ON PROVING PROPERTIES OF CONCURRENT ALGORITHMS FOR DISTRIBUTED CSPs

Markus Hannebauer

GMD — German National Research Center for Information Technology
Research Institute for Computer Architecture and Software Technology (FIRST)
Kekuléstr. 7, D-12489 Berlin, Germany
markus.hannebauer@gmd.de

ABSTRACT

Because of social, economical and technical reasons, some years ago the classical CSP model has been extended to the model of Distributed Constraint Satisfaction Problems (DCSPs) in which several computational agents try to solve a connected CSP collectively. Many recent systems tackling this problem lack a theoretically founded specification and safety or liveness property proofs. This may be due to the difficulty of modeling and verifying concurrently running threads and their interaction. In this article we will briefly present a general approach to the modeling and verification of concurrent algorithms tailored to DCSP solving and based on algebraic Petri nets. We will present a realistic case study on distributed agreement finding and prove according termination and consistency properties.

1. INTRODUCTION

The model of *Constraint Satisfaction Problems* (CSPs) and its according methods in information propagation and search have proven to be well-suited to solve common practical problems, especially when the problems can be modeled using constraint systems over finite domains. Because of social, economical and technical motivation, some years ago the classical CSP model has been extended to the model of *Distributed Constraint Satisfaction Problems* (DCSPs) in which several computational agents try to solve a connected CSP collectively. Since then, researchers from the constraint programming community as well as from the intelligent agents community have proposed various algorithms for coordinating concurrently running agents in a collaborative CSP solving process (see Section 2).

Though referential realizations of concurrent systems in production planning, scheduling and control are various, systems in particular from the agent community lack theoretically founded specifications and safety or liveness property proofs. It is surely much more difficult to specify and verify a set of concurrently running threads and their interactions than specify/verify a single thread of control. Nevertheless, several researchers have contributed to this research area, using as different calculi as temporal logics [3], compositional verification [5] or automata and Petri nets [2, 4, 7].

In this article we will briefly present a general approach to the modeling and verification of concurrent algorithms specialized to DCSP solving and based on the intuitive means of algebraic Petri nets. We will start in Sect. 2 with a short review of related work in DCSP solving and according algorithms. In Sect. 3 we will present principles for modeling DCSP algorithms and apply them to a case study on agreement finding. The main Sect. 4 identifies some desirable properties of DCSP algorithms, introduces the means of invariants, a concurrent deduction operator and proof graphs and proves safety and liveness properties of the agreement finding algorithm. Conclusions and remarks on future work finish our article.

2. DCSPs AND ALGORITHMS

There are several models for DCSPs and even more algorithms for solving them. A good, though a little outdated overview can be found in [6]. The authors identify four basic characteristics of DCSP algorithms: centralized or decentralized control, shared or separated search space, message-passing or shared memory and termination. Another, orthogonal classification distinguishes variable-based approaches (in which every agent cares for a subset of variables), domain-based approaches (in which every agent cares for a subset of values for common variables) and function-based approaches (in which costly computations in centralized CSP solving are distributed to speed them up). In this article we will only consider decentralized variable-based algorithms on a shared search space using asynchronous message-passing, since this is the closest model to socially embedded and naturally distributed CSPs. We call these algorithms *concurrent DCSP algorithms*.

There seem to be at least two major strategies towards solving DCSPs. The first considers mainly binary, often weakly structured academic CSPs and tries to distribute successful sequential algorithms for constraint propagation and search using constraint representations based on no-good sets. The work of Yokoo, Durfee, Ishida and others [15, 16] is very important in this area, namely their Asynchronous Backtracking and Weak-Commitment Search algorithms. While they concentrated on the search part of DCSP solving, others [17, 8] have investigated distributed derivatives of sequential constraint propagation techniques, namely DistAC and DisAC-4. Researchers following the second strategy also employ knowledge gained from classical CP, like backtracking, backjumping and so on, but apply it usually to more structured practical problems with complex n-ary constraints employing specific heuristics. The research work of Sycara, Roth, Sadeh and Fox [13] on Distributed Constrained Heuristic Search has been ground-breaking in this area. More recently, researchers often use off-the-shelf solvers, such as ILOG, CHIP or ECL^{i}PS^e, to solve intra-agent problems and negotiation-based approaches to solve inter-agent (interface) problems [1, 12].

In our article we won't prefer any of these strategies. But we will build upon a DCSP definition that is slightly more general than the classical one of Yokoo. It allows for complex constraints and additionally extends the notion of distribution to the constraints also. That means that the administration of the knowledge needed to test consistency of a constraint is not necessarily implied by the distribution of the variables but can be in the competence of totally different agents. This definition of a DCSP also differs in terms of privacy from the one found in [11], in which constraint administration is distributed among agents but variables can be accessed freely by several agents.

Definition 1 (Distributed Constraint Satisfaction Problem) *A* distributed constraint satisfaction problem *(DCSP) is specified by a 5-tuple* $\Pi_{\text{dcsp}} = (X, D, C, A, \phi)$.

- $X = \{x_1, \ldots, x_n\}$ *is a set of* domain variables x_i *each ranging over its* domain D_i *from the set* $D = \{D_1, \ldots, D_n\}$.
- *A* labeling $\lambda : X \longrightarrow D_1 \cup \ldots \cup D_n, \lambda = (v_1, \ldots, v_n) \in D_1 \times \ldots \times D_n$ *assigns a value* $v_i \in D_i$ *to each domain variable* x_i. *A* partial labeling $\lambda' : (\{x_{\lambda'_1}, \ldots, x_{\lambda'_l}\} = X' \subseteq X) \longrightarrow (D_{\lambda'_1} \cup \ldots \cup D_{\lambda'_l} = D')$ *assigns values to the domain variables in* X'*, such that* $\forall x_i \in X' : \lambda'(x_i) = v_i \in D_i$.
- $C = \{C_1, \ldots, C_m\}$ *is a set of* constraints $C_i \subseteq D_{i_1} \times \ldots \times D_{i_k}$*, each of which is a relation of type* $2^{D_{i_1} \times \ldots \times D_{i_k}}$ *restricting the set of* feasible *(partial) labelings.*
- $A = \{(a_1, \kappa_1), \ldots, (a_p, \kappa_p)\}$ *is a set of* agents *each possessing a local consistency predicate* $\kappa_i : (X' \longrightarrow D') \times 2^C \longrightarrow \{\texttt{true}, \texttt{false}\}$ *over (partial) labelings and sets of constraints.*
- $\phi : X \cup C \longrightarrow A$ *assigns an agent to each domain variable and each constraint.*

Given this specification, the problem is to find a labeling $\lambda \in \Lambda(C) = \{(v_1, \ldots, v_n) \in D_1 \times \ldots \times D_n | \forall C_i \in C : (v_{i_1}, \ldots, v_{i_k}) \in C_i\}$ *from the* solution space Λ *defined by* C.

