Token-controlled place refinement in hierarchical Petri nets with application to active document workflow

David G. Stork[1] and Rob van Glabbeek[2]

[1] Ricoh California Research Center
2882 Sand Hill Road Suite 115
Menlo Park, CA 94025-7022
stork@rii.ricoh.com

[2] Department of Computer Science
Stanford University
Stanford, CA 94305
rvg@cs.stanford.edu

Abstract. *We propose extensions to predicate/transition nets to allow tokens to carry both data and control information, where such control can refine special "refinable place nodes" in the net. These formal extensions find use in active document workflow, in which documents themselves specify portions of the overall processing within a workflow net. Our approach enables the workflow designer to specify* <u>*which*</u> *places of the target predicate/transition net may be refined and it enables the document author to specify* <u>*how*</u> *these places will be refined (via attachment of a token-generated "refinement net"). This apportionment of the overall task allows the workflow designer to set general constraints within which the document author can control the processing; it prevents conflicts between them in foreseeable practical cases. Refinable places are augmented with a permission structure specifying which document authors can refine that place and which document tokens can execute a node's refinement net. Our refined nets have a hierarchical structure which can be represented by bipartite trees.*

1 Introduction

Document process workflow — such as the sequence of operations on documents in a loan application, employment application, insurance claim, purchase requisition, online credit verification at purchase, processing of patient medical records, distribution and sign-off on memos in an office, or editorial steps in the production of a magazine — is an increasingly important application of concurrency theory. While a number of ad-hoc high-level languages and commercial systems such as COSA, Visual Workflow, Forte Conductor, Verve Workflow, iFlow, InConcert, and SAP R/3 Workflow have been developed to serve such applications [16], there is nevertheless a need for a formal language that would allow workflow properties to be derived, for example halting, reachability, invariances, deadlock, and livelock. Further, if various high-level workflow languages can be expressed in a common formal language, their functional differences and similarities can then be exposed and analyzed in that formal language. Finally, a unifying formal language would enable business partners to merge their workflows (for instance through chaining or synchronization) even though they use different high-level workflow languages [2].

Wil van der Aalst and his colleagues [1] have argued persuasively that Petri nets (of which predicate/transition nets form a subclass) provide such a formal foundation for document workflow for these and other reasons, specifically:

Formal semantics: Petri net formalism provides precise definitions and clear semantics of both basic systems and those enhanced with attributes such as color, time, and hierarchy.

Graphical nature: Petri nets have natural graphical representations and are thus intuitive, easy to learn, and admit natural human-machine interfaces supporting drag and drop, click and link, and other operations on icons representing processes and documents.

Expressiveness: Basic Petri nets can support the functional primitives needed to model existing document workflow systems.

Management systems can be modeled: Local states in Petri nets are represented explicitly and this allows for the modeling of implicit choices and milestones.

Reasoning about properties: Petri-net-based process algebra rests on firm mathematical foundations, and this facilitates reasoning about network properties.

Analysis: A wealth of formal Petri net analysis techniques have been developed for proving properties (safety, invariance, deadlock, ...) and for calculating performance measures (response times, waiting times, occupation rates, ...).

Vendor independence: Petri nets are a tool- and vendor-independent framework and as such they will not vanish amidst inevitable turmoil in the marketplace.

Rather than model or analyze existing workflows, in the below work we build upon and extend Petri nets to provide a foundation for more powerful, enhanced future workflow systems. While basic Petri nets are indeed attractive as a foundation for current workflow systems, Petri net theory as it stands is incapable of expressing properties we seek in such expanded workflows. In particular we explore the properties of *active document workflow* and introduce extensions to Petri net theory that enable them. In Section 2 we review the use of Petri nets in traditional document workflow, sketch what functionality we want in active document workflow, and mention analogies in other areas of computer science. Then, in Section 3 we review several Petri net and related formalisms and show that they are insufficient to express such active document workflow. In Section 4 we give the formal definition of token-controlled place refinement and we conclude in Section 5 with some future directions.

2 Document workflow

2.1 Traditional workflow

In traditional document workflow, a workflow designer specifies the processing steps to be applied to a (passive) document. If the workflow is implemented as a Petri net, then *tasks* or *transitions* (represented by squares) specify the operations to be performed, and *local states* or *places* (represented as circles) specify the status. The causal flow through the Petri net is indicated by arrows or *arcs* linking transitions to states and states to transitions (but never states to states or transitions to transitions). Documents are often represented as structureless tokens that flow through the network. States or conditions within the workflow — such as "copier tray empty," "return receipt sent," and so forth — are represented by place nodes, and the status is indicated by the presence or absence of structureless tokens.

Figure 1 shows a simple document workflow that might be used by a newspaper publisher. Ignore for the moment the bold circle, and consider how such a network implements a traditional workflow. Reporters in the field submit drafts of articles to the newspaper editorial office either by fax or by email; if an article is faxed, then the transmitted document is electronically scanned at the office to create an electronic version. The submission is logged, and then the electronic document is passed to an editor who makes corrections and passes the edited document to the typesetter.

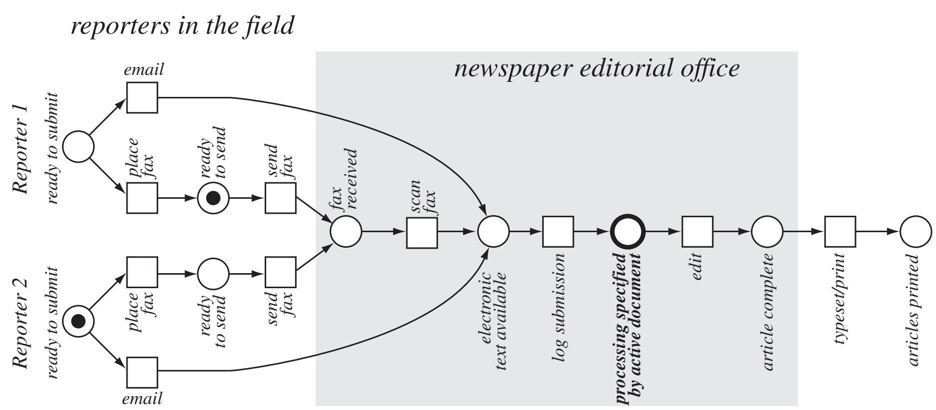

Fig. 1. A workflow implemented as a Petri net specifies the tasks (squares) and states (circles) linked by arcs. In this schematic example, two newspaper reporters can submit articles to the editorial office either by fax or by email, ultimately leading to typeset articles in the newspaper. In the current state, the draft of an article is being faxed by Reporter 1 (but has not yet been scanned); Reporter 2 is ready to submit an article. This state is shown by the dots (tokens), constituting a "marking." (Because this network has two input places, it differs slightly from workflow nets as defined in [1].) Active document workflow exploits processing in the bold circle (see Sect. 2.2).

