On the Relationship between Workflow Models and Document Types

Kees van Hee [a] Jan Hidders [b,*] Geert-Jan Houben [c]
Jan Paredaens [b] Philippe Thiran [d]

[a] *Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, the Netherlands*

[b] *Department of Mathematics and Computer Science, University of Antwerp, Antwerp, Belgium*

[c] *Department of Computer Science, Vrije Universiteit Brussel, Brussels, Belgium*

[d] *Louvain School of Management and University of Namur, Namur, Belgium*

Abstract

The best practice in information system development is to model the business processes that have to be supported and the database of the information system separately. This is inefficient because they are closely related. Therefore we present a framework in which it is possible to derive one from the other. To this end we introduce a special class of Petri nets, called Jackson nets, to model the business processes, and a document type, called Jackson types, to model the database. We show that there is a one-to-one correspondence between Jackson nets and Jackson types. We illustrate the use of the framework by an example.

Key words: workflow management system, Petri net, document management system, data type, information system design methodology

* Contact author: University of Antwerp, Department of Mathematics and Computer Science, Middelheimlaan 1, BE-2020 Antwerp, BELGIUM, tel. +32 3 2653873, fax. +32 3 2653777

Email addresses: k.m.v.hee@tue.nl (Kees van Hee), jan.hidders@ua.ac.be (Jan Hidders), geert-jan.houben@vub.ac.be (Geert-Jan Houben), jan.paredaens@ua.ac.be (Jan Paredaens), philippe.thiran@fundp.ac.be (Philippe Thiran).

1 Introduction

Data modeling and process modeling are two essential activities in requirements analysis and design of information systems. They are using different techniques and normally they are performed independently. Since both techniques are defining essential aspects of an information system they have to be integrated at some point in the development process, but normally this is at the level of programming. In this paper we show that data modeling and process modeling can go hand in hand from the beginning of the development process of so-called *case-based information systems*. The characteristic of these information systems is that they are developed to support the handling of *cases*, such as the treatment of a patient, the handling of an order or the delivery of a service. For each case type there is a *workflow* defining the tasks to be performed for the case. A workflow is a process with a clearly defined start and end state. In this paper we use a special class of Petri nets to model workflows, the so-called workflow nets [40]. Since in each task of the workflow something happens to the case, it is to be expected that the data type to record the case data is related to the structure of the workflow. The case data is recorded in the *case document*, the structure of which is a *document type*. We show that if we restrict ourselves to a special class of workflow nets, the so-called *Jackson nets*, then there is a tree shaped document type for the case data, called the *Jackson type*, that contains the same information as the workflow net. One of the main results in this paper is that there is a one-to-one correspondence between the document type and the workflow description, so from one we can derive the other. This is similar to the classical program design method of Jackson [16] which is the reason we called the workflow nets and the document types after this author.

The organization of the rest of this paper is as follows. In Section 2 we give the system development context for our work and we give a motivating example. In Section 3 we introduce Jackson types. In Section 4 we introduce the Jackson nets and in Section 5 we study the relationships between Jackson nets and Jackson types. In particular we prove that if two Jackson nets are derived from the same Jackson type they are isomorphic and that if it is possible to derive the same Jackson net from two different Jackson types, these types are algebraical equivalent. In Section 6 we continue with the motivating example. Here we show how we can derive an XML document type from a Jackson net and demonstrate how it provides a logical structure that helps the user to formulate queries over the cases of the workflow. In Section 7 we discuss the usefulness of Jackson types. Finally, we discuss related work in Section 8. The conclusion of the paper is given in Section 9.

2 Context and motivation

2.1 Historical perspective

In the requirements analysis and design phases of an information system we describe the desired functionality of a system from different perspectives. In the early stages of systems design, say until 1970, the systems designers started to describe the processes the system had to fulfil in terms of *flowcharts*. Since flowcharts describe only sequential processes (one thread of control) the interactions between processes was left out.

In the eighties the *data modeling* techniques became popular. Versions of the *entity relationship* model or the *relational* model were used for this. The big advantage of using this so-called *database-oriented* approach was that after the types of the data stores where established by a data model, several designers could model concurrently the processes that would act on the data stores . The modeling of the *operations*, i.e. of transformations on data objects, was done again at the low level of flowcharts or directly in a programming language.

In the nineties the *object-oriented* approach became popular. In this approach one tries to model the data aspect and the operations on the data in an *integrated* way. However the processes of a system were still second class citizens. Therefore *process-aware* information systems were identified as special class of systems [8]. This went so far that special software components were designed for the coordination of many interacting processes. Terms as "workflow management", "orchestration" and "choreography" are used to refer to this functionality. Special coordination engines were developed, for instance *workflow management systems*.

Modeling languages for the process appeared. They are also used to configure the coordination engines, like the database schema is a configuration parameter of a database management system. There are two families of formal languages for modeling processes: process algebra's and Petri-nets. Besides these there are several industry standards for modeling processes, such as BPEL (Business Process Execution Language) [29], UML activity diagrams [31] and BPMN (Business Process Modeling Notation) [32]. These languages allow us to design the process aspect of a system in isolation. These process modeling languages allow concurrency and so the problems of the days of the flowcharts were overcome.

The problem that we address is the integration of the different views: the data view and the process view. Already in the seventies there was a successful attempt to design the data and process aspect in an integrated way, JSP, Jackson's programming method [16] and later the method was lifted to the

level of system design, JSD, Jackson's development method [17]. (In *Software requirements and specifications* [18] an overview is presented.) In this approach *hierarchical* program structures where derived from the hierarchical input and output data structures, but they became out of fashion when the relational data model appeared. More recently UML also allows the specification of links between the process models and data models, but these models are here only loosely coupled and they remain essentially independent.

The programming method JSP was based on the idea that programs transform data streams into data streams. A data stream was a sequence of data elements and these data streams had a hierarchical data type. In fact, the data types of the input and output streams had to be describable by regular expressions composed of three kinds of operators: *sequential composition*, *selection* and *iteration*. The input and output data types were represented as so called *tree diagrams* and they were combined into one tree that represented the program structure. In fact, the program structure was also a tree diagram and the input tree and the output tree could be derived from the program tree by projections. The central idea of JSP was that the data structures determine the program structure. So JSP started with designing the input and output data structures. This idea is in line with the database oriented approach although in JSP hierarchical data structures are essential instead of the relational structure.

The similarity between the Jackson data structures and regular expressions was a reason to compare JSD, the development method based on JSP, with the language for communicating sequential processes, CSP, which can describe regular expressions as well. Therefore Sridhar and Hoare expressed JSD in CSP [39]. To our knowledge this was the first attempt to relate Jackson data structures and process structures in a fundamental way, but there was not much follow up from this attempt.

Another approach to formally integrate processes and data are colored Petri nets where tokens have values that may be changed by transitions [20]. The values are represented as colors and these colors can be linked to edges to indicate that only tokens with a certain color are consumed or produced through them. However, this approach does not offer a way to integrate the types of these colors into a global data model for the process as a whole.

The best practice today in information system development is to model the business processes that have to be supported and the database of the information system separately. This seems to be inefficient because they are often closely related. Like the observations of Jackson, we should try to exploit this relationship as much as possible.