Remark 1 a) Since domains are always directly associated with their domain variables (variable-based approach), the distribution of domains depends on the distribution of variables and is not discussed further.
b) ϕ defines a partition on the union of domain variables and constraints that can be denoted by $\phi^{-1} = \{(a_1, \phi^{-1}(a_1)), \ldots, (a_p, \phi^{-1}(a_p))\}$ with $\phi^{-1}(a_i) = \{xc \in X \cup C \,|\, \phi(xc) = a_i\}$. Hence, $\bigcup_{a \in A} \phi^{-1}(a) = X \cup C \wedge \forall a_i, a_j \in A, i \neq j : \phi^{-1}(a_i) \cap \phi^{-1}(a_j) = \emptyset$. Additionally, $\phi_C^{-1}(a_i) = \{c \in C \,|\, \phi(c) = a_i\}$ and $\phi_X^{-1}(a_i) = \{x \in X \,|\, \phi(x) = a_i\}$.
c) The existence of a local consistency predicate κ_i only makes sense for an agent that administrates some constraints. In this case, the predicate can be specified more detailed by $\kappa_i : (X' \longrightarrow D') \times 2^{\phi_C^{-1}(a_i)} \longrightarrow \{\texttt{true}, \texttt{false}\}$, with X' being a subset of the variables connected to the constraints which are administrated by a_i.

The algorithms named above are mostly presented filling several pages of pseudo code. This seems to be due to the sequential tradition of many of these algorithms. Unfortunately, this kind of representation is not very intuitive and error-prone. Canonically, the authors often use sequential calculi, derivatives of the Hoare calculus, to prove important properties of their algorithms, like (partial) correctness, termination, and completeness. Yokoo and colleagues for example prove the completeness of their algorithms by more or less verbal argumentation chains [15]. Nguyen and Deville prove certain static assertions [8] of single threads and combine them to prove the overall system behavior. All these proofs run quite well since the underlying algorithms contain some sequential aspects, like the total order among agents in Yokoo's algorithms. The same holds for the algorithms in [1] and [12]. Things are getting harder when one is trying to analyze the behavior of fully detached concurrent threads. Zhang and Mackworth for example describe a fully distributed algorithm with an endless loop but do not detail on stabilization properties [17]. Sycara et al. leave out theoretical aspects on termination completely [13]. Several other agent-based approaches to solving practical constrained problems lack a theoretical analysis of their properties, too (refer e. g. to [10] for a detailed overview).

The problem with using sequential proof methods for concurrent algorithms is that in general the behavior of a system of threads does not only depend on the behavior of the single threads but also on matters like synchronization, interaction and so on. Therefore, in the following sections we will detail how a concurrent model of computation, algebraic Petri nets, and an according concurrent proof method can be used to present a sample DCSP algorithm intuitively without going into unnecessary details and to prove some of its properties.

3. MODELING CONCURRENT DCSP ALGORITHMS

3.1. INTRODUCING PETRI NETS

A good modeling paradigm for concurrent algorithms offers a representation of the concurrent flow of control, well-defined data types, a natural way to specify asynchronous behavior and mutual exclusion, a concise notion of fairness and the ability to use different layers of abstraction to compose model parts. Perhaps the most important property of such a modeling paradigm is the notion of local state. The global state of the system should always be derivable solely from the set of local states of its different components. Petri nets in general, and algebraic Petri nets in particular satisfy these demands. Additionally, they use a quite suggestive and nevertheless formally sound graphical notation.

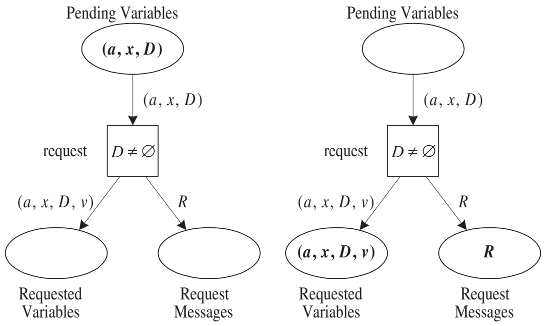


Figure 1: Before and After Firing of `request`

In Petri nets local states are represented by *places* (denoted by ellipses) and *tokens* on these places (drawn within ellipses). Atomic processes are represented by *transitions* (drawn as rectangles). Arcs connect places with transitions and model control and data flow. In algebraic Petri nets places, tokens, transitions and arcs are typed according to algebraic abstract data types. Take a look at the transition `request` in Fig. 1 to get an example for this. `request` is connected to the places `Pending Variables`, `Requested Variables` and `Request Messages`. The process represented by `request` can happen (*fire*), iff the inscription of the incoming arc (a, x, D) matches a token on `Pending Variables` and the *guard* $D \neq \emptyset$ evaluates to `true` under this matching. In this case, `request` takes away (a, x, D) from `Pending Variables`, produces a value v and places (a, x, D, v) on `Requested Variables` and several request messages R on `Request Messages`. All transitions in the net behave according to this rule totally independently and concurrently. In this context, a transition t in conflict to other transitions for the same token is treated *fair* by a system run (denoted by φ), iff t does not fire only finitely often, though being activated infinitely often.

Of course, we cannot give a further detailed introduction to algebraic Petri nets. Please refer to [9] for details. We will rather present a brief case study to allow an intuitive understanding of the strength of algebraic Petri nets for our purpose. This case study consists of a simplified, yet realistic algorithm for finding agreement on the labeling of several variables among several agents. The protocol does not describe a full DCSP solving algorithm since it lacks the ability of some kind of backtracking. It solely ensures a labeling procedure that can be used to concurrently label variables considering all constraints imposed on it. Backtracking can be implemented by a protocol allowing for cancellation of committed labelings but is out of scope of this paper. Another way to look at the presented protocol is an interpretation as a distributed forward checking algorithm without backtracking.