2.2 Active document workflow

Consider now how the full network in Fig. 1 (i.e., with the bold circle) can represent *active* document workflow in the case of a single article submitted by Reporter 1. As before, Reporter 1 submits a draft article, but in this case the reporter has added special control instructions to the document. After the submission has been logged in the workflow network, this special control information is read and executed at the state node indicated by the bold circle. The control information in the document must be in a machine-readable form, and could specify tasks such as "translate paragraph 2 from French to English," "convert the entries in Table 5 from Japanese yen into U.S. dollars using last October's exchange rate," "send a copy of this document to every person in the international news division," or combinations of such operations. This processing is invoked much like function or library calls to a set of resources made available by the editorial office and known by the reporter. These are operations that for any number of reasons can be done more easily and rapidly in the office than in the field. For instance, the relevant extra information to be incorporated or processing software may reside behind a firewall of the editorial office; resources such as a translator or image processing systems may be available only within the office; and so forth.

In other applications, active documents might contain the following control information:

– "send a return receipt to the sender"
– "print this letter on corporate letterhead and mail it to the following address"
– "crop photograph 3 to be square"
– "encrypt this document text using *PGP*"
– "determine the name of the recipient's corporate president and replace every occurrence of '`president`' in this document with his or her name"
– "change the text from one column to two column, except for tables and the appendix"
– "route this document to the CFO and request an acknowledgment; if an acknowledgment is received, pass the document to the next stage in the target workflow, otherwise encrypt the document and then pass the document to the next stage"
– "create a PDF duplicate of this document and send it to the internal document archive"

We stress that this enhancement is not merely the token-controlled selection of pre-existing operations anticipated by the workflow designer. Instead, it is the putting together of operations in an arbitrarily large number of unanticipated ways. Up to now, when workflow designers or users have specified such control tasks they have done so in an ad-hoc or informal way. For instance, in our example the reporter might have telephoned or sent a separate message to a secretary at the editorial office specifying the alterations to be performed on the submitted draft article.

The increase in electronic workflow — office intranets, networked office appliances, business-to-business nets, the world wide web — presents an opportunity for automating these processes and scaling them to large workflow systems. Our research goal is thus to provide a formal foundation for workflow that includes methods and protocols for automating the processing of active documents. Thus in the newspaper example, active document workflow would automate the processing and thereby ensure the editor and others receive a document in a form the author wishes, rapidly and with a minimum of direct human intervention. Standards for such active document workflow — specification in a formal language, conventions for calls to libraries of resources, acknowledgement protocols, and so on — would facilitate interorganizational active document workflows as well.

In our proposed extension to traditional workflow, the workflow designer specifies *which* places in the network can support token-specified processing, as well as the resources available for such processing; the document author specifies *how* those resources will be used for document processing. This apportionment of the overall document processing and routing allows flexibility to different applications. In fixed or rigid applications, the workflow designer retains absolute control and does not permit any document-controlled processes in the workflow network. That is, there would be no nodes such as the bold circle in Fig. 1, just as in traditional (passive) document workflow. In other applications, the workflow designer may permit some limited flexibility or freedom to the document author, perhaps only after important workflow processes have been completed, such as logging or archiving a copy of the original document. Moreover, the workflow designer provides a set of resources (libraries) and this set implicitly limits the operations that can be invoked by the document author. Finally, in networks serving multiple or highly variable operations, the workflow network can have multiple places where the author can specify processing, and provide large libraries of basic operations that can be invoked by the active document.

An example of rigid workflow is the processing of issued traffic tickets and fines at a police station or motor vehicle administration. Here the offender has no freedom in specifying how the ticket and fine are to be processed. An example in which modest freedom is granted to the document author is the newspaper example above. Consider now applications in which the document author has great freedom. Suppose a mobile professional has an office computer running an active document processing system. In this case the author might compose a business letter on a small portable device, send it electronically to his home computer that supports active document workflow. The document itself might specify that the letter is to be printed on corporate letterhead on special letter stock, be bound with the latest version of the corporate financial sheets resident on the home machine, and mailed to a particular set of recipients. Active document workflow of this type is particularly useful in low-bandwidth mobile or other non-synchronous applications. We can imagine, too, an outsource document processing service, where customers submit active documents specifying typesetting, layout, translation, graphics processing, postal or electronic distribution, archiving, and notary services.

Our approach deliberately blurs and reduces the distinction between data and control, much as lambda expressions describe both functions and arguments in the λ-calculus and in the high-level programming languages based on the λ-calculus such as *Lisp* and *Scheme* [27, 8]. In an indirect and informal way, file headers or extensions such as `.jpg` or `.gif` indicate that images are to be processed or rendered in a particular way, again in analogy to what we propose here. A stronger and richer analogy to our active document processing is the TeX typesetting and document preparation system, where a single source file contains a mixture of text and control functions that specify how that document should be formatted and the text rendered [15]. None of these systems, however, encompass the tasks of distribution and workflow more generally.

2.3 The roles of humans

Humans are essential components of any workflow system, of course, and we must clarify their responsibilities during design and use of our enhanced systems. An expert workflow designer constructs the workflow within the newspaper office, specifying the processing steps, flow relation, the refinable places and the library of functions that can be called by active documents. Similarly, this expert specifies the permission structures which will control which document authors can refine a given place (see Sect. 2.4). Document authors — here, the reporters — are non-experts and need only understand the functions available in the libraries. In this newspaper example, yet other humans serve as resources specified in the library. Thus the operation "translate from French to English" would send a document to a bilingual expert who needs no expertise in document workflow or other conventions.