Today there is a revival of hierarchical data structures as illustrated by the popularity of the many XML-based standards. There are several reasons for this. One is that hierarchical structures occur frequently in practice. For instance the bill of material of a physical artefact like bicycle or an airplane is a hierarchical structure. In the service industry we encounter also many hierarchical data structures, consider for instance the electronic patient record in health care, a bill of lading for a complex transport or the insurance portfolio of a company. In fact they all are described by a *document* and documents have hierarchical structures, composed with the operators: *sequence*, *selection* and *iteration*. In relational databases these documents are refined into their constituting elements and these are distributed over many tables. As soon as a document is needed the elements are retrieved from the tables and presented as a whole to the user who can update this view and restore it. From an implementation point of view this might be efficient, but from a conceptual point of view it is more natural to consider a document as one, structured, entity. The relational view is only interesting if management information is considered where a survey over different documents is needed.

Because documents are a natural concept for modeling data in business processes that produce physical artifacts or services, generic software components were developed to take care of documents, the *document management systems*. There is a natural relationship with workflow management systems, since both type of components are supporting (primary) business processes. In business process management [43] the processes and the data are equally important. The linking pin is what is called the *case*. A case is an instance of a case type and it is the "thing" that is moving through the business process. For instance in a bicycle factory the case is the construction of the bicycle from the order form till the final product. In a service organization like a hospital the case is the treatment of a patient, starting with its first visit till his final one (see Section 6).

There is often a case document that records everything that happened to the case, so the state of the process can be reconstructed from the case document and vice versa. This is not always necessarily the situation at the level of processes and document types, i.e., the document type does not contain a complete process description. There is however often a close relationship, e.g., the bill of material of a bicycle has a structure that reflects the construction process of the bicycle [36]. In this paper we define and study a class of models for which there is such a one-to-one relationship between document types and processes, namely the Jackson types and Jackson nets which are introduced in Section 3 and Section 4.

There are many Electronic Patient Record (EPR) systems that are used to record and plan the medical events in the treatment of a patient [15]. The focus of these systems is in registration of observations and decisions. Today medical *protocols* play an important role in the patient care processes. The protocols describe a care process that can be seen as the best practice. Medical experts have protocols for deriving a diagnosis as well as for a treatment. The traditional EPR systems are database-oriented and have little support for process control. In the Patient Care systems of the future the process control aspect will become more important and therefore the process knowledge should be integrated with the patient data. In fact a Patient Care system is a very good example of a case handling system, where we may consider the treatment of each medical problem as a different case. An alternative, that we do consider here is to view the whole life of a patient as one case.

As an illustration we consider a simplified care process of patient care in a hospital. The process is expressed as a Petri net in Figure 1. A formal definition of a (labeled) Petri net is given in Section 4.1. A Petri net is a bipartite graph with nodes of type *place* and nodes of type *transition*. A place indicates a possible *stage* or *phase* in the care process. A place may be marked with a *token*, which is in our situation a reference to the patient. A transition models an *event*, *activity* or *task* in the care process, and the label of the transition indicates the type of event. The *case* is here the patient. The set of all tokens belonging to one patient indicates the *state* of the patient. Note that a patient can be in different stages at the same time. So the stages a patient is in at some moment form its state.

Some transitions are only needed to describe the control flow and have no real task associated to it. This is the case with task 11: "Double test" and task 14: "End double test". Next we describe the meaning of the process model.

A patient who enters the hospital first goes to the reception desk (task 1: Patient identification). If the patient comes to the hospital for the first time, the patient's personal data is registered (task 3: New patient). This data consists of the patient's name and address (street, zip code and city) and a reference to its general physician. In case the patient is known to the hospital only an identity card is requested and the relevant personal data is fetched from the database (task 2: Known patient). Then the patient's problem is registered (task 4: Problem registration), a doctor is selected for a first examination and the patient receives an admission ticket that contains a number, the date and time of the admission.

After the patient has explained its problem, a preliminary diagnosis is made

(task 5: Preliminary diagnosis).

Depending on the outcome of this diagnosis, either Test 1, or Test 2, or both Test 1 and Test 2 in parallel, or both Test 1 and Test 2 in any order, or some treatment protocol is chosen from Protocols 1, 2 and 3. It may occur that no treatment is possible or needed, in which case the patient leaves the hospital and some administration is performed (task 16: Exit). Examples of tests are laboratory tests like urine or blood tests and image generation like X-ray or a MRI-scan. Today there are many protocols for medical treatment. Protocols may consist of tests as well as therapies and may be refined to sub-processes.

All tests result in data of the same type: the type of result (chosen from the official list of activity types from the hospital), the date, and the resulting values (outcomes) of the analysis.

After the tests or protocols have been executed they are evaluated in a new diagnosis (task 15: Diagnosis). Depending on the outcome of this diagnosis, a selection of further activities is made. This is repeated until the decision is made that further treatment is not useful anymore.

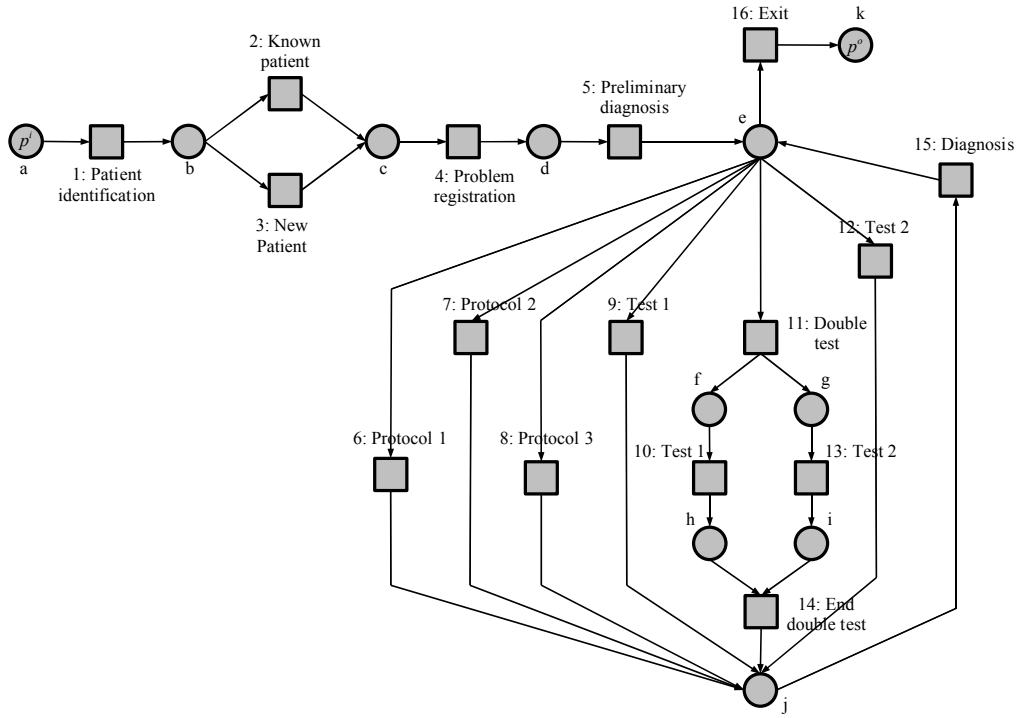


Fig. 1. A workflow for handling a medical problem

There is for each patient (case) a *dossier* which is the EPR. Two typical instances are displayed in Figure 2. The dots represent data entered by the medical experts, and may include observations, decisions or any data involved in the event. The first dossier starts with the information for identifying the patient, the registration of the new patient, the registration of the problem and

the result of the preliminary diagnosis. Then there is a list of treatments and finally the registration of the exit of the patient. The list of treatments consists here of three treatments all ending with a diagnosis. In the final treatment we see that the double test is applied and so the information involved in preparing the two tests, the two tests themselves and the combination of the test results is stored. In the second dossier we see largely the same type of information except that here the patient is registered as a known patient and the list of treatments consists of the double test followed by the protocol3 test. It is not hard to see how the data structure of such dossiers can often be described by a type consisting of recursively nested records and lists.