3.2. BASICS OF THE PROTOCOL

The agreement finding protocol involves p agents some of which care each for a subset of the n variables $x_1, \ldots, x_n \in X$ and some of which care each for a subset of the m constraints $C_1, \ldots, C_m \in C$ according to the distribution function ϕ. Agents caring for variables are called *variable agents*, agents caring for constraints are called *constraint agents*. A variable x has only one variable agent but may have several constraint agents that administrate constraints on x. Variables and their annotations are given by a tuple (a, x, D) in which a is the agent that cares for the variable, x is the variable itself and D is the domain of x. The set of constraint agents administrating constraints on x is denoted by $N(x) = \{(a, C)\}$ in which a is the agent and $C = \phi_C^{-1}(a)$ is the set of constraints administrated by a. $N(x)$ is also called the set of *neighbors* of x. An agent a may be variable agent for several different variables, hence there may be several tuples $(a, \cdot, \cdot)$. That means that a single agent may have several protocol instances running concurrently for different variables in its role as a variable agent. The same agent may also administrate several constraints, but because of consistency matters there is only one (a, C) tuple. This ensures, that an agent is only participating in a single protocol instance at a time in its role as a constraint agent which is important for the consistency property of our protocol. The result of the whole protocol shall be that if a variable x is labeled with a certain value, then all neighbors $N(x)$ have agreed on that labeling and have stored that information.

Figure 2 provides a coarse model of the agreement finding algorithm. The left half of Fig. 2 models the run of every variable in the course of the protocol, the right half models the run of constraint agents. In the middle the asynchronous message passing system is denoted. Message tuples contain the sender as the first component, the receiver as the second component and further components as message content. The message system is assumed to be reliable, though not necessarily preserving sequence.

The basic idea our protocol is to reduce conflicts and such search in the distributed problem space by making decisions on variable labelings well-founded. This involves a pre-processing phase in the protocol that is comparable to constraint propagation in traditional constraint programming. Before making a proposal for a specific value for a certain variable, a variable agent collects as much information on the other agents' constraints on this variable. This information can then be included in the internal problem solving procedure of the variable agent. This hopefully will remarkably reduce the probability to request agreement on an inconsistent value.