2.4 Permission structures

Consider next a workflow network within a corporation having both a legal department and a finance department. Documents such as letters, faxes or emails received by the corporation are classified and then routed through the corporation's document workflow to the appropriate department. Employees in such a corporation may wish to tailor ("reprogram") the workflow by means of documents themselves, even if they are outside the office. Surely, only members of the legal department should have ability and permission to alter the processing of documents that are routed to the legal department, and analogously for the finance department.

Our approach supports such alterations by allowing both temporary changes (affecting only the current document) and semi-permanent changes (affecting multiple, future documents) to the workflow. To this end we allow active documents not only to change (or refine) a workflow net, but also to undo or reverse such changes. The altered network is therefore hierarchical, where a network implementing the new functions lies at a deeper level than the original network. Our formalism supports *permission structures* that specify who can make a change, and which documents are affected by such a change, as discussed below.

3 Related formalisms

Given that our formalism for active document workflow will build upon Petri nets for the reason mentioned, there are nevertheless a number of related general formal approaches and specific proposals that must be considered. As we shall see, though, none of them have all the properties needed to enable the active document processing we envision.

Graph rewriting algebras: Graph rewriting algebras specify how to replace subgraphs with other subgraphs [6]. There is a structure governing these substitutions, such as formal composition, reversion (contraction), and so on. These algebras are inadequate for active document workflow because they contain no notion of the document itself controlling the rewriting.

Action refinement: In concurrency theory and process algebra, action refinement refers to the substitution of complicated actions for simpler ones [4]. By itself, this technique is insufficient for active document workflow as there is no provision for these substitutions to take place during the execution of a process.

Petri nets with refinement: Refinement in a Petri net is the process by which a node in a network is replaced by other networks [24, 11, 7]. In standard Petri nets with refinement of places or transitions, however, the refinement is specified by the creator of the network rather than by information in tokens themselves. We shall employ token-controlled refinement as the mechanism for implementing the processing specified by a document author. Furthermore, our refinement need not *replace* an existing section of a net, but instead *attach* a refinement net in a way that will be described below.

Traditional workflow nets: Traditional workflow nets [1, 2] represent documents by tokens, but in essence ignore the contents of those documents. For this reason they are not sufficiently expressive for token-controlled refinement. Furthermore, most effort on formal document workflow employs Petri nets such as basic or colored nets rather than the predicate/transition nets we shall require [3].

Petri nets with structured tokens: In higher-level nets [23], such as predicate/transition nets [10, 9] and colored Peri nets [13, 14], tokens carry structured information that can be exploited at transitions. Higher-level net-based document workflow supports processing such as "if the document is a bill for over $10,000 from a familiar supplier, forward the bill to the Chief Financial Officer." Existing higher-level nets however do not allow changes to the workflow network, and hence no token-controlled changes to the workflow network in particular. To our knowledge, higher-level nets have never been used for refinement.

Reconfiguration: Dynamic reconfiguration, where processes are changed dynamically based on intermediate computed results, is used in a range of applications such as FPGA-based reconfigurable computing. The FPGA approach does not rest on formal foundations such as Petri nets, and the message packets in this approach bear only weak correspondence to the tokens in Petri nets for workflow [20]. Badouel and Oliver describe a Petri-net self-modifying or "reconfigurable net," which modifies its own structure by rewriting some of its components, but the reconfiguration information is not passed by the tokens [5]. Similar ideas appear already in the work of Valk [25].

Hierarchical networks: In [26, 18] hierarchical nets are studied in which tokens (objects) are nets as well, whose transitions may synchronize with the ones from the system net in which they travel. Although this approach is close to ours in a number of ways, it separates the behavior of the system net and the object net, and as of yet does not allow the passing of document contents from one to the other.

In summary, no previous formalisms support the full range of active document workflow and structures we envision. We now turn to the specification of such a formalism.

4 Token-controlled refinement in hierarchical nets

We begin with an informal description of basic token-controlled refinement, and subsequently develop, in various successive stages, formal definitions of the kind of nets that support such refinement. Because active documents will carry both data ("text") and control information ("control"), documents must be represented by *structured* tokens in a Petri net. If we consider the document as a file, the data and control may be intermixed so long as each is tagged and separable.

We call the traditional workflow net the "target net," to distinguish it from "refinement nets" specified by the structured token. In active document workflow, the workflow designer specifies the target net and any number of special "refinable places," which will serve as the loci of document-controlled operations. Figure 2 shows the internal structure of a refinable place, such as the bold circle in Fig. 1.

The top portion of Fig. 3 shows a target Petri net which includes a single refinable place. When a structured token enters the refinable place, refinement is then "enabled." Information in the token specifying the refinement net is read and the refinement net may then be attached between the input and output places i_s and o_s, as shown. In our approach, local *places* rather than transitions are refined because it simplifies formal definition and emphasizes that refinement amounts to the addition of extra tasks to a workflow process rather than a detailed explanation of how to execute an already specified process.

It is natural to consider the refinement net to lie at a different level than the target net, as shown in Fig. 3. Below we shall describe how another token carrying the instruction that a refinement should be undone may arrive, and the resulting *contraction* eliminates the refinement net.

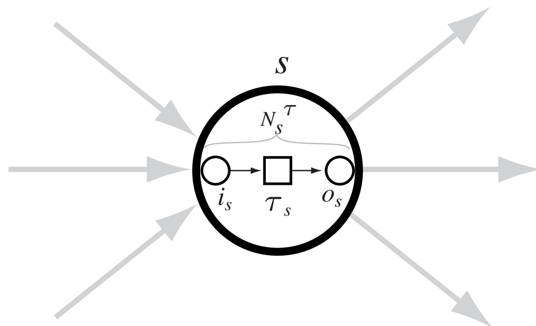

Fig. 2. A refinable place, $s \in S^R$, denoted by a bold circle, contains a formally simple "τ-net," N^τ, which consists of a unique input place, i_s, a unique output place, o_s, and transition τ_s, linked as shown. During token-controlled place refinement, i_s and o_s serve as "anchors" for the inserted refinement net, as shown in Fig. 3. The τ-transition represents a no-op transition and leaves the document unchanged. We use the symbol τ by analogy to τ-transitions in the calculus of communicating systems, CCS [19]. As we shall see, documents will pass through such a τ-transition if there has been no refinement, or if the document does not have permission to execute a refinement net attached to this refinable place.