Observe that the relative vertical and horizontal orientation of the steps in the dossiers has meaning here: a step that is just below another step describes an event that followed the event of the step just above it, and steps that are next to each other describe events that were executed in parallel. This relationship between the parts of the dossier may determine how the dossier is allowed to grow. For example, the information for "preliminary diagnosis" may not be entered before the information for "problem registration" is entered, but for the double test the information for "test2" may be entered before that of "test1". Therefore we extend the notion of type such that it also captures these relationships and we investigate the precise relationship between such types as a workflow description formalism and certain workflow nets.

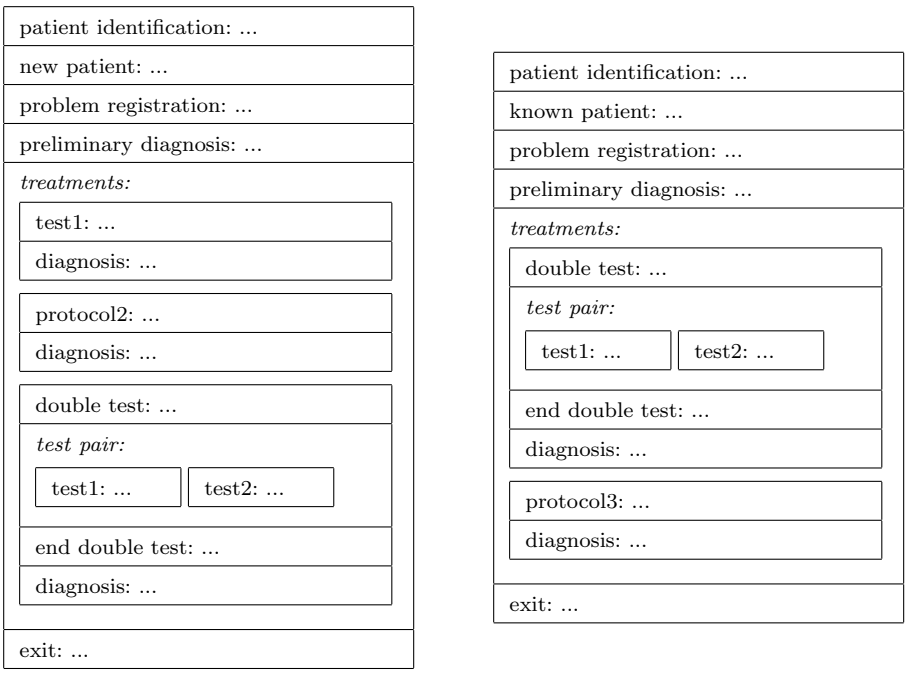


Fig. 2. Examples of two patient dossiers

In the presented example the different pieces of information are associated with the firing of transitions, i.e., each firing of a transition generates some information that is to be stored in the patient dossier. It can however in

some cases be more natural to think of the information as being associated with the tokens, for example if the token represents a document containing a diagnosis or a form that contains the result of a test. Therefore we assume in the following of the paper that information can be associated both with the firing of a transition and with the tokens that are consumed and produced.

3 Jackson Types

In this section we introduce types that we use to represent workflow document types, i.e., data structures that can contain all the information that is involved in a single case of the workflow that is described by a workflow net. We show that these types (1) can indeed contain all the involved information and (2) have a natural correspondence to the hierarchical structure of the workflow net.

We postulate a set of atomic types $\mathcal{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots\}$ that describe data structures that contain all the information involved in a certain transition or place of a workflow net. Note that these atomic types are only atomic for the purpose of describing the workflow document type and may in a later phase of the modeling process be broken down into smaller components. From these atomic types we construct types by using constructors for sequencing (;), parallelism ($\|$), choice (+), and loop (#).

Definition 1 (Type) *The set of* types J *is defined by the following syntax:*

$$J ::= \mathcal{A} \mid (J \ ; \ J) \mid (J \ \| \ J) \mid (J \ + \ J) \mid (J \ \# \ J).$$

The types can be thought of as a combination of a data type and a process specification. The type $(\tau_1; \tau_2)$ denotes the type of ordered records. This type describes records with fields of type τ_1 and τ_2 and indicates that in the process the event associated with the field of type τ_1 precedes the event associated with the field of type τ_2. The type $(\tau_1 \parallel \tau_2)$ denotes the type of unordered records with fields of type τ_1 and τ_2 that describes records with fields of type τ_1 and τ_2 and indicates that in the proces there is no particular order. The type $(\tau_1 + \tau_2)$ denotes the type of variant records that contain either a value of type τ_1 or τ_2. Finally the type $(\tau_1 \# \tau_2)$ denotes nonempty lists of values of type τ_1 separated by values of type τ_2.

The notion of trace set is introduced to formalize the concept of all information that is involved in a single run of a workflow. Here a single trace is a string of atomic types and a trace set is a set of such strings. If α and β are such strings then we will denote the concatenation of α and β as $\alpha \cdot \beta$.

The trace set that is associated with a certain type is defined as follows.

Definition 2 (Trace-set of Types) *The* trace-set of a type τ, $Tr(\tau)$ *is defined by induction upon the structure of* τ *as follows:*

- $Tr(\tau) = \{\tau\}$ *if* $\tau \in \mathcal{A}$
- $Tr((\tau_1; \tau_2)) = \{\alpha \cdot \beta \mid \alpha \in Tr(\tau_1), \beta \in Tr(\tau_2)\}$
- $Tr((\tau_1 \parallel \tau_2)) = \{\alpha_1 \cdot \beta_1 \cdot \ldots \cdot \alpha_k \cdot \beta_k \mid k \geq 0, \alpha_1 \cdot \ldots \cdot \alpha_k \in Tr(\tau_1), \beta_1 \cdot \ldots \cdot \beta_k \in Tr(\tau_2)\}$
- $Tr((\tau_1 + \tau_2)) = Tr(\tau_1) \cup Tr(\tau_2)$
- $Tr((\tau_1 \# \tau_2)) = \{\alpha_1 \cdot \beta_1 \cdot \alpha_2 \cdot \ldots \cdot \beta_{n-1} \cdot \alpha_n \mid n > 0, \alpha_i \in Tr(\tau_1), \beta_i \in Tr(\tau_2)\}$

We have for example

- $Tr(((\mathsf{a}; \mathsf{b}) + \mathsf{c})) = \{\mathsf{ab}, \mathsf{c}\}$
- $Tr((\mathsf{a}; (\mathsf{b} \# \mathsf{c}))) = \{\mathsf{ab}(\mathsf{cb})^n \mid n \geq 0\}$
- $Tr((\mathsf{b} + \mathsf{d})) = \{\mathsf{b}, \mathsf{d}\}$
- $\{\mathsf{abdcbcb}, \mathsf{abcbb}, \mathsf{abb}\} \subset Tr((\mathsf{a}; (\mathsf{b} \# \mathsf{c})) \parallel (\mathsf{b} + \mathsf{d}))\ )$

Remark that $\mathsf{a} \# \mathsf{b}$ stands for $(\mathsf{a}; \mathsf{b})^*; \mathsf{a}$, using the Kleene-star.