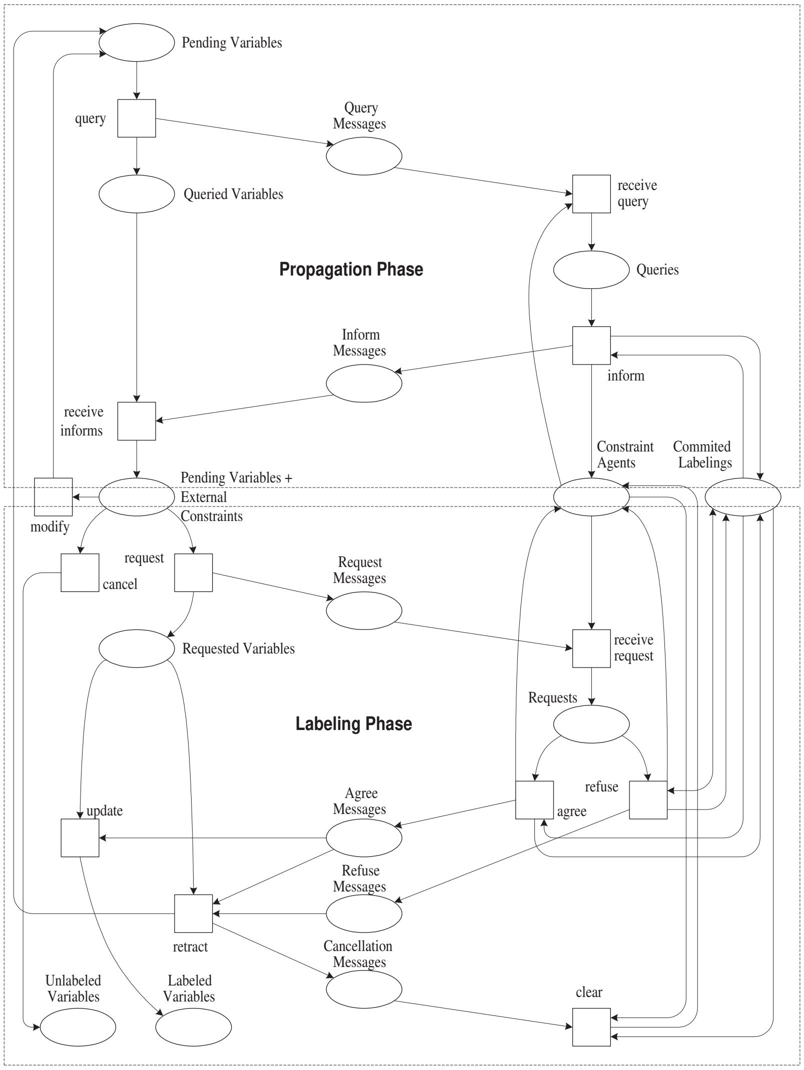


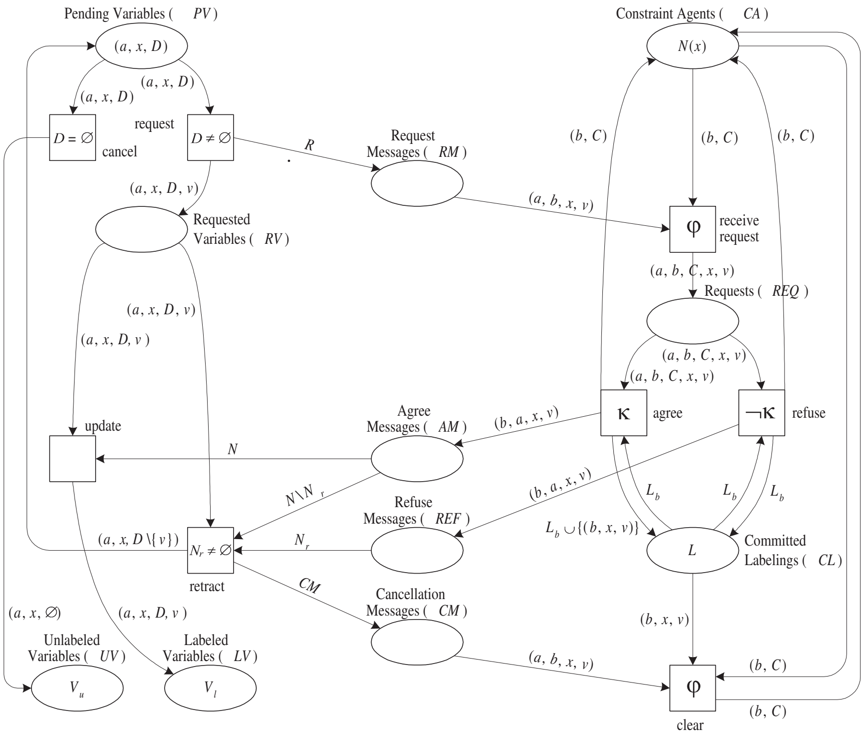
Figure 2: Coarse Petri Net Model Σ_{af}

3.3. A RUN OF THE PROTOCOL

We will follow exactly one instance of this protocol to exemplify its behavior following Fig. 2. Given a set of pending variables, the variable agent initiates the protocol by entering the *constraint propagation phase* and sending a query to all neighbors for gaining information on external constraints on its current variable. The variable agent can specify its requirements for this variable in the query messages. The state of the variable is changed from `Pending Variables` to `Queried Variables`. The task of all neighbors in this phase is to answer the query on constraints restricting the current variable according to their knowledge about already given commitments and constraints. Neighbors are assumed to be benevolent, so they will answer honestly. Nevertheless, they will follow their own strategies and safety requirements in posting information on that variable. The variable agent collects its variable together with its neighbors' answers on `Pending Variables + External Constraints`. The constraint information is collected asynchronously by waiting for the responding messages. After a certain timeout the constraint information is considered to be complete. Since constraint propagation is only an informative phase, non-responding neighbors are not critical. The variable agent can decide to start the next protocol phase based on the acquired information or to modify its requirements and restart the constraint propagation phase.

In the next phase of the protocol, called *labeling phase*, the variable agent a selects a certain value v out of D to label x (D has to be non-empty for this). For choosing a value for the current variable, the variable agent solves a full internal CSP considering all constraint information provided by the neighbors in the preceding phase. The labeling phase then ensures the agreement of all neighbors to the value selected by the variable agent. Since this protocol phase is critical to the correctness of the algorithm, Fig. 3 presents a more detailed model of this phase.

Following the selection of a certain value v, the constraint agent sends request messages to all neighboring constraint agents. Based on the received request, its constraints C and its maximum set of already committed labelings



Symbol	*Description*
$R = \{(a,b,x,v) \vert (b,\cdot) \in N(x)\}$	set of request messages from variable agent a to all neighbors $(b,\cdot) \in N(x)$ concerning variable x with value v
$N = \{(b,a,x,v) \vert (b,\cdot) \in N(x)\}$	set of agree messages from all neighbors $(b,\cdot) \in N(x)$ to variable agent a concerning variable x with value v
$N_r = \{(b,a,x,v) \vert (b,\cdot) \in N_r(x) \subset N(x)\}$	(non-empty) set of refuse messages from some neighbors $(b,\cdot) \in N_r(x) \subset N(x)$ to variable agent a concerning variable x with value v.
$N \backslash N_r = \{(b,a,x,v) \vert (b,\cdot) \in N(x) \backslash N_r(x)\}$	set of agree messages from some neighbors $(b,\cdot) \in N(x) \backslash N_r(x)$ to variable agent a concerning variable x with value v
$CM = \{(a,b,x,v) \vert (b,\cdot) \in N(x) \backslash N_r(x)\}$	set of cancellation messages from variable agent a to all neighbors $(b,\cdot) \in N(x) \backslash N_r(x)$ that have agreed on variable x with value v
$L_b = \{(b,\cdot,\cdot)\}$	all committed labelings agreed by neighbor b
$\kappa = \kappa_b(L_b \cup \{(b,x,v)\}, \phi_C^{-1}(b))$	evaluation of local consistency predicate of neighbor b under L_b together with the requested variable x with value v and constraints administrated by b

Figure 3: Detailed Petri Net Model Σ_{afl} of the Labeling Phase

$L_b = \{(b,\cdot,\cdot)\}$ each constraint agent b can decide to agree to the labeling or to refuse it by evaluating its consistency predicate $\kappa(L_b \cup \{(b,x,v)\}, \phi_C^{-1}(b))$. In case of agreement the constraint agent adds the labeling (b,x,v) to its set of committed labelings L_b and sends an agree message. Otherwise, the set of committed labelings remains unchanged, a refuse message is sent to the variable agent a and the constraint agent returns to its idle state.

The variable agent is blocked until all neighbors have answered. It can only update the requested variable (a,x,D,v) to the state `Labeled Variables` when all neighbors have sent agree messages. This full demanded set of agree messages is denoted by N. The according incoming arc of the `update` transition ensures the desired behavior. A single refusing neighbor is enough to hinder firing of the `update` transition. But even if there is already a refusal, the variable agent will have to wait for all responds of its neighbors to inform the agreed neighbors on the failure of the agreement. This is realized by the transition `retract`. This transition can fire iff there is a set $N_r \neq \emptyset$ of refusal messages from some neighbors (these neighbors are denoted by $N_r(x)$) and exactly the complementary set of agreement messages $N \backslash N_r$ from the other neighbors (denoted by $N(x) \setminus N_r(x)$). The transition `retract` deletes v from the domain of x and starts a new protocol instance. Additionally, all neighbors that have already agreed on the labeling v are informed by according cancellation messages to free them from their given commitments via the transition `clear`.

The variable agent can start another labeling phase of the protocol as long as there are values left in the domain D to chose the value v from. Since every time a value fails to find agreement among the neighbors it is removed from the domain, after a finite number of unsuccessful labeling phases the domain will eventually be empty. In this case, only the transition `cancel` is enabled by the guard $D = \emptyset$ and the variable is changed to the state of `Unlabeled Variables`.

4. PROPERTIES AND PROOFS

4.1. DESIRABLE PROPERTIES

Though desirable properties of concurrent algorithms for solving DCSPs are manifold, we will concentrate on two major classes, namely *safety* and *liveness* properties. Safety properties are strongly related to assertions in standard proof methods for sequential programs. Usually they are stated by a propositional expression p. The safety property p is said to hold in a Petri net Σ iff in each reachable state of Σ p is true. Safety properties can be checked relatively easily during the run of a concurrent algorithm and when they are violated the time point of the violation is determined at once. A typical safety property is consistency. An algorithm represented by Σ is said to preserve how-ever defined consistency iff in each reachable state of Σ the global state is consistent according to its definition. We will define our notion of consistency in the following based on partial labelings.

Definition 2 (ϕ-Consistency of a Partial Labeling) *Given a DCSP* $\Pi_{\text{dcsp}} = (X, D, C, A, \phi)$. *A partial labeling* $\lambda' : \{x_{\lambda'_1}, \ldots, x_{\lambda'_l}\} \longrightarrow D'$ *is called* ϕ-consistent*, iff*

$$\forall (a, \kappa) \in A : \kappa\,(\lambda', \phi_C^{-1}((a, \kappa))).$$

Remark 2 a) The function $\lambda' : \emptyset \longrightarrow \emptyset$ is also a valid and ϕ-consistent partial labeling.
b) The ϕ-consistency property is usually weaker than the consistency property achieved by testing consistency with the full information of all constraints. This holds even in case we are using the same algorithms for local and global consistency checking. In this context, "weaker" means that more partial labelings are accepted as consistent than actually may be. Nevertheless, ϕ-consistency of a full labeling implies global consistency, iff $\kappa_i(\lambda, \phi_C^{-1}(a_i)) \Rightarrow \lambda \in \Lambda(\phi_C^{-1}(a_i))$.
c) In the Petri net model Σ_{afl} a partial labeling is represented on the variable agents' side by the tokens on `Labeled Variables` and on the constraint agents' side by the tokens on `Committed Labelings`.