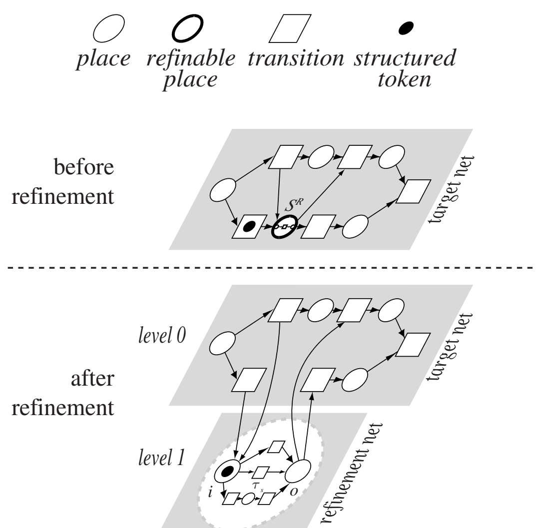

Fig. 3. Token-controlled process refinement occurs when a structured token encounters a refinable place. In this case, the control information in the token has been read, and implemented as a refinement net at the next level.

Transitions in workflow nets Since we consider documents that contain control information, we must use net representations that employ structured tokens. In our systems, network transitions will modify documents, and to capture that fact we build upon the predicate/transition net formalism [10, 9, 23]. Figure 4 illustrates a transition in such a net where lower-case Roman letters (e.g., x and y) represent documents and upper-case Roman letters represent the operations performed on single or multiple documents. Thus, a Japanese translation of document y might be denoted $J(y)$, the concatenation of documents x and y might be denoted $x; y$, and so forth.

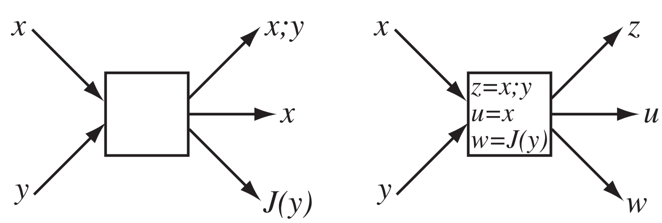

Fig. 4. Transitions in our workflow networks are based on those in traditional predicate/transition nets. In this example, documents x and y enter the transition. Operations are denoted by upper-case letters, with documents as arguments; a Japanese translation of y is denoted $J(y)$, the concatenation of the documents x and y is denoted by $x; y$, and so on, as shown on the left. Such processed documents — including ones subject to the *null* or "no-op" process, such as x above — are then emitted by the transition. A formally equivalent representation is to label the output arcs by symbolic names of the corresponding emerging documents, and write the transformations within the box, as shown on the right. This second representation emphasizes the fact that all information passing along an arc can be considered to be within a document. However, this second representation, as in colored Petri nets, obscures the structural information in documents, for instance that a particular document is a concatenation of two other documents.

In practice, some of these operations are automatic, such as "encrypt this electronic document," while others require human intervention, such as "translate this document into Japanese." While we acknowledge that these operations may be quite difficult to implement, here we need not specify them beyond attributing a label to each so they could be listed in a library and called as needed.

While colored Petri nets [13, 14] may have sufficient formal expressive power to serve as a foundation for our networks, colored nets are nevertheless awkward and unnatural for our needs because they obscure the structure of documents and their relationships. For instance, if two documents x and y are represented by tokens of differen colors, then the composite or concatenated document $x; y$ would be represented by a third color, thereby obscuring the document's composite structure. In fact, giving an equivalent predicate/transition net is often the most efficient way of specifying a colored net.

Predicate/transition nets We let Σ be a *signature*, a list of names of operations f and predicates p on documents, each of which has an associated *arity*, $a(f), a(p) \in \mathbb{N}$. For instance, Σ could contain a binary operator ";" which is interpreted as a concatenation of two documents, or a unary operator $J(\cdot)$, interpreted as a Japanese translation of the document argument. It could further have a binary predicate $Q(\cdot,\cdot)$, which when evaluated to `True` says that its first argument is a document containing an endorsement for the statements contained in its second document, or a unary predicate $K(\cdot)$, which when evaluated to `True` says that its argument document has been signed by the president.

We also let $V = \{x, y, \ldots\}$ be a set of *variables* ranging over documents. Then $\mathbb{T}(V, \Sigma)$ denotes the set of *terms* over V and Σ, such as for instance $J(x); y$, the Japanese translation of document x concatenated with the document y. Furthermore, $\mathbb{F}(V, \Sigma)$ is the set of *formulas* over V and Σ in the language of first-order logic, such as $(\exists y.\ Q(y, x)) \vee K(x)$, saying that the statements in document x either have an endorsement, or have been signed by the president.

A *predicate/transition net* over V and Σ is given as a quadruple (S, T, F, λ) where S and T are disjoint sets of *places* and *transitions*, $F \subseteq (S \times V \times T) \cup (T \times V \times S)$ is the *flow relation*, and $\lambda : T \to \mathbb{F}(\Sigma, V)$ allocates to each transition a first-order formula over Σ and V called the *transition guard* [23].

The elements of S and T are graphically represented by circles and boxes, respectively, while an element $(p, x, q) \in F$ is represented as an *arc* from p to q, labeled with variable x. The formulas $\lambda(t)$ are written in the transition t. An arc $(s, x, t) \in F$ with $s \in S$, $x \in V$ and $t \in T$ indicates that upon firing the transition t, a document x is taken from place s. An arc $(t, y, s') \in F$ with $t \in T$, $y \in V$ and $s' \in S$ indicates that upon firing t a document y is deposited in place s'.

The transition guard $\lambda(t)$ selects properties of the input documents that have to be satisfied for the transition to fire, and simultaneously specifies the relation between the input and the output documents. The variables allocated to the arcs leading to or from t may occur free in the formula $\lambda(t)$. They provide the means to talk about input and output documents of this transition. Consider as an example a transition that consumes input documents x and y, and produces an output document z. The transition guard could be a formula that says that the transition may only fire if x is a PGP document that successfully decrypts with the key presented in document y; if these conditions are met, the decrypted document z is emitted.