Definition 3 (Trace Equivalence of Types) *Two types* τ *and* τ' *are called* trace equivalent*, denoted as* $\tau \equiv_{tr} \tau'$*, iff* $Tr(\tau) = Tr(\tau')$*.*

Theorem 4 *There is no finite set of equivalence rules that defines the trace equivalence of types.*

Proof. Let us assume that there is such a finite set of equivalence rules. Then this set of rules will also define trace equivalence if there is only one letter in the alphabet. Under this assumption $e_1 \parallel e_2 \equiv_{tr} e_1 + e_2$, so there is also such a set of rules for expressions that do not contain $\parallel$. We can express the # operator with the Kleene-plus (denoted e^+) and vice versa, because $e^+ \equiv_{tr} (e \# e) + (e; (e \# e))$ and $e_1 \# e_2 \equiv_{tr} e_1 + (e_1; (e_2; e_1)^+)$. It follows that there is also such a set of rules for the language with the Kleene-plus but without #. There is also such a set of rules if we add the empty string (denoted as ε) since we can rewrite every expression to either a ε-free e or $\varepsilon + e$ with e ε-free by using only a finite set of equivalence rules. There will then also be such a set of rules for the language with the Kleene-plus replaced with the Kleene-star (denoted e^*) since one can be expressed with the other, and vice versa: $e^* \equiv_{tr} \varepsilon + e^+$ and $e^+ \equiv_{tr} e; e^*$. Note that the resulting language is exactly the language of regular expressions. However, for that language it has been shown by Aceto, Fokkink and Ingólfsdóttir [1] that such a finite set of rules does not exist, even under the assumption that there is only one symbol in the alphabet. □

Conjecture 5 *Deciding trace inequivalence of types is EXPSPACE complete.*

The problem is very similar to the problem of deciding trace inequivalence of regular expressions extended with interleaving operations, which was shown to be EXPSPACE complete by Mayer and Stockmeyer [25].

Next to trace equivalence we also define another coarser notion of equivalence that can be informally thought of as defining when two types represent the same data type. For example the types $(\mathsf{a};(\mathsf{b};\mathsf{c}))$ and $((\mathsf{a};\mathsf{b});\mathsf{c})$ can be seen as representations of the type $(\mathsf{a};\mathsf{b};\mathsf{c})$, i.e., the type of ordered tuples with the fields a, b and c in that order. Another example are $(\mathsf{a} \parallel \mathsf{b})$ and $(\mathsf{b} \parallel \mathsf{a})$ which both represent the type of unordered tuples with the fields a and b. This leads to the following definition.

Definition 6 (Algebraic Equivalence of Types) *The* algebraic equivalence $\equiv_{alg}$ *is the smallest equivalence relation on the set of types that fulfils the identities of Figure 3.*

$$
\begin{array}{rcl}
(\tau_0 \ ; \ \tau_1) \ ; \ \tau_2 & \equiv_{alg} & \tau_0 \ ; \ (\tau_1 \ ; \ \tau_2) \\
(\tau_0 \ \parallel \ \tau_1) \ \parallel \ \tau_2 & \equiv_{alg} & \tau_0 \ \parallel \ (\tau_1 \ \parallel \ \tau_2) \\
\tau_0 \ \parallel \ \tau_1 & \equiv_{alg} & \tau_1 \ \parallel \ \tau_0 \\
(\tau_0 \ + \ \tau_1) \ + \ \tau_2 & \equiv_{alg} & \tau_0 \ + \ (\tau_1 \ + \ \tau_2) \\
\tau_0 \ + \ \tau_1 & \equiv_{alg} & \tau_1 \ + \ \tau_0 \\
(\tau_0 \ \# \ \tau_1) \ \# \ \tau_2 & \equiv_{alg} & \tau_0 \ \# \ (\tau_1 \ + \ \tau_2)
\end{array}
$$

Fig. 3. Defining identities for algebraic equivalence

Note that the identity between $\tau_0 \# (\tau_1 \# \tau_2)$ and $(\tau_0 \# \tau_1) \# \tau_2$ is not included since these two types might not even be trace equivalent. For example, the trace aca is in $Tr(((\mathsf{a}\#\mathsf{b})\#\mathsf{c}))$ but not int $Tr((\mathsf{a}\#(\mathsf{b}\#\mathsf{c})))$, and the trace abcba is in $Tr((\mathsf{a}\#(\mathsf{b}\#\mathsf{c})))$ but not in $Tr(((\mathsf{a}\#\mathsf{b})\#\mathsf{c}))$. The definition of the notion of algebraic equivalence of types will be further motivated later on in the paper where it is shown that for a certain non-deterministic procedure that derives types for a certain class of workflow nets it captures exactly the ambiguity of this procedure, i.e., there may be more than one possible result type but they are all algebraically equivalent.

That algebraic equivalence is indeed coarser then trace equivalence is established by the following theorem.

Theorem 7 *For two types τ and τ' it holds that $\tau \equiv_{tr} \tau'$ if $\tau \equiv_{alg} \tau'$ but not conversely.*

Proof. In order to prove the if-part we have to prove that $\tau \equiv_{alg} \tau'$ implies $\tau \equiv \tau'$ for each of the seven rules of Figure 3. For the first five rules this is trivial. For the sixth rule we have $Tr((\tau_0 \# \tau_1) \# \tau_2) = \{\alpha_1^1 \cdot \beta_1^1 \ldots \beta_{n_1-1}^1 \cdot \alpha_{n_1}^1 \cdot \gamma_1 \ldots \gamma_{k-1} \cdot {\alpha_1}^k \cdot {\beta_1}^k \ldots \beta_{n_k-1}^k \cdot \alpha_{n_k}^k \mid n_i, k > 0, \alpha_i^j \in Tr(\tau_0), \beta_i^j \in Tr(\tau_1), \gamma_i \in$

$Tr(\tau_2)\} = \{\alpha_1 \cdot \delta_1 \ldots \delta_{m-1} \cdot \alpha_m |\ m > 0, \alpha_i \in Tr(\tau_0), \delta_i \in Tr(\tau_1) \cup Tr(\tau_2)\} = Tr(\tau_0 \ \# \ (\tau_1 + \tau_2))$.

