand, Sects. 4 and 5 suggest an alternative approach. Indeed, we have shown how to express the problem of executing a global specification N in a distributed manner as a problem of (1) representing the distribution problem as a (centralized) control problem and then (2) transforming a centralized controller into a distributed one. This means that to solve the original distributed control problem, we end up with two kinds of control invariants to be distributed: The original control invariant Ψ and the control invariant $\Psi_\prec$ characterizing the implementation relation we want to guarantee. This suggest the possibility to jointly distribute their conjunction. This is the *integrated approach* applied in [14] in an ad hoc manner. We now discuss and compare these two approaches to distributed control.

6.1 Distribution and control

To distribute the control invariants Ψ and $\Psi_{\prec_{looseSync}}$ jointly, as suggested in the integrated approach, amounts to strengthening locally the enabling condition $en^{\pi}_{t_\pi}$ of Theorem 5 so that a local state s_π must additionally *know* that t_π is authorized by Ψ in s_π to allow t_π to be fired. Note here that "t_π *is authorized by* Ψ *in* s_π" does not have a clear meaning because Ψ is defined on global states and global transitions. In contrast, what we need here is a control invariant that applies to N^Π rather than N.

Interestingly, the issue of how Ψ extends to N^Π is also implicitly there in the layered approach. We illustrate this on priority constraints. Figure 5 shows again our running example assuming a priority order $\{g \ll d\}$. In N, this constraint is trivially enforced as g is enabled only after termination of d. Therefore, we obtain $N_\Psi = N$. However, for all the considered implementation relations, the state $\{p_3, p_4, p_8\}$ is reachable. In that state d is *partially executed*, and both g *and* d_1 may be fired. Note that $\prec_{fullSync}$ prevents g from being fired as the controller of π_3 waits for d to finish before allowing π_3 to proceed further. In contrast, $\prec_{looseSync}$ and $\prec_{noSync}$ do not imply such a requirement and allow g to be fired in this state.

Now, whether this is acceptable or not depends on how priorities are interpreted in a distributed setting: Does $g \ll d$ mean that g should not start when d is enabled? Or that g should not start while d is enabled or ongoing? Or that it should not be possible that g is still ongoing when d gets enabled? Lifting this ambiguity requires to extend the interpretation of Ψ—which is defined on N—to N^Π, and we denote this extended interpretation Ψ^Π. Given Ψ^Π, one must prove that the distributed implementation of N_Ψ according to $\prec$ (as in Theorem 5) guarantees by construction that Ψ^Π is satisfied on the resulting distributed system.

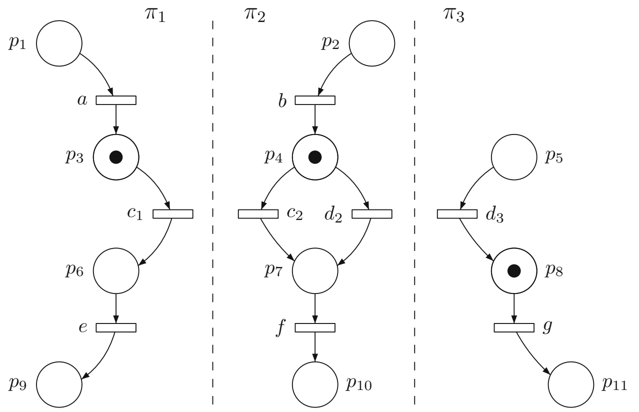

Fig. 5 Distributed Petri net with priority order $\{g \ll d\}$

To summarize, in order to apply this layered verification strategy—i.e., construct N_Ψ, verify φ on N_Ψ, and then, forgetting Ψ and φ, distribute N_Ψ according to $\prec$—we must prove that the distribution algorithm preserves both φ and Ψ, or more precisely that $\prec$ preserves φ^Π and Ψ^Π. This may nevertheless seem advantageous compared to the integrated approach, especially in the case where $\prec$ preserves Ψ^Π. The reason is that N_Ψ has fewer reachable states than N, and therefore, local states may have more knowledge in N_Ψ (Theorem 4). In fact, this knowledge may also be used in the integrated approach, as we discuss in the next sections. This is a strong argument in favor of the integrated approach over the layered one, because the former may require less, more fine-grained communication between processes, through the primitives available on the platform, than the latter, which resorts to synchronizations, the only communication mechanism available for Petri nets.[7]