Marked predicate/transition nets and the firing rule Recall that Σ is a signature. A Σ-*algebra* is a domain $\mathcal{D}$ over which the operations and predicates of Σ are defined. In our case, of course, the elements of $\mathcal{D}$ denote documents. An *evaluation* $\xi : V \to \mathcal{D}$ over a Σ-algebra $\mathcal{D}$ assigns to every variable $x \in V$ a document $\xi(x) \in \mathcal{D}$. Such an evaluation extends in a straightforward manner to terms and formulas over V and Σ, where $\xi(t) \in \mathcal{D}$ for terms $t \in \mathbb{T}(V, \Sigma)$, and $\xi(\varphi)$ evaluates to `True` or `False` for formulas $\varphi \in \mathbb{F}(V, \Sigma)$.

A *marking* of a predicate/transition net over V and Σ is an allocation of tokens to the places of the net. As these tokens are documents that are represented by elements in a Σ-algebra $\mathcal{D}$, we speak of a marking *over* $\mathcal{D}$. A *marked* predicate/transition net over V, Σ and $\mathcal{D}$ is a tuple (S, T, F, λ, M) in which (S, T, F, λ) is a predicate/transition net over V and Σ, and $M : S \to \mathbb{N}^{\mathcal{D}}$ a *marking*, which associates with every place $s \in S$ a multiset of documents from the Σ-algebra $\mathcal{D}$. The multiset $M(s) : \mathcal{D} \to \mathbb{N}$ is a function that tells for every possible document how many copies of it reside in that place.

For two markings M and M' over $\mathcal{D}$ we write $M \leq M'$ if $M(s)(d) \leq M'(s)(d)$ for all $s \in S$ and $d \in \mathcal{D}$. The marking $M + M' : S \to \mathbb{N}^{\mathcal{D}}$ is given by

$$(M + M')(s)(d) = M(s)(d) + M'(s)(d).$$

Thus, the addition of two markings yields the union of the respective multisets of tokens in each place in the net. The function $M - M' : S \to \mathbb{Z}^{\mathcal{D}}$ is given by $(M - M')(s)(d) = M(s)(d) - M'(s)(d)$; this function need not always yield a marking because it might specify a negative number of documents in a place. For a transition $t \in T$ in a predicate/transition net and an evaluation $\xi : V \to \mathcal{D}$, the *input* and *output* markings ${}^\bullet t[\xi]$ and $t[\xi]^\bullet$ of t under ξ are given by

$${}^\bullet t[\xi](s) = \{\!\!\{\xi(x) \mid (s, x, t) \in F,\ x \in V\}\!\!\}$$

and

$$t[\xi]^\bullet(s) = \{\!\!\{\xi(x) \mid (t, x, s) \in F,\ x \in V\}\!\!\}$$

for $s \in S$, in which $\{\!\!\{, \}\!\!\}$ are multiset brackets. A transition t is *enabled* under a marking M over $\mathcal{D}$ and an evaluation $\xi : V \to \mathcal{D}$, written $M[t, \xi\rangle$, if ${}^\bullet t[\xi] \leq M$. In that case t can *fire* under M and ξ, yielding the marking $M' = M - {}^\bullet t[\xi] + t[\xi]^\bullet$, written $M[t, \xi\rangle\, M'$.

Workflow nets A *workflow net*, as defined by van der Aalst [2], is a Petri net (S, T, F) with two special places, i and o, that represent the input and output of the net. A workflow net does not have an initial marking. Instead, such a net acts on documents that are represented by tokens deposited by the environment in the input place i. Even if a net is finite and without loops, it may nevertheless represent an ongoing behavior, as the environment may continue to drop documents in the input place. The documents that arrive at the output place o are then carried away from there by the environment. Furthermore, i does not have incoming arcs (within the workflow net) and o does not have outgoing arcs (within the workflow net), and for every place $s \in S$ there should be a path in the net from i to o via s. Although in general it makes sense to consider multiple input and output places (as in Fig. 1), in the below discussions for simplicity we will follow van der Aalst in assuming just one of each.

Van der Aalst typically abstracts from the contents of documents by modeling them all as unstructured tokens. Here we give contents to documents by expanding place/transition nets to predicate/transition nets. Thus, a *predicate/transition workflow net* (over V and Σ) is a tuple (S, T, F, λ, i, o), and a *marked* predicate/transition workflow net (over V, Σ and $\mathcal{D}$) a tuple $(S, T, F, \lambda, i, o, M)$. The other parts of his definition are not affected.

An *initial marking* in a workflow net is a marking that puts only a single document in the input place of the net. In [2], workflow nets are often required to be *sound* in the sense that

- If an initial marking evolves into a marking that has a document in the output place, then there is only one document in the output place, and there are no documents left elsewhere in the net.
- If an initial marking can evolve into a marking M, then M can evolve further into a marking that has a token in the output place.
- There are no transitions in the net that can never be fired.

In van der Aalst [2], workflows are considered to be *case driven*, meaning that every case (started by a document token dropping into the input place) is executed in a fresh copy of the workflow net. This guarantees that documents in the workflow corresponding to different cases do not influence each other. Here we consider cases to be executed in parallel in the same workflow net, thereby creating the possibility for one case to influence the execution of another one. If we want the cases to be independent, we can augment each document token with a color or number, and require that transitions fire only when all incoming documents have the same color or number. Of course, each output then has the corresponding color or number. It is simple to implement the required bookkeeping tasks in predicate/transition nets.

Simple hierarchical workflow nets In our formalization of token-controlled refinement, a workflow net may have one or more refinable places, thereby becoming a tuple $(S, T, S^R, F, \lambda, i, o)$ with $S^R \subseteq S - \{i, o\}$ the set of refinable places. The input and output place are not refinable. It may be helpful to think of a refinable place $s \in S^R$ as consisting of an input place i_s, an output place o_s, and a transition τ_s, as indicated in Fig. 2. All arcs leading to s go to i_s, whereas all arcs out of s go out of o_s. The transition τ_s has i_s as its only input place and o_s as its only output place. The τ_s transition does not have any observable effect and passes documents through unchanged. In fact, the behavior of any net up to branching bisimulation equivalence [12] is unaffected under substitution of a net as in Fig. 2 for any place s.