The problem with safety properties is that they are somehow "unprogressive". The easiest way to preserve a safety property is simply to do nothing. There is no way to guarantee that an algorithm "does something reasonable" only by asserting safety properties. This is why we need the second class of properties, namely liveness properties. Liveness properties ensure that the algorithm starting from a certain state in which p holds will finally produce a certain state in which q holds. Termination and completeness belong to this second category. Liveness is much harder to prove since there is not always a point in time at which we can check whether the property can eventually be satisfied or not. The most general case is even undecidable. We can nevertheless proof liveness properties in special cases. In algebraic Petri nets we are using temporal logics to do this. In the following subsections we will informally introduce techniques to prove safety and liveness properties of concurrent algorithms modeled by Petri nets and will apply this knowledge to prove consistency preservation and termination of our case study algorithm.

4.2. PLACE INVARIANTS

Safety properties are important to concurrent algorithms, since they allow statements such as "at each reachable state of Σ holds p", with p a propositional expression. Techniques to verify such properties include *place invariants*. Informally, a set of places defines a place invariant if under the firing of each connected transition the set of removed tokens equals the set of produced tokens. A formalization of this property in algebraic Petri nets is quite technical and involves the use of multisets and linear algebra. To simplify things, we will not define invariants directly on Σ_{afl}, but on a transformed net Σ_{afp} under the assumption that there is only one variable to label. This will be sufficient for our proof later. The transformed net model is shown by Fig. 4. On the left half of the figure we are simply projecting each token and arc inscription onto x thus gaining always the same token $\bullet$ that represents an instance of the variable token. On the right half we are projecting each token and arc inscription onto b thus gaining always the same token $\blacksquare$ that represents an instance of a constraint agent. As you can also see on the arcs leading from the left to the right half and vice versa the weight of a token $\bullet$ is always $|N|(\bullet)$ times higher than the weight of a token $\blacksquare$. Such, there are one variable agent instance and $|N|(\bullet)$ many constraint agent instances involved in a single protocol instance.

Figure 5 details one important invariant of Σ_{afp}. This invariant can be formally validated by solving a linear homogeneous equation but we will only motivate it intuitively. Invariant 1 involves the five places `Requested Variables` (RV), `Request Messages` (RM), `Requests` (REQ), `Agree Messages` (AM) and `Refuse Messages` (REF). Looking at Fig. 5 it seems obvious that

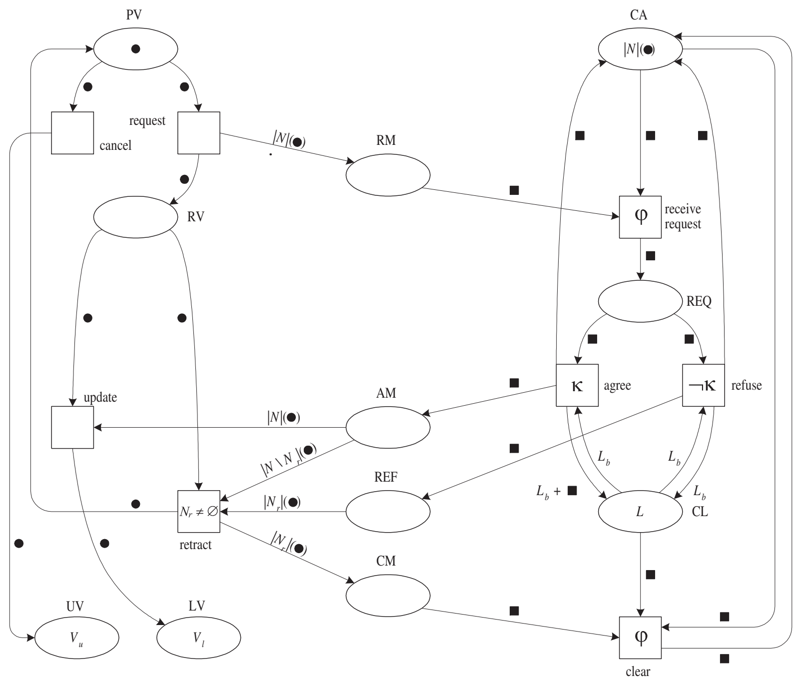


Figure 4: Petri Net Model Σ_{afp} with Projections for Finding Invariants

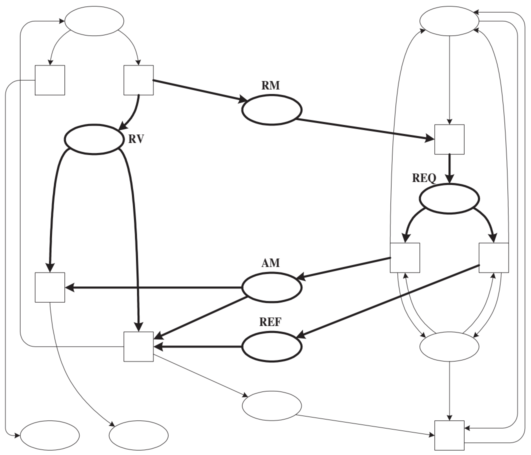


Figure 5: Invariant 1

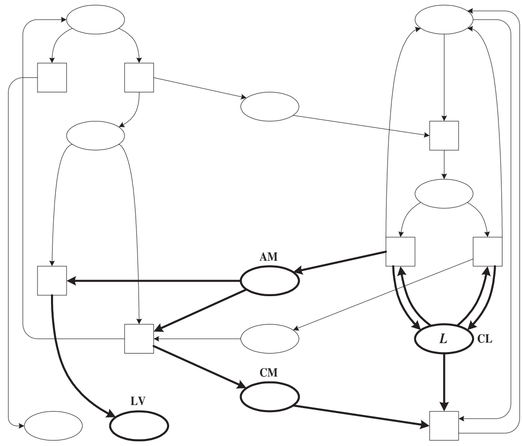


Figure 6: Invariant 2

each transition connected to one of the invariant places adds the same amount of tokens to the invariant places as it consumes. Graphically this can be verified by following the flow of tokens along equally directed arcs between RM, REQ, AM and REF. The only exception is place RV since the flow of tokens reverses in this place with arcs directed into the opposite direction of all other arcs. Additionally, RV has a higher weight as explained above. The consequence is that the weighted instance sum of all tokens on RM, REQ, AM and REF equals the instance sum of tokens on RV times $|N|(\bullet)$. This is formalized by introducing a special addition operation defined on multisets, counting instances in this case. Such, invariant 1 as illustrated by Fig. 5 is formally given by

Inv. 1: $RM + REQ + AM + REF = |N|(\bullet) \cdot RV.$

Further invariants can be derived likewise and are illustrated by Fig. 6 and Fig. 7.