6.2 Preservation of knowledge

Our initial motivation for defining a knowledge-based formulation of the distribution problem was that this would offer us the possibility to use statically available knowledge calculated on the Petri net for optimizing a distributed controller. In particular, we could derive from a generic solution, for example, the one obtained by using the α-core protocol, a more efficient distributed solution for any given application, simply by exploiting statically available knowledge to avoid sending a message when the destination does not need it, or to avoid waiting for messages transmitting already available knowledge.

[7] This makes clear that for a different platform and a different high level specification paradigm, the response could be different. Our aim here is to propose criteria for choosing the best approach in any given setting.

As mentioned earlier, to follow this approach, it must be guaranteed that the knowledge computed in the centralized Petri net is preserved in the distributed system. We now show some preservation results for knowledge properties contributing to the local enabledness predicates. The key concept for this is stability.

Predicates required for evaluating Ψ. As we have seen, such predicates may not be preserved by distribution, depending on the choice of $\prec$ and of the interpretation of these predicates in a distributed setting. Still, some knowledge may be preserved. For example, consider priority orders, $\prec_{fullSync}$ and the following interpretation of $\Psi_{\ll}$ for our example of Fig. 5: "g has highest priority in s" means "d is already completely terminated in s or it is not possible to reach a state in which d is firable before g is completely terminated." Then, static knowledge computed on the original Petri net, namely that d always precedes g, can be combined with the definition of $\prec_{fullSync}$ to establish that no control is needed to enforce $g \ll d$. If one wants to use a looser implementation relation, for example, $\prec_{looseSync}$, we may use the mixed approach: In general, synchronization after the execution of a transition is not required, but the constraint added to guarantee the priority rule would force synchronization after d.

Predicates $selected_t$ used to guarantee atomicity. One may determine statically[8] that $selected_t$ holds in a global state if there is no transition in conflict with t, and therefore no transition which could be chosen *instead of* t. In a global state s offering a choice between several transitions, s may be extended so as to statically select one of them, or so as to allow one of the processes to do the selection. Because $selected_t$ is stable in $N^{\Pi}_{\Psi_\prec}$, knowledge is preserved from the centralized to the distributed Petri net: If a process π knows $selected_t$ in a local state of (N, Π), then it also knows $selected_t$ in the corresponding local state of $N^{\Pi}_{\Psi_\prec}$.

Predicates contributing to global readiness. These are the predicates required for synchronizing *before* and/or *after* a transition, e.g., for the implementation relations $\prec_{fullSync}$ and $\prec_{looseSync}$. We express such synchronization constraints by means of the predicates $done_{t_\pi}$ that hold if process π has at least progressed this far. These are stable predicates: Once a process *knows* $done_{t_\pi}$, it knows this forever. Nevertheless, these predicates cannot be established from statically computed knowledge: They require communication to allow processes to learn about the progress of others. Despite of this difficulty, static knowledge may be useful for establishing enabledness at the Petri net level: We have already shown that $[t_\pi\rangle$ is not a stable property, but $\neg[t_\pi\rangle$ may be stable (enough) to allow π to *know* $\neg[t_\pi\rangle$ in some local state, that is, to exclude t a priori from the set of transitions from which to select the next step.[9]

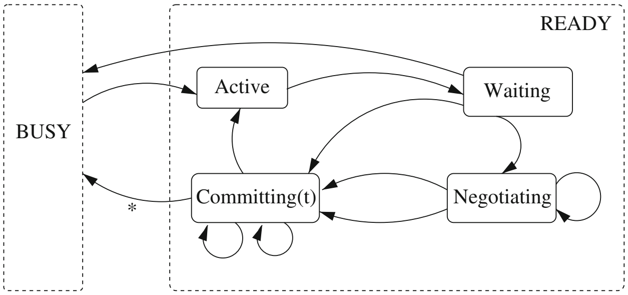

Fig. 6 State diagram of the algorithm presented in [14]

We have explained in Sect. 5.2 that whenever one of the above-mentioned knowledge predicates is known—either from an analysis of the global state space or as a consequence of earlier communication—this may be used to avoid unnecessary additional communication. A consequence of the preservation results sketched above is the possibility to derive some knowledge in the distributed implementation from knowledge in the original (constrained) Petri net. This is very useful because the state space of N_Ψ is usually much smaller than the one of the distributed implementation.