A *simple hierarchical workflow net* is a tuple $(S, T, S^R, F, \lambda, i, o, R)$ with (S, T, F, λ, i, o) a predicate/transition workflow net, $S^R \subseteq S - \{i, o\}$ a set of *refinable places*, and R a function that associates with every refinable place $s \in S^R$ a simple hierarchical workflow net $R(s)$.[1] The *refinement net* $R(s)$ is inserted in

[1] As the nets $R(s)$ are simple hierarchical workflow nets themselves, a formal definition involves recursion: The class of *simple hierarchical Petri nets* is the smallest class $\mathbb{H}$ of tuples $N = (S, T, S^R, F, \lambda, i, o, R)$ with (S, T, F, λ, i, o) a predicate/transition workflow net, $S^R \subseteq S - \{i, o\}$ a set of *refinable places*, and $R : S^R \to \mathbb{H}$.

the *top-level net* N at the place s: whenever a document arrives in s (i.e., in i_s), the document is transferred to the input place of the net $R(s)$, and then $R(s)$ runs concurrently with the top-level net. When a document reaches the output place of $R(s)$ it is transferred back to s (in fact to o_s), and can be used as input for transitions that need a document in place s in the top-level net.

A *marked* simple hierarchical workflow is a tuple $(S, T, S^R, F, \lambda, i, o, M, R)$ with (S, T, F, λ, i, o) a predicate/transition workflow net, $S^R \subseteq S - \{i, o\}$ a set of refinable places, $M : S \to \mathbb{N}^{\mathcal{D}}$ a marking of the top-level net, and R a function that associates with every refinable place $s \in S^R$ a marked simple hierarchical workflow net $R(s)$. Under this definition, a *marking* of a simple hierarchical workflow net has itself a layer at each level of the hierarchy; M is just the top layer, whereas the other layers reside in R. Transitions in such a net can fire at every level of the hierarchy, following the definitions for predicate/transition nets given before, except that tokens that arrive in refinable places end up one level lower in the hierarchy, and that tokens arriving in output places of a refinement net end up one level higher.

A non-hierarchical predicate/transition workflow net can be regarded as the simple case of a hierarchical workflow net where $S^R = \emptyset$. Classical notions of *place refinement* in Petri nets can be regarded as methods to flatten hierarchical nets into non-hierarchical ones. Such a flattening operation on workflow nets is extremely easy to define: just insert the net $R(s)$ at s by identifying the input place of $R(s)$ with i_s and the output place with o_s, while deleting the transition τ_s. As we will explore ways to change the hierarchical structure of nets dynamically (in particular by undoing a refinement during the execution of a net), it is important to clearly separate the refinement net associated to a refinable place from the top-level net. Therefore we will not flatten the net during refinement, but instead work with hierarchical nets. The operational behavior of a hierarchical net however can best be understood by picturing the flattened net.

Hierarchical workflow nets A *hierarchical workflow net* is defined just as a simple hierarchical workflow net above, except that the refinement function R has more structure. In particular, R associates to every refinable place not just a single refinement net, but a *list* of guarded refinement nets. Here a *guarded* net is a pair of a guard and a net, the guard being a first-order logic formula over V and Σ, with a distinguished variable x, which is the only variable that may occur free in that formula. When a structured token $d \in \mathcal{D}$ arrives in s, the guard of the first guarded net in the list is evaluated by taking x to be d. If the guard evaluates to `True`, the token descends to the input place of the corresponding net. Otherwise, the second guard is tested, and so on. The last element in the list is always the τ-net (cf., Fig. 2) with a guard that always evaluates to `True`. When a token arrives in the output place of any net in the list specified by $R(s)$, it moves upwards to s for further use in the higher-level net.

Our aim in defining guards in this way stems from our recognition that a particular refinement may be useful only for documents of a particular type or from a particular author. The guards check in some way whether a document has "permission" to enter a particular refinement net. In case a document does not have permission to enter any meaningful refinement net associated to a certain place s, the structured token performs the τ-transition instead.

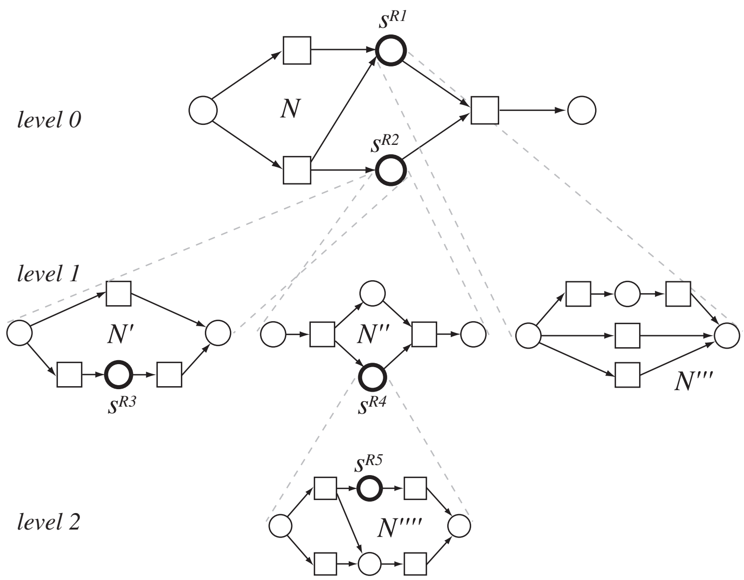

Fig. 5. Successive semi-permanent refinements lead to a hierarchical net. The target net N at the top has two refinable places. Here s^{R1} has been refined with the attachment of workflow net N''' and s^{R2} with both N' and N''. Finally, s^{R4} in N'' has been refined with N''''.

Figure 5 shows an example of a hierarchical net in which a single place has multiple refinement nets attached to it. The hierarchical structure of such a net can be represented by means of a bipartite tree, as indicated in Fig. 6. Each path from the root to a leaf consists of an alternating sequence of nets and refinable places.

Token-controlled refinement In order for our hierarchical workflow nets to allow token-controlled refinement, the refinement function $R(\cdot)$ will have even more structure than indicated above.

In active document workflow we allocate identifiers to refinable places, and documents carry instructions such as "when you land in a refinable place with this identifier, add the guarded net G to the list of guarded refinement nets." A refinable place s now carries an extra *refinement guard* $g_s(x)$ that checks whether an incoming document has permission to initiate a refinement.[2] Moreover, it has a *refinement net extractor*, $ref_s(x)$, that reads the instruction pertaining to the

[2] To avoid confusion, the guards discussed above will henceforth be called *entry guards*.