Inv. 2: $|N|(\bullet) \cdot LV + AM + CM - CL = -L$
Inv. 3: $PV + RV + LV + UV = 1$
Inv. 4: $CA + REQ = |N|(\bullet)$

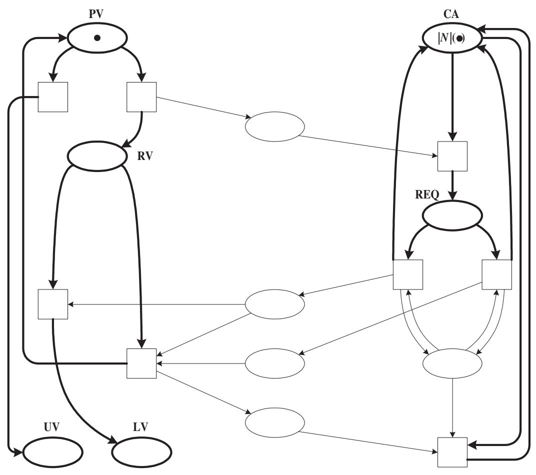


Figure 7: Invariants 3 and 4

4.3. CAUSES AND PROOF GRAPHS

We informally introduce a concurrent deduction operator, called *causes* and denoted by $\hookrightarrow$, to prove liveness properties of concurrent algorithms. Again, please refer to [9] for details. $\Sigma \models p \hookrightarrow q$ ("in Σ holds p *causes* q") means that in Σ each reachable state in which p holds is always followed by a reachable state in which q holds. The *causes* operator can be described further by inscribing the transitions that are involved in the process on top of the $\hookrightarrow$ symbol. We are using so-called *pick-up patterns* to derive liveness properties directly from the static structure of an algebraic Petri net. Though there are some more basic ones, the following pick-up patterns are sufficient for our purpose. Every pattern preserves a certain context α that remains unchanged by transitions in the pattern. $A = U$ means that exactly the set U of tokens is on A. $A.U$ means that at least the set U of tokens is on A.

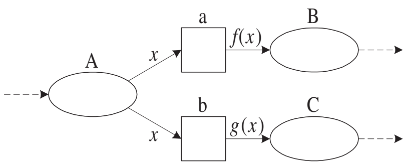


Figure 8: Pattern 1

The pattern shown by Fig. 8 covers alternative forward branching. Transitions a and b are in conflict for the tokens on A. Given $A = U$, these transitions can nondeterministically and concurrently consume a token x from A and produce $f(x)$ or $g(x)$, respectively. The final result of firing a and b as often as possible is that A is empty and the transformed tokens have been placed on B or C, respectively. The set of tokens V consumed by a unified with the set of tokens W consumed by b again equals U. Hence, we can formally state

Pattern 1: $(\alpha \wedge A = U) \overset{a(A)\vee b(A)}{\hookrightarrow} (\alpha \wedge A = 0 \wedge B.f(V) \wedge C.g(W) \wedge U = V \cup W)$

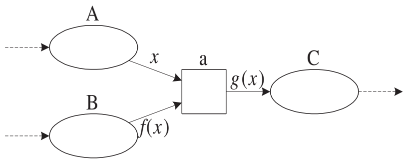


Figure 9: Pattern 2

The pattern shown by Fig. 9 covers synchronization. Given that $A = U$ and $B.f(U)$ (invariant notation $f(U) \leq B$), transition a can consume tokens x from A and $f(x)$ from B as long there are tokens on A. The final result of firing a as often as possible is that A is empty and $g(U)$ has been placed on C. Hence, we can formally state

Pattern 2: $(\alpha \wedge A = U \wedge f(U) \leq B) \overset{a(A)}{\hookrightarrow} (\alpha \wedge A = 0 \wedge C.g(U))$

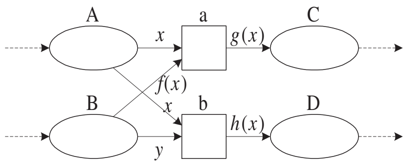


Figure 10: Pattern 3

The pattern shown by Fig. 10 covers alternative synchronization. Transitions a and b are in conflict for the tokens on A and B. Let's assume that we can derive somehow that, given $A = U$ and $f(U) \leq B$, transition b is prevented from firing for some reason. Then we can adapt pattern 2 and formally state

Pattern 3: Let $A = U$ prevent $b(f(U))$. Then $(\alpha \wedge A = U \wedge f(U) \leq B) \overset{a(A)}{\hookrightarrow} (\alpha \wedge A = 0 \wedge C.g(U))$

Since, if $\Sigma \models p \hookrightarrow q$ and $\Sigma \models q \hookrightarrow r$ then $\Sigma \models p \hookrightarrow r$, we can connect simple *causes*-deductions and propositional deductions to form more complex proof sequences. In fact, we can even form graphs of deduction steps. These graphs are called *proof graphs*. We will exemplify such a proof graph in the next subsection.

4.4. PROVING TERMINATION AND CONSISTENCY

After undergoing this brief description of tools for verifying concurrent algorithms we apply this new knowledge to our case study. We prove the following theorem stating ϕ-consistency and termination of the protocol modeled by Σ_{afl}.

Theorem 1 *Given a DCSP* $\Pi_{\text{dcsp}} = (X, D, C, A, \phi)$. *Then, each possible run of* Σ_{afl} *is finite and will finally produce a* ϕ*-consistent partial labeling* $\lambda' : X' \subseteq X \longrightarrow D'$ *potentially trying each domain value of each variable* $x \in X$.