6.3 Illustrating example

Let us now discuss the added value of the reasoning presented in the previous sections on an example. In [14], an algorithm is presented which builds a distributed implementation of a prioritized specification for systems with binary synchronizations. It is inspired by α-core but differs from it by the fact that it handles specifications with (global) priorities and implements a less static conflict resolution. In both algorithms, the platform is assumed to ensure reliable and order-preserving transmission of messages. The precise organization of the protocol of [14] is beyond the scope of this paper, but an abstract general view of the main steps of the protocol phase is shown in Fig. 6. This protocol takes place in each local state of each process, that is, all the states represented in the diagram correspond to the same place in the centralized Petri net. The transition in the diagram labeled $*$ corresponds to the start of the execution of a local transition (which takes place in the *Busy* state and leads to a state change in the Petri net). Other transitions represent sending and receiving of messages expressing information about local enabledness of transitions, as well as commitment of a process to a given transition t—the last step before achieving $selected_t$, unless the other process rejects t.

[8] Note that this does not hold if $selected_t$ is also used for guaranteeing fairness properties.

[9] This is the knowledge exploited in [22].

We can use the results of the previous sections in two ways here. First, we now have a generic formal support for proving the correctness of the algorithm under consideration. Indeed, transition correctness, atomicity, and synchronization as defined in Sect. 4.1 are satisfied if and only if a process π may take the transition labeled $*$, that is, fire locally a transition t of the original Petri net, only when it has the required *knowledge* for it. For example, $selected_t$ holds in that case because both processes π and π' participating in transition t have committed to it. The protocol obliges π' to wait for the consent of π, and π therefore, *knows* that π' cannot fire any other transition than t. In other words, in all the global states consistent with the local view of π in this local state, $selected_t$ holds.

Second, once formalized the knowledge properties associated with each local state, we can use them in combination with the properties obtained by static analysis of the centralized Petri net. That is, for a local state s_π of π, all states of the associated communication protocol are enriched with (preserved) local knowledge of π in s_π. Based on this, π may not have to wait for all messages to arrive before progressing, as it now has enough knowledge to fire a transition without them. In addition, if messages are clearly identified as questions and answers—as is the case here and often in such protocols—then π may in such case omit some question messages as it does not need them. However, this forbids the analysis of knowledge properties to rely on question messages.

The resulting clear separation between the generic protocol and its implementation for a given centralized Petri net seems promising as this approach is scalable (the distributed system as a whole is never analyzed) and still understandable: The centralized Petri net and the protocol are analyzed separately, then used together in a correct-by-construction manner by optimizing its performance.

7 Conclusion

In this paper, we discuss two strategies for handling control and distribution jointly. A layered approach, advocated in [11], which consists in solving first the original control problem on the centralized specification, then distributing the obtained controller and finally distributing the resulting constrained system agnostic of the original control problem. We propose here a second, integrated approach which solves the control and distribution problems jointly. We have shown that this approach may avoid unnecessary synchronizations while allowing us to exploit the same knowledge as in the layered approach.

We have presented the problem of achieving a distributed execution of a centralized specification as a control problem, and we have proposed a knowledge-based approach to derive a specification of a distributed controller achieving that goal. We have also discussed how to achieve the knowledge needed by the controllers of individual localities by communication or by statically available knowledge, which could be derived from platform or application domain induced conditions or obtained by a priori static analysis of the centralized specification—under the condition that this knowledge is preserved in the distributed context. One interesting side effect of this work is to make explicit some underlying assumptions made by previous knowledge-based work on distributed control.