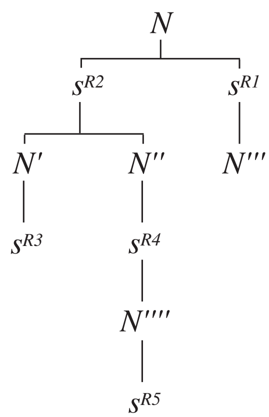

Fig. 6. The hierarchcial nets produced through token-controlled refinement, such as the one in Fig. 5, can be represented by a bipartite tree. In such a tree each path from the root, N, to a leaf consists of an alternating sequence of nets and refinable places.

refinement of s in a document x, and extracts the appropriate guarded refinement net from the document text. When, for an incoming document d, the refinement guard evaluates to `True` (i.e., $g_s(d)$ holds), and the refinement net extractor yields a guarded refinement net G (i.e. $\mathit{ref}_s(d) = G$), then this net is added to the head of the list in $R(s)$.

We also want to give documents the possibility to *remove* certain refinement nets. However, removal must not occur when that refinement net is still active, that is, when there is a token in the refinement net. Therefore we merely allow documents to change the entry guard of a refinement net to `False`, thereby preventing subsequent documents from entering that net. As soon as such a refinement net becomes inactive, in the sense that it can do no further steps, its role in the workflow comes to an end. We can eliminate idling "ghost nets" by incorporating a garbage collection process that regularly checks the workflow for inactive refinement nets with entry guards `False`, and removes them from the workflow.

In order to model token-controlled refinement fully, the R-component of a hierarchical net should have the structure $R(s) = (g_s(x), \mathit{ref}_s(x), L_s)$, where $g_s(x)$ is the refinement guard of s, $\mathit{ref}_s(x)$ is the refinement net extractor, and L_s is a list of *guarded refinement nets*, which are triples $(e(x), N, r(x))$ in which N is a refinement net, $e(x)$ is an *entry guard*, and $r(x)$ a *removal guard*. All guards are first-order logic formulas over V and Σ that have only the variable x occurring free. When a document d enters the refinable place s, first the removal guards of the guarded refinement nets in L_s are evaluated by substituting d for the free variable x. Each guarded refinement net $(e(x), N, r(x))$ for which $r(d)$ evaluates to `True` is earmarked for removal by assigning $e(x)$ to `False`. Next, the refinement guard is evaluated. In case $g_s(d)$ evaluates to `True`, the guarded refinement net $\mathit{ref}_s(d)$ is added to the head of the list L_s. Finally, the token d enters one of the refinement nets in L_s. To this end the token is evaluated by the entry guards one by one, and as soon as a guarded refinement net $(e(x), N, r(x))$ is encountered for which $e(d)$ evaluates to `True`, the token d is transferred to the input place of N.

In its initial state, any hierarchical workflow net has a refinement function R, such that for every refinable place s, the list L_s in $R(s)$ has only one guarded refinement net in it, namely $(\texttt{True}, N^\tau, \texttt{False})$. Because the removal guard of that τ-net always evaluates to `False`, that net will never be removed. Because that net's entry guard evaluates to `True`, this guarantees that every document will always succeed in entering the τ-net, should the document fail to enter any other refinement net.

The machinery above has been set up to facilitate semi-permanent token-controlled refinement. In order to achieve temporary refinement, in which only the token creating a refinement net may enter that net, the removal guard of the refinement net could be `True` (as well as the entry guard). This way, the next token that visits the refinable place will close that refinement net.

Summary We now summarize the formal definitions of the entities supporting active document workflow, based in part on traditional definitions and notations such as in [22, 21].

Places: A set S (German, "stellen"), whose elements are indicated by circles.
Transitions: A set T (German, "transitionen"), whose elements are indicated by squares.
Variables: A set V of symbolic names, to be instantiated by documents.
Flow relation: A relation $F \subseteq (S \times V \times T) \cup (T \times V \times S)$, indicated in a network by a set of directed arcs labeled with variables.
Input place: The unique place i in a workflow net that accepts tokens (documents) from the environment. (In more general nets, there could be multiple input places.)
Output place: The unique place o in a workflow net that emits tokens (documents) to the environment. (In more general nets, there could be multiple output places.)
Signature: A set Σ of names of n-ary operations and predicates on documents.
Terms: $\mathbb{T}$ over V and Σ.
Formulas: $\mathbb{F}$ over V and Σ, using the language of first-order logic.
Transition guards: Formulas $\lambda(t)$ allocated to each transition t.
Predicate/transition workflow net: A tuple (S, T, F, λ, i, o).
Tau net: A net $N_s^\tau = (S_s^\tau, T_s^\tau, F_s^\tau, \lambda_s^\tau, i_s, o_s)$, where $S_s^\tau = \{i_s, o_s\}$, $T_s^\tau = \{\tau_s\}$, $F_s^\tau = \{(i_s, x, \tau_s), (\tau_s, x, o_s)\}$, and $\lambda_s^\tau(\tau_s) = \texttt{True}$.
Entry guard: A formula $e(x)$ such that $e(d)$ tells whether document d has permission to enter a refinement net N.
Removal guard: A formula $r(x)$ such that $r(d)$ tells whether document d has both the intention and the permission to remove a refinement net N.
Guarded refinement net: A triple $(e(x), N, r(x))$ consisting of a refinement net with an entry and a removal guard.
Guarded τ-net: A tuple $(\texttt{True}, N_s^\tau, \texttt{False})$.

Refinable places: A set $S^R \subseteq S - \{i, o\}$, whose elements are indicated by bold circles.

Refinement guard: A formula $g_s(x)$ such that $g_s(d)$ tells whether document d has permission to refine place s.

Refinement net extractor: A term $\mathit{ref}_s(x)$, such that $\mathit{ref}_s(d)$ extracts from document d the guarded refinement net that according to that document should be inserted at place s.

Refinement function: A function R associates with each refinable place s a triple $R(s) = (g_s(x), \mathit{ref}_s(x), L_s)$, consisting of a refinement guard, a refinement net extractor, and a list L_s of guarded refinement nets, this list ending in the guarded τ-net.

Algebra of tokens: An algebra based on the set $\mathcal{D} = \{d_1, \ldots\}$ of tokens equipped with the operators and predicates of Σ.

Marking: The assignment $M : S \rightarrow \mathbb{N}^{\mathcal{D}}$ of structured tokens to places in a net.