That the converse does not hold follows from Theorem 4 but for illustration we will also give a counterexample. Let $\mathsf{a} \in \mathcal{A}$ then clearly $Tr((\mathsf{a}\#\mathsf{a})\#\mathsf{a}) = Tr(\mathsf{a}\#\mathsf{a}) = \{\mathsf{a}^{2n+1} \mid n \geq 0\}$. Hence $\mathsf{a}\#\mathsf{a} \equiv_{tr} (\mathsf{a}\#\mathsf{a})\#\mathsf{a}$. On the other hand $\mathsf{a}\#\mathsf{a} \not\equiv_{alg} (\mathsf{a}\#\mathsf{a})\#\mathsf{a}$ since no identity of Figure 3 can be applied to $\mathsf{a}\#\mathsf{a}$. □

In this paper we will mostly consider a specific subset of types that correspond with a certain class of Petri nets that describe workflows. This causes certain restrictions on the types because the atomic types associated with the places and transitions need to alternate properly in the type since places are followed by transitions and vice versa. Moreover, it also restricts the operators allowed in certain places of the type. For example, after a basic type associated which a transition we can have the $\|$ operator but not the $+$ operator since a transition can define an AND-split but not an OR-split. Likewise, after a basic type associated with a place there can be a $+$ operator but not a $\|$ operator, since a place can define an OR-split but not an AND split. The restricted set of types is called the set of *Jacskon types* and defined given two sets, $\mathcal{A}^t$ and $\mathcal{A}^p$, which are defined such that $\mathcal{A} = \mathcal{A}^t \cup \mathcal{A}^p$ and represent the atomic types that can be associated with transitions and with places, respectively.

Definition 8 (Jackson Type) *The set of* Jackson types *is described by the following syntax of* J^0*:*

$$J^0 ::= \mathcal{A}^p \mid (\mathcal{A}^p; (J^t; \mathcal{A}^p)).$$
$$J^t ::= \mathcal{A}^t \mid (J^t; (J^p; J^t)) \mid (J^t + J^t).$$
$$J^p ::= \mathcal{A}^p \mid (J^p; (J^t; J^p)) \mid (J^p \parallel J^p) \mid (J^p \# J^t).$$

Note that the Jackson types are indeed a subset of the set of types. Clearly $(\mathsf{a}; (\mathsf{b} + \mathsf{c}); \mathsf{b})$ and $(\mathsf{a}; ((\mathsf{a}; \mathsf{b}; \mathsf{a}) + \mathsf{a}); \mathsf{a})$ are Jackson types, while $((\mathsf{a}; (\mathsf{b}\#\mathsf{c})) \parallel (\mathsf{b} + \mathsf{d}))$, $(\mathsf{a}; (\mathsf{a} \parallel \mathsf{b}); \mathsf{a})$ and $((\mathsf{a}; \mathsf{b}) + \mathsf{c})$ are not.

4 Jackson Nets

4.1 Petri nets and workflow nets

We start with the basic terminology for Petri nets and workflow nets in particular. Next we will define the subtype of Jackson nets.

Definition 9 (Labeled Petri Net) *A* labeled Petri net *is a tuple* (P, T, F, λ)

with P a set of places, T a set of transitions ($P \cap T = \emptyset$) and $F \subseteq (T \times P) \cup (P \times T)$ the flow relation. The function λ associates a label to each place and transition.

Note that λ is not required to be injective and can therefore map different places and transitions to the same label. Given a labeled Petri net (P, T, F, λ) and a transition $t \in T$ we let $\bullet t$ and $t\bullet$ denote input places and output places of t, i.e., $\bullet t = \{p \mid (p, t) \in F\}$ and $t\bullet = \{p \mid (t, p) \in F\}$. Similarly, for a place p we let $\bullet p$ and $p\bullet$ denote the producing transitions and consuming places, i.e., $\bullet p = \{t \mid (t, p) \in F\}$ and $p\bullet = \{t \mid (p, t) \in F\}$.

Definition 10 (Graph of a labeled Petri Net) *The* graph of a labeled Petri net (P, T, F, λ) *is its underlying directed graph $G = (P \cup T, F)$.*

Definition 11 (Workflow Net) *A* workflow net or net *is defined as a tuple $\Omega = (P, T, F, p^i, p^o, \lambda)$ such that*

- *(P, T, F, λ) is a labeled Petri net;*
- *$p^i \in P$ is the* input place *such that $\bullet p^i = \emptyset$;*
- *$p^o \in P$ is the* output place *such that $p^o\bullet = \emptyset$; and*
- *in the graph of Ω there is for each node n a directed path from p^i to n and a directed path from n to p^o.*

Workflow nets are represented in the straightforward way. In Figure 4 four workflow nets are shown.

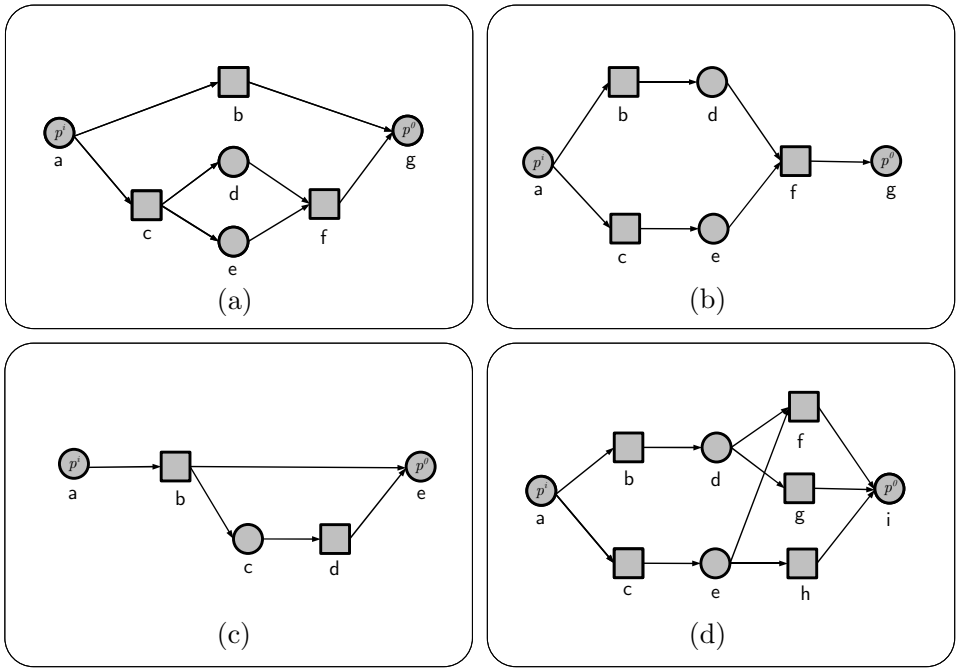


Fig. 4. Four workflow nets

Definition 12 (Marking) *Given a net* $\Omega = (P, T, F, p^i, p^o, \lambda)$ *a* marking *is a function* $m : P \to \mathbb{N}$.

If P' is a set of places in Ω then we let P' denote the marking $m : P \to \mathbb{N}$ that is defined such that $m(p) = 1$ if $p \in P'$ and $m(p) = 0$ if $p \notin P'$. Markings for a certain net can be added and subtracted: $m_1 + m_2$ $(m_1 - m_2)$ is the marking m' such that $m'(p) = m_1(p) + m_2(p)$ $(m'(p) = m_1(p) - m_2(p))$. Note that $m_1 + m_2$ is always defined, but $m_1 - m_2$ is defined iff $m_1(p) \geq m_2(p)$ for all $p \in P$. The product of a natural number k and a marking m, denoted as $k \cdot m$, is defined such that for all $p \in P$ it holds that $(k \cdot m)(p) = k \cdot m(p)$. We say that a transition $t \in T$ is *enabled* in a marking m if it holds that $m - {}^\bullet t$ is defined.