Note that we only considered abstract specifications without taking into account data-flow and non-functional aspects, such as timing. We made quite minimal assumptions on the properties provided by the platform. Extending the approach to take into account data and data transformation is almost straightforward: Data are distributed in almost the same way as demonstrated here for knowledge predicates, and data-dependent enabling conditions are treated like an additional constraint imposed on the local enabling conditions. For platforms providing stronger guarantees, for example, bounded communication delays, it is possible to deduce stronger knowledge properties from communication actions which in turn can be used to establish execution time bounds.

References

1. Bagrodia, R.: Process synchronization: design and performance evaluation of distributed algorithms. IEEE Trans. Softw. Eng. **15**(9), 1053–1065 (1989)
2. Basu, A., Bensalem, S., Peled, D., Sifakis, J.: Priority scheduling of distributed systems based on model checking. In: Proceedings of CAV'09, vol. 5643 of LNCS, pp. 79–93. Springer (2009)
3. Bensalem, S., Bozga, M., Graf, S., Peled, D., Quinton, S.: Methods for knowledge-based controlling of distributed systems. In: Proceedings of ATVA'10, vol. 6252 of LNCS, pp. 52–66. Springer (2010)
4. Bensalem, S., Bozga, M., Quilbeuf, J., Sifakis, J.: Knowledge-based distributed conflict resolution for multiparty interactions and priorities. In: Proceedings of FMOODS-FORTE'12, vol. 7273 of LNCS, pp. 118–134. Springer (2012)
5. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages twelve years later. Proc. IEEE **91**(1), 64–83 (2003)
6. Boussinot, F., de Simone, R.: The esterel language. In: Proceedings of the IEEE, Special Issue on Synchronous Programming, vol. 79, pp. 1293–1304 (1991)
7. Caspi, P., Girault, A.: Execution of distributed reactive systems. In: Proceedings of Euro-Par'95, vol. 966 of LNCS, pp. 15–26. Springer (1995)
8. Chu, P.M., Liu, M.T.: Synthesizing protocol specifications from service specifications. In: Proceedings of Computer Networking Symposium, pp. 173–182. IEEE (1988)
9. Fagin, R., Halpern, J.Y., Vardi, M.Y., Moses, Y.: Reasoning about knowledge. MIT Press, Cambridge, MA (1995)
10. Genrich, H.J., Lautenbach, K.: System modelling with high-level Petri nets. Theor. Comput. Sci. **13**, 109–136 (1981)

11. Graf, S., Peled, D., Quinton, S.: Achieving distributed control through model checking. In: Proceedings of CAV'10, vol. 6174 of LNCS, pp. 396–409. Springer (2010)
12. Graf, S., Peled, D., Quinton, S.: Monitoring distributed systems using knowledge. In: Proceedings of FMOODS-FORTE'11, vol. 6722 of LNCS, pp. 183–197. Springer (2011)
13. Graf, S., Quinton, S.: Knowledge for the distributed implementation of constrained systems. In: 10th International Conference on Integrated Formal Methods, iFM 2013, Turku, 10–14 June. Proceedings, vol. 7940 of LNCS, pp. 77–93. Springer (2013)
14. Graf, S., Quinton, S.: Building distributed controllers for systems with priorities. J. Log. Algebr. Program. **80**(3–5), 194–218 (2011)
15. Gotzhein, R., von Bochmann, G.: Deriving protocol specifications from service specifications including parameters. ACM Trans. Comput. Syst. **8**(4), 255–283 (1990)
16. Halpern, J.Y., Fagin, R.: Modelling knowledge and action in distributed systems. Distrib. Comput. **3**(4), 159–177 (1989)
17. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress, pp. 471–475 (1974)
18. Kant, C., Higashino, T., von Bochmann, G.: Deriving protocol specifications from service specifications written in lotos. Distrib. Comput. **10**(1), 29–47 (1996)
19. Katz, S., Peled, D.: Verification of distributed programs using representative interleaving sequences. Distrib. Comput. **6**(2), 107–120 (1992)
20. Katz, G., Peled, D., Schewe, S.: Synthesis of distributed control through knowledge accumulation. In: Proceedings of CAV'11, vol. 6806 of LNCS, pp. 510–525. Springer (2011)
21. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979)
22. Laurie Ricker, S.: Know means no: incorporating knowledge into discrete-event control systems. IEEE Trans. Autom. Control **45**(9), 1656–1668 (2000)
23. Laurie Ricker, S., Rudie, K.: Knowledge is a terrible thing to waste: using inference in discrete-event control problems. IEEE Trans. Autom. Control **52**(3), 428–441 (2007)
24. Lin, F., Wonham, W.M.: Decentralized supervisory control of discrete-event systems. Inf. Sci. **44**(3), 199–224 (1988)
25. Pérez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurr. Pract. Exp. **16**(12), 1173–1206 (2004)
26. Peterson, J.L.: Petri Net Theory and Modeling of Systems. Prentice Hall, Englewood Cliffs (1981)
27. Probert, R.L., Saleh, K.: Synthesis of communication protocols: survey and assessment. IEEE Trans. Comput. **40**(4), 468–476 (1991)
28. Reisig, W.: Petri Nets, an Introduction. EATCS Monographs on Theoretical Computer Science. Springer, Berlin (1985)
29. Rudie, K., Wonham, W.M.: Think globally, act locally: decentralized supervisory control. IEEE Trans. Autom. Control **37**(11), 1692–1708 (1992)
30. von Bochmann, G., Gotzhein, R.: Deriving protocol specifications from service specifications. In: Proceedings of SIGCOMM'86, pp. 148–156. ACM (1986)
31. Wong, K.C., Wonham, W.M.: Modular control and coordination of discrete-event systems. Discrete Event Dyn. Syst. **8**(3), 247–297 (1998)
32. Yamaguchi, H., El-Fakih, K., von Bochmann, G., Higashino, T.: Deriving protocol specifications from service specifications written as predicate/transition-nets. Comput. Netw. **51**(1), 258–284 (2007)
33. Yoo, T.-S., Lafortune, S.: A general architecture for decentralized supervisory control of discrete-event systems. Discrete Event Dyn. Syst. **12**(3), 335–377 (2002)
34. Yamaguchi, H., Okano, K., Higashino, T., Taniguchi, K.: Synthesis of protocol entities' specifications from service specifications in a petri net model with registers. In: Proceedings of ICDCS'95, pp. 510–517 (1995)
35. Zafiropulo, P., West, C.H., Rudin, H., Cowan, D.D., Brand, D.: Towards analyzing and synthesizing protocols. IEEE Trans. Commun. **COM–28**(4), 651–661 (1980)

Susanne Graf received her Ph.D. in Computer Science in 1984 from Grenoble Polytechnical Institute (INPG). Currently, she is a "Directeur de Research" at CNRS in the Verimag laboratory in Grenoble. Her research interests include theories, algorithms and tools for modeling and verification and their application to the design and certification of embedded and real-time as well as mixed-criticality systems. Presently she is focussing on knowledge and contract-based methods to achieve scalability. Some of her main contributions are in the domain of abstraction where in particular, she has proposed together with Hassen Saidi a method called predicate abstraction. She has been the Organisation Chair of ETAPS 2002 in Grenoble, she has been the PC chair of TACAS 2000, SPIN 2004, ATVA 2006, and FORTE 2015. She is on the editorial board of Springer's STTT journal and has participated in a large number of Program Committees.

Sophie Quinton is junior researcher at Inria Grenoble—Rhône-Alpes. She received her M.Sc. degree in Computer Science from the University of Rennes, France, in 2005 and her Ph.D. degree from the University of Grenoble, France, in 2011. She was a student at the École Normale Supérieure de Cachan and later a graduate research assistant at the VERIMAG laboratory and a postdoc in the Embedded System Design Automation group of the Institute of Computer and Network Engineering at TU Braunschweig. Her research interests include real-time schedulability analysis, contract-based design and verification of systems of components, and discrete control of distributed systems. She has been a PC member of a dozen conferences and workshops and has published in conferences such as CAV, RTSS, DATE, ECRTS, etc.