Marked hierarchical workflow net: A net $N = (S, T, S^R, F, \lambda, i, o, M, R)$.

Transition firing in hierarchical active document workflows Firing in networks supporting active document workflow is more complicated than in traditional workflows, of course, because permissions, refinement and contraction are supported.

The firing of a transition t now entails the following:

1. for an evaluation ξ of the variables that makes $\lambda(t)$ `True`
   (a) extract documents $\xi(x)$ from place s for every $(s, x, t) \in F$
   (b) deposit documents $\xi(y)$ in place s for every $(t, y, s) \in F$
2. when a structured token d enters a refinement place $s \in S^R$:
   (a) for each guarded refinement net $(e(x), N, r(x))$ in L_s, if $r(d) =$ `True`:
       change $e(x)$ into `False`
   (b) evaluate $g_s(d)$. If $g_s(d) =$ `True`
       add guarded refinement net $\mathit{ref}_s(d)$ to the head of the list L_s in $R(s)$
   (c) go through the elements $(e(x), N, r(x))$ of L_s one by one until $e(d) =$ `True` and move d to the input place of N
3. when a structured token d enters an output of a guarded refinement net in $R(s)$
   (a) transfer that token one level up to s
   (b) continue executing the target net.

5 Future directions

In summary, we have identified a new yet general and powerful operation, token-controlled refinement, and have proposed modifications to predicate/transition nets based on that operation support enhanced workflow. Our formal definitions of elementary properties lay a foundation for more sophisticated work, both in the theory of concurrency and workflow applications. Ideally, we would like to prove the preservation under refinement of properties such as workflow soundness, as put forth by van der Aalst [2]. Alas, in general predicate/transition nets his soundness property need not be preserved under refinement, and thus neither can this be the case for our extensions to predicate/transition nets, at least not without significant restrictions to the expressive power of our nets. Future work, then, will focus on properties such as the preservation of liveness through token-controlled refinement, much as has been shown for the composition of Petri nets [17]. These are steps toward an concurrent formalization of expanded workflow that should have important practical applications.

References

1. Wil M. P. van der Aalst. Three good reasons for using a Petri-net-based workflow management system. In *Information and process integration in enterprises: Rethinking documents*, pages 161–182. Kluwer Academic, Norwell, MA, 1998.
2. Wil M. P. van der Aalst. Interorganizational workflows: An approach based on message sequence charts and Petri nets. *Systems analysis – Modelling – Simulation*, 35(3):345–357, 1999.
3. Wil M. P. van der Aalst, Jörg Desel, and Andreas Oberweis, editors. *Business Process Management: Models, Techniques, and Empirical Studies*. Springer, New York, NY, 2000.
4. Luca Aceto and Matthew Hennessy. Adding action refinement to a finite process algebra. *Information and Computation*, 115(2):179–247, 1994.
5. Eric Badouel and Javier Oliver. Reconfigurable nets, a class of high level Petri nets supporting dynamic changes. *(WFM) Workshop within the 19th International Conference on Applications and Theory of Petri Nets*, pages 129–145, 1999.
6. Michel Bauderon and Bruno Courcelle. Graph expressions and graph rewriting. *Mathematical Systems Theory*, 20(83–127), 1987.
7. Wilfried Brauer, Robert Gold, and Walter Vogler. A survey of behaviour and equivalence preserving refinement of Petri nets. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, number 483 in LNCS, 1991.
8. R. Kent Dybvig. *The Scheme programming language: ANSI Scheme*. Prentice Hall, Upper Saddle River, NJ, 1996.
9. Hartmann J. Genrich. Predicate/transition nets. In *Advances in Petri Nets 1986*, volume 254 of *LNCS*, pages 207–2471. Springer-Verlag, 1987.
10. Hartmann J. Genrich and Kurt Lautenbach. System modelling with high-level Petri nets. *Theoretical Computer Science*, 13(1):109–136, 1981.
11. Rob van Glabbeek and Ursula Goltz. Refinement of actions in causality based models. In Jaco W. de Bakker, Willem Paul de Roever, and Grzegorz Rozenberg, editors, Proceedings REX Workshop on *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness,* Mook, The Netherlands, May/June 1989, volume 430 of *Lecture Notes in Computer Science (LNCS)*, pages 267–300. Springer, 1990.
12. Rob van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.
13. Kurt Jensen. Coloured Petri nets and the invariant-method. *Theoretical Computer Science*, 14(3):317–336, 1981.
14. Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use, Vol. 1.* Springer–Verlag, 1992.

15. Donald E. Knuth. *TEX: The Program, Computers and Typesetting*. Addison-Wesley, Reading, MA, 1986.
16. Frank Leymann and Dieter Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, Upper Saddle River, NJ, 2000.
17. Michael Köhler, Daniel Moldt, and Heiko Rölke. Liveness preserving composition of agent Petri nets. Technical report, Universität Hamburg, Fachbereich Informatik, 2001.
18. Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling the structure and behaviour of Petri net agents. In José-Manuel Colom and Maciej Koutny, editors, *Applications and Theory of Petri Nets 2001*, pages 224–241, 2001.
19. Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, Cambridge, UK, 1999.
20. Ethan Mirsky and Andre DeHon. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In Peter Athanas and Kevin L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 157–166, 1996.
21. Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
22. Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and analysis with Petri nets*. Springer, Berlin, Germany, 1998.
23. Einar Smith. Principles of high-level Petri nets. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri nets I: Basic models*, volume 1491 of *Advances in Petri nets*, pages 174–210. Springer, 1998.
24. Robert Valette. Analysis of Petri nets by stepwise refinements. *Journal of Computer and System Sciences*, 18(1):35–46, 1979.
25. Rüdiger Valk. Self-modifying nets, a natural extension of Petri nets. In *Proceedings ICALP '78*, volume 62 of *Lecture Notes in Computer Science (LNCS)*, pages 464–476. Springer, 1978.
26. Rüdiger Valk. Petri nets as token objects: An introduction to elementary object nets. In Jörg Desel and Manual Silva, editors, *Application and Theory of Petri Nets*, volume 1420 of *Lecture Notes in Computer Science (LNCS)*, pages 1–25. Springer, 1989.
27. Patrick Henry Winston and Berthold Klaus Paul Horn. *Lisp*. Addison-Wesley, Reading, MA, third edition, 1988.