Definition 13 (Reachability Graph) *Given a net* $\Omega = (P, T, F, p^i, p^o, \lambda)$, *we define its* reachability graph *as an edge-labeled graph* (V, E) *such that*

(1) V *is the set of all markings for* Ω, *and*
(2) $E \subseteq V \times T \times V$ *such that* $(m_1, t, m_2) \in E$ *iff*
  (a) t *is enabled in* m_1 *and*
  (b) $m_2 = m_1 - {}^\bullet t + t^\bullet$.

In addition, we define two special markings: m^i, called the *initial marking*, that places one token in place p^i and nowhere else, and m^o, called the *final marking*, which puts one token in p^o and nowhere else. A path in the reachability graph is called a *transition path*. A *run* is defined as a nonempty transition path that starts from m^i. Such a run is said to be a *full run* if the last edge ends in m^o. A *firing sequence* of the net Ω is the sequence of transition labels of transitions as they are encountered in a full run. (Note that normally the term firing sequence is used for the sequences of transitions.) For example, b and cf are the firing sequences of the workflow net in Figure 4 (a).

The notion of workflow net is often accompanied by a notion of soundness that excludes certain types of anomalies. Consider for example workflow net (b) in Figure 4. If we start with one token in the place labeled a then the transition labeled b and the transition labeled c are enabled. If either one of these transitions fires then there is either a token in the place labeled d or in the place labeled e, but not both, so the transition labeled f is not enabled and the workflow cannot finish properly, i.e., reach a state with only one token in p^o. A similar problem is demonstrated in workflow net (c) which, when starting with one token in p^i, always ends with two tokens in p^o. To prevent this we require that sound workflow nets can always terminate properly, i.e., for every marking reachable from m^i we can reach the final marking m^o. Another type of anomaly is demonstrated in workflow net (d) which always finishes properly, but it contains a transition labeled f which will never be enabled because there is in every reachable marking either a token in place d or place e but never

in both. The transition labeled f is therefore superfluous and could have been omitted from the workflow net. Therefore we also require for sound workflow nets that every transition is enabled in at least one reachable marking. This leads to the following definition.

Definition 14 (Sound Net) *A net* $\Omega = (P, T, F, p^i, p^o, \lambda)$ *is said to be* sound *if it holds in the reachability graph of* Ω *that*

(1) from every marking reachable from m^i*, we can reach* m^o *and*
(2) for every transition $t \in T$ *there is a run with an edge* (m_i, t, m_j)*.*

Remark that in a sound net m^o is the only marking that (a) is reachable from m^i and (b) has a token in place p^o. In Figure 4 the workflow net (a) is indeed sound, and the nets in (b) and in (c) are not, since m^o is not reachable, and (d) is also not sound because the transition labeled f will never be enabled.

4.2 Jackson nets and soundness

From now on we suppose that the places and the transitions of nets are labeled by a Jackson type. The intuition behind this association of Jackson types and nets is that thus we can integrate process and data aspects. If all the labels of the net are atomic types and hence belong to $\mathcal{A}$ we call it an *atomic net*.

We introduce the semantics of a net by defining its trace-set. A trace of a net can be informally described as a sequence of the Jackson types of the places and transitions in the order that they are visited or fired. The formal definition of the traces of a net is based on the notion of firing sequence which, we recall, is defined as the sequence of transition labels of transitions as they are encountered in a full run. For an illustration consider the first workflow net in Figure 5 for which the set of firing sequences can be described by the regular expression $(\mathsf{bg} + \mathsf{c}(\mathsf{jl})^*\mathsf{h})$. Clearly this is not the desired notion of trace since it ignores the labels of the places. To remedy this we introduce the notion of place-expanded net which informally can be defined as the net that is obtained by splitting every place into two places and an intermediate transition.

Definition 15 (Place-expanded Net) *Given a net* Ω *we define its associated* place-expanded *net* $\hat{\Omega}$ *as the net that is obtained by replacing each place* p *by two new places* p_1 *and* p_2 *that are connected by one new transition* t_1*. The places* p_1 *and* p_2 *and the new transition* t *get the label of* p *and the incoming edges of* p *are copied to* p_1 *and the outgoing edges of* p *are copied to* p_2*.*

In Figure 5 the bottom net is the associated place-expanded net of the top net. It's set of firing sequences is described by the regular expression $(\mathsf{a}((\mathsf{b}(\mathsf{de} + \mathsf{ed})\mathsf{g}) + (\mathsf{cfh}(\mathsf{jklf})^*))\mathsf{i})$. It is this set that seems to correctly model the traces of

the top net in the sense that it takes both the labels of the places an transitions into account. This leads to the following formal definition.

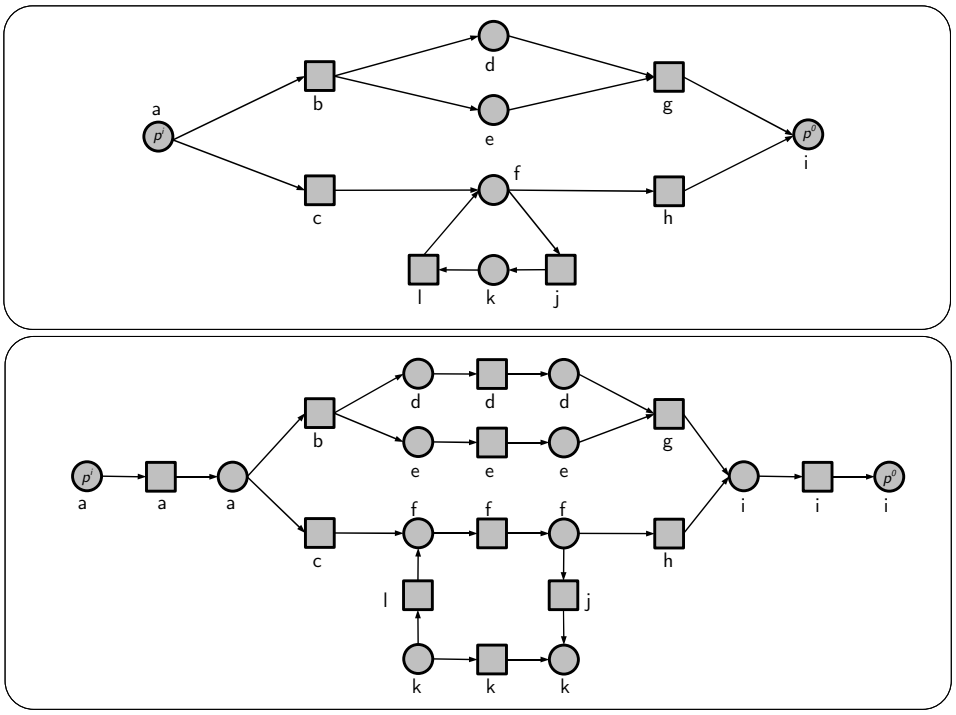


Fig. 5. A net with its place-expanded net

Definition 16 (Trace-set of Nets) *Given a net* Ω*, with its place-expanded net* $\hat{\Omega}$*. A* trace *of* Ω *is a firing sequence of* $\hat{\Omega}$*. The set of traces of* Ω *is denoted as* $Tr(\Omega)$*.*

Observe that the trace-set of the top net of Figure 5 is equal to the trace-set of the type $(\mathsf{a};((\mathsf{b};(\mathsf{d}\|\mathsf{e});\mathsf{g})+(\mathsf{c};(\mathsf{f}\#(\mathsf{j};\mathsf{k};\mathsf{l}));\mathsf{h}));\mathsf{i})$. This type arguably corresponds more closely to the structure of this net than the previously presented regular expression describing the same set. It is this correspondance that is one of the fundamental properties of Jackson types that we investigate more closely in the remainder of this paper.