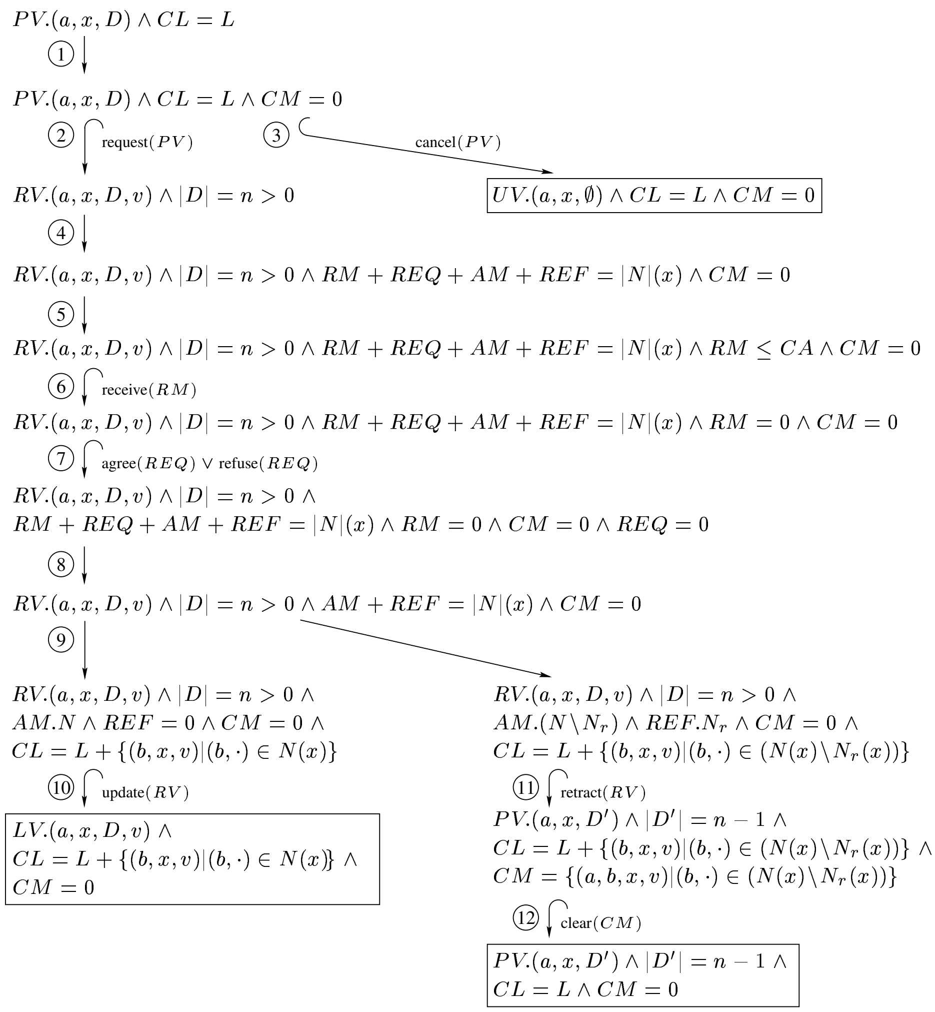


Figure 11: Proof Graph for Theorem 1

Proof

Since the empty labeling $\emptyset \longrightarrow \emptyset$ is also a ϕ-consistent partial labeling, we can make the assumption that at the present situation we already have a ϕ-consistent partial labeling, represented on variable agents' side by the set of labeled variables V_l on LV and on constraints agents' side by the set of committed labelings L on CL. It is an important observation that all arc expressions in Σ_{afl} are directly bound to a specific variable x, except of the incoming arcs (b, C) and L_b of the transitions `receive`, `agree`, `refuse` and `clear`. Therefore, different instances of the protocol concerning different variables only conflict in these transitions. Because there is only a single (b, C) token on CA for each constraint agent these conflicting accesses to CA and CL are synchronized producing an arbitrary but strictly ordered sequence. I. e. only a single protocol instance can fire a `receive` and `agree`/`refuse` sequence or a `clear` transition at a time. All protocol instances are totally synchronized and it is therefore enough to take a closer look at the properties of a single protocol run for adding a single variable (a, x, D) to the existing partial labeling. Figure 3 reflects exactly this situation with only one variable on PV and the according constraint agents $N(x)$ on CA.

Adding a variable to the existing partial labeling preserves ϕ-consistency, iff every constraint agent can still evaluate its consistency predicate $\kappa(L_b \cup \{(v, x, v)\}, \phi_C^{-1}(b))$ to `true`. Thus, every constraint agent (b, C) has to check this property by testing the new labeling proposal (a, x, D, v) against its set of constraints and recent committed labelings. If only one constraint agent refuses the new labeling proposal is inconsistent and has to be discarded. The following proposition formalizes this demand to the protocol Σ_{afl}

$$
\begin{aligned}
\Sigma_{\text{afl}} \quad \models \quad & PV.(a,x,D) \wedge CL = L \hookrightarrow && (1)\\
& (\ \ LV.(a,x,D,v) \wedge && (2)\\
& \quad CL = L + \{(b,x,v)|(b,\cdot) \in N(x)\} \wedge \\
& \quad CM = 0 \\
&) \ \vee \\
& (UV.(a,x,\emptyset) \wedge CL = L \wedge CM = 0). && (3)
\end{aligned}
$$

In words this proposition means that given a pending variable (a, x, D) under an existing partial labeling represent by L (1), every possible run of the protocol causes either that (a, x, D, v) has been successfully labeled and every neighbor has committed to it (2) or the domain of x has been traversed without success, $(a, x, \emptyset)$ is in state `Unlabeled Variables` and not a single neighbor is still committed to the related requested labeling (3).

Figure 11 shows the proof graph that proves the aforementioned proposition. To justify the deduction steps in this graph we use invariants 1 to 4 from Subsect. 4.2 and the *causes* pick-up patterns presented in Subsect. 4.3.

① Invariant 2.

② + ③ Pattern 1 according to $D \neq \emptyset$ or $D = \emptyset$.

④ Invariant 1 with $RV = 1$ and invariant 2.

⑤ Combining invariants 1 and 4 with $RV = 1$ provides $RM + AM + REF - CA = 0$ and such $RM \leq CA$.

⑥ No conflicting transition, but only fair `clear` together with pattern 2.

⑦ No conflicting transition, L_b may be $\emptyset$ and pattern 1.

⑧ Propositional logic.

⑨ Invariant 3 with $RV = 1$ yields $LV = 0$ and such invariant 2 with $CM = 0$ yields $CL = AM + L$. Fall differentiation leads to two possible cases.

⑩ Conflicting transition `retract` is prevented since $REF = 0$. Pattern 3.

⑪ Conflicting transition `update` is prevented since $\neg AM.N$. Pattern 3.