Next, we give five rules R1,..., R5, displayed Figure 6 to generate nets starting with only one place. We say that Ω *generates* $\tilde{\Omega}$ iff $\tilde{\Omega}$ can be obtained from Ω by applying zero or more times a rule of Figure 6 [1] , without applying rules R3 and R4 to the input place or the output place. Moreover, if rule R1 is applied to the input (output) place then p_2 (p_3) becomes the new input (output) place. We also say that $\tilde{\Omega}$ can be *reduced* to Ω. To apply the rules note that the label of the place or transition to be refined has to satisfy a structure that is reflected in the equation of the rule. So for example, rule R1, which is denoted

[1] In Rule $R1$, p_1 is the input place iff p_2 is the input place; p_1 is the output place iff p_3 is the output place.

by $\lambda(p_1) = (\lambda(p_2); \lambda(t_1); \lambda(p_3))$, means that the label of place p_1 has at the top-level the structure of a sequence and therefore it may be expanded into a sequence of a place (p_2) a transition (t_1) and again a place (p_3), each with its own label $\lambda(p_2)$, $\lambda(t_1)$ and $\lambda(p_3)$ respectively.

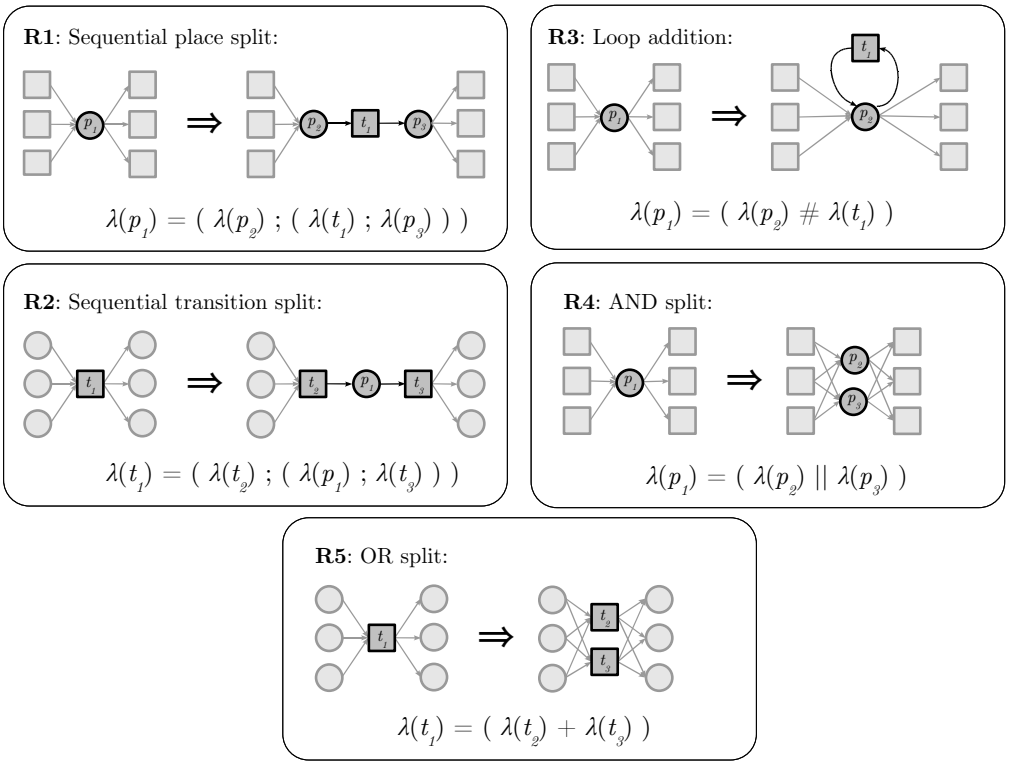


Fig. 6. The generation rules for Jackson nets

Definition 17 (Jackson Net) *We call a net without transitions and only one place labeled by a Jackson type an* singleton net. *A* Jackson net Ω *is a net that can be generated, from a singleton net, by applying the rules* $R1, .., R5$ *recursively, starting with type* τ *in the singleton net. We say that the Jackson net* Ω *is* generated *by* τ.

Remark that the net of Figure 4 (a) is a Jackson net. Its generation is given in Figure 7. The other nets (b), (c) and (d) in the same Figure are not Jackson nets. The (a) net is also the only sound net in this figure. As is shown by Theorem 18 it holds that every Jackson net is a sound net, but the converse does not hold as is demonstrated in Theorem 19 where we show that the sound net in Figure 8 is not a Jackson net.

The *is-generated-by* relationship between Jackson types and Jackson nets is defined by a non-deterministic rewriting process, i.e., at one point in the process it can be that multiple rewrite rules apply and we have to make an arbitrary choice. This relationship is therefore not necessarily a function and may associate several Jackson nets with the same Jackson type. The same

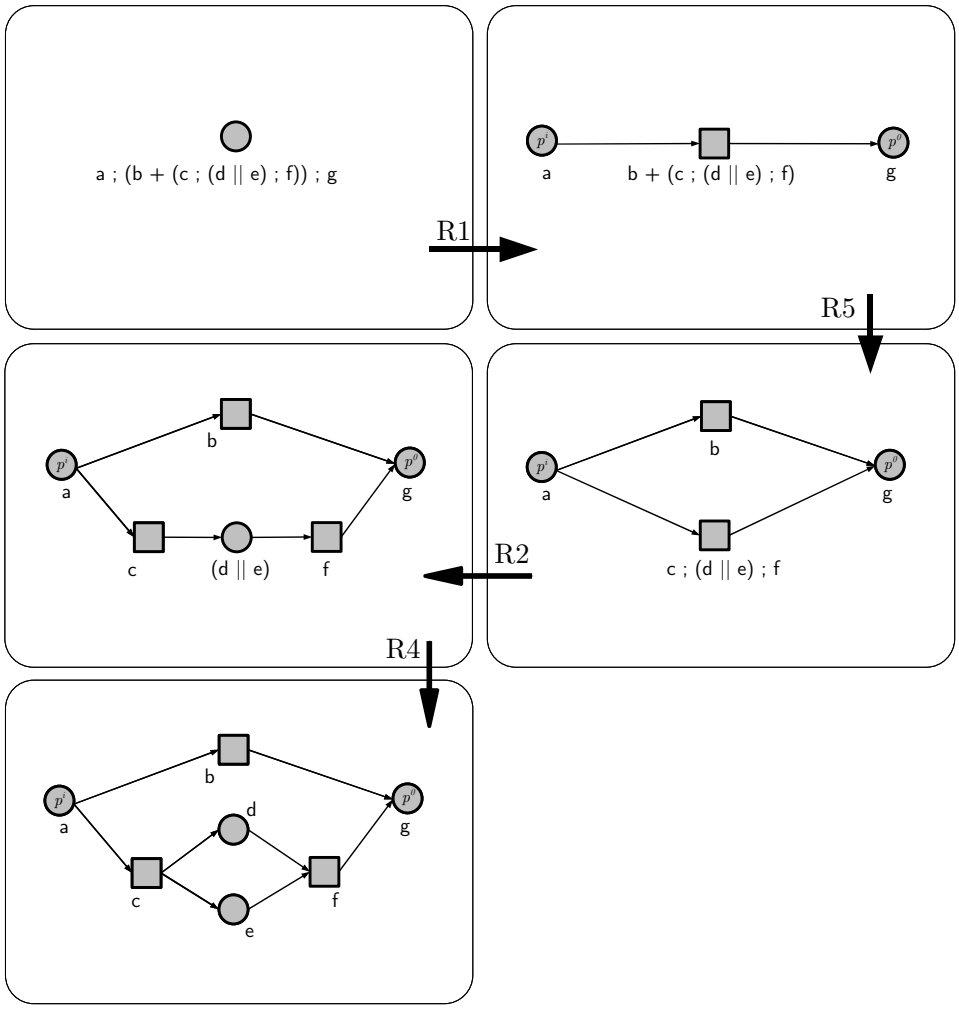


Fig. 7. Generation of a net

holds for the reverse *is-generated-from* relationship, which can be assumed to be defined by the same rewriting process in reverse. So with a certain Jackson net there may be more than one Jackson type that generates it. However, as is discussed in Section 5, the relationship is in fact very close to a one-to-one relationship.

Rules such as those in Figure 6 were studied by Berthelot in [3] and Murata in [27] as reduction rules that preserve liveness and boundedness properties of Petri nets. The rules are often called the "Murata rules". In fact Murata considers one rule more, a loop addition with a (marked) place, similar to R3. We do not use this rule since it would destroy the soundness property. The rules that we present here are also used by Reijers in [35] and Chrzastowski-Wachtel et al. in [6] to generate workflow nets. Finally, note that the rule R1 can be used to describe the earlier defined notion of *place-expanded net* by saying that if we ignore the labeling this is the net that is obtained by

applying this rule once to all places.

Theorem 18 *Every Jackson net is a sound net.*

It is well-known that the Murata rules preserve liveness and boundedness of Petri nets (see [27]) with respect to a given marking. The marking of the generated net should be derived from the marking of the original net in the following way: for R1 the tokens of p_1 should be distributed over p_2 and p_3 (arbitrarily), for R2 the place p_1 should be empty, for R3 the number of tokens in p_1 and p_2 are the equal, for R4 the tokens of p_1 are duplicated to p_2 and p_3 and for R5 nothing has to be done. In [42] it is shown that soundness is equivalent with liveness and boundedness of the *closure* of a workflow net, i.e. the Petri net obtained from a workflow net by adding one transition $t^\star$ that connects the output place p^0 to the input place p^i, in the initial marking m^0. Since we could not find a complete and formal proof for the preservation properties of the Murata rules, we give a direct soundness proof in Appendix A. In fact, we give a proof of a stronger property called *generalized soundness* [45] which requires that for every natural number k it holds that from every marking reachable from $k \cdot m^i$, i.e., k tokens in p^i, we can reach $k \cdot m^o$, i.e., k tokens in p^o.

Theorem 19 *Not every sound net is a Jackson net.*

Proof. That not every sound net is a Jackson net is shown by the sound net in Figure 8. That it is not a Jackson net can be shown in two ways. The first is by enumerating all Jackson nets with at most 4 places and 4 transitions. This can be done by exhaustively applying the generation rules until we find nets with more than 4 places or more than 4 transitions since all rules either increase the number of places or the number of transitions. It can then be observed that the net in Figure 8 is not in this finite list of nets. Another proof can be given by observing that none of the right-hand sides of the generation rules can be matched in the net, i.e., there is no part of the net that might be the result of the application of one of the generation rules, so it cannot be generated by any of the rules from a smaller net. □

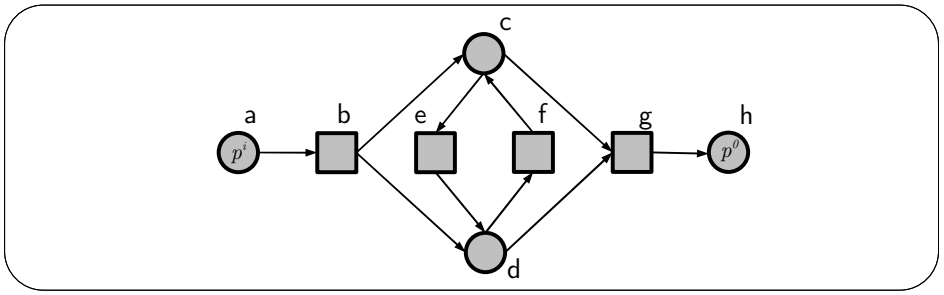


Fig. 8. Not a Jackson net

5 The Jackson Types of Jackson Nets

In the preceding section we introduced the relationship between Jackson types and Jackson nets that defines when a Jackson type generates a Jackson net. Recall that the definition did not make it clear wether this is a many-to-many, one-to-many or one-to-one relationship, which is what is investigated in more detail in this section. The variety in the nets that are generated by a certain Jackson type indicates how well the structure of the net is represented by the type. It is shown in this section (Theorem 20) that this is perfect, i.e., up to isomorphism the Jackson net is completely determined by the type. The relationship can also be used in reverse to determine the data type for the dossiers of the cases of a certain workflow. In that case the variety in Jackson types that all generate the same Jackson net indicates the variety of dossier data types that are generated for a certain workflow, which should ideally be as small as possible. It is shown in this section that although there is some variety this is small and can be characterized by a few simple algebraic identities (Theorem 27). The final part of this section discusses to which extent Jackson nets, the set of which are a proper subset of the set of workflow nets, restrict the ability or make it harder to express certain workflows.

5.1 The variety in Jackson nets generated by a certain Jackson types

In the following theorem we establish to which extent multiple Jackson nets can be generated by the same Jackson type.

Theorem 20 *Two atomic Jackson nets Ω and Ω' are generated by the same Jackson type iff Ω and Ω' are isomorphic.*

Proof. It is easy to see that if Ω and Ω' are isomorphic Jackson nets then they are generated by a common Jackson type since for both nets we can use the same generation up to the choice of the new nodes.

That two Jackson nets are isomorphic if they are generated by the same Jackson type is shown as follows. Consider the syntax tree of the Jackson type τ as defined by the syntax in Definition 1. From this tree we derive the *abstract syntax tree*, denoted as T_τ, as follows: (1) the leaves for brackets are omitted, (2) the J-nodes with an $\mathcal{A}$-child with an atomic-type child is replaced with just the atomic-type node and (3) the J-nodes that have a child labeled with one of ";", "||", "+" or "#" are now themselves labeled with this operator and the child in question is removed. Note that the result is a rooted ordered node-labeled binary tree where leaves are labeled with elements of $\mathcal{A}$ and internal