⑫ `receive`, `agree` and `refuse` are the only transitions in conflict to `clear`. `receive` is fair and will hence not deny `clear` forever. Due to invariant 4 all constraint agents will always return to CA after answering a request (`agree` and `refuse` do not block forever if there is a request on REQ). Also from invariant 4 we can derive that $CA.(b, C) \longrightarrow \neg REQ.(\cdot, b, C, \cdot, \cdot)$. Hence, if $CA.(b, C)$ `agree` and `refuse` are prevented and `clear` will finally fire.

The proof graph starts at $PV.(a, x, D) \wedge CL = L$ (1) and has three possible outcomes two of which directly correspond to (2) and (3) of our proposition. The third outcome $PV.(a, x, D') \wedge |D'| = n - 1 \wedge CL = L \wedge CM = 0$ corresponds to the starting point of the proof graph with a reduced domain. Nevertheless, this situation cannot be produced infinitively, since D is fully traversed, $|D|$ reduces by a constant in every protocol cycle and will in the long run reach $|D| = 0$. Only `cancel`(PV) ③ is enabled in this situation and the protocol terminates after trying each value v from D. □

Corollary 1 *Given a DCSP* $\Pi_{\text{dcsp}} = (X, D, C, A, \phi)$. *If a run of* Σ_{afl} *has produced a full labeling* $\lambda : X \longrightarrow D_1 \cup \ldots D_n$, λ *is globally consistent under the assumption that* $\kappa_i(\lambda, \phi_C^{-1}(a_i)) \Rightarrow \lambda \in \Lambda(\phi_C^{-1}(a_i))$.

5. CONCLUSIONS

The purpose of this article has been to show how successfully one can use algebraic Petri nets as formal but suggestive means to model and verify concurrent algorithms for DCSPs. Following a simple case study on agreement finding we have informally introduced the necessary tools to prove safety (via place invariants) and — more important and difficult — liveness properties (via the temporal *causes* operator). We have applied these tools to prove termination and consistency of our agreement algorithm.

The algorithm presented in this paper is itself a novel contribution since it allows for complex and additionally fully private constraints. This is especially important in the context of socially embedded constraint solving for which the DCSP model seems to be most appropriate. Good examples for social embeddedness include supply networks that consist of entities that are strongly connected by common variables and constraints but do not allow for full information exchange. E. g. in the case of one enterprise delivering goods to another enterprise, the delivery date directly influences the constraint problem within the recipient. Even though this is true, the providing enterprise will not offer full access to its constraint and variable information. It will only provide the guaranteed delivery date. In our algorithm, this can easily be incorporated by modelling the providing enterprise as a constraint agent deciding on its own what constraint information to offer in the propagation phase and on which values to agree on in the labeling phase.

Nevertheless, our algorithm does not guarantee completeness. Hence, it is part of our future work to augment the algorithm with concurrent backtracking facilities, model them appropriately and prove their validness. Research in dynamic backtracking [14] and the work of Yokoo et al. on variable ordering is promising to further our work into this direction. We have some initial ideas on this issue that base on collecting conflict constraints when receiving refusals from neighbors. These conflict constraints can then be used to make a better informed choice in a newly initialized search stage. Though, we have some difficulties in assigning this newly created conflict constraints to constraint agents for administration. Our current approach even needs the possibility to spawn new constraint agents and such entails an explicit increase in the density of the constraint graph. Further results are to come.

ACKNOWLEDGEMENTS

Many thanks go to Wolfgang Reisig, Hans Schlenker, Sebastian Müller, Frank Rehberger and Armin Wolf for fruitful discussions on the concurrent algorithms' side of this paper as well as on the constraint programming matters, respectively.

REFERENCES

[1] P. Berlandier and B. Neveu. Problem partition and solvers coordination in distributed constraint satisfaction. In *Proceedings of the Workshop on Parallel Processing in Artificial Intelligence (PPAI-95)*, Montréal, Canada, 1995.

[2] H.-D. Burkhard. Fairness and control in multi-agent systems. *Theoretical Computer Science*, 189:109–127, 1997.

[3] M. Fisher and M. Wooldridge. On the formal specification and verification of multi-agent systems. *International Journal of Cooperative Information Systems*, 6, special issue on Formal Methods in Cooperative Information Systems. Multi-Agent Systems, 1997.

[4] T. Holvoet. Agents and Petri nets. *Petri Net Newsletter*, 49, 1995.

[5] C. M. Jonker and J. Treuer. Compositional verification of multi-agent systems: a formal analysis of pro-activeness and reactiveness. In W. P. de Roever, H. Langmaack, and A. Pnueli, editors, *Proceedings of the International Workshop on Compositionality (COMPOS-97)*. Springer, 1997.

[6] Q. Y. Luo, P. G. Hendry, and J. T. Buchanan. Comparison of different approaches for solving distributed constraint satisfaction problems. Research Report RR-93-74, Department of Computer Science, University of Strathclyde, Glasgow G1 1XH, UK, 1993.

[7] K.-P. Neuendorf and M. Hannebauer. Formal modeling of multi-agent interaction in distributed scheduling. In *Proceedings of the 16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation (IMACS-2000)*, Lausanne, Switzerland, 2000.

[8] T. Nguyen and Y. Deville. A distributed arc-consistency algorithm. *Science of Computer Programming*, 30(1-2):227–250, 1998.

[9] W. Reisig. *Elements of Distributed Algorithms — Modeling and Analysis with Petri Nets*. Springer, 1998.

[10] W. Shen and D. H. Norrie. A hybrid agent-oriented infrastructure for modeling manufacturing enterprises. In *Proceedings of the Knowledge Acquisition Workshop (KAW-98)*, Banff, Canada, 1998.

[11] M. C. Silaghi, D. Sam-Haroud, and B. Faltings. Distributed asynchronous search with private constraints. In *Proceedings of the Fourth International Conference on Autonomous Agents (AGENTS-2000)*, pages 177, 178, Barcelona, Spain, 2000.

[12] G. Solotorevsky, E. Gudes, and A. Meisels. Modeling and solving distributed constraint satisfaction problems (DCSPs). In *Proceedings of the Conference on Constraint-Processing (CP-96)*, 1996.

[13] K. P. Sycara, S. F. Roth, N. Sadeh, and M. S. Fox. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1446–1461, 1991.

[14] G. Verfaillie and T. Schiex. Dynamic backtracking for dynamic constraint satisfaction problems. In *Proceedings of the ECAI-94 Workshop on Constraint Satisfaction Issues Raised by Practical Applications*, pages 1–8, Amsterdam, The Netherlands, 1994.

[15] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and DATA Engineering*, 10(5), 1998.

[16] M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98)*, pages 372–379, Paris, France, 1998.

[17] Y. Zhang and A. K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Proceedings of the IEEE-Symposium on Parallel and Distributed Processing*, pages 394–397, 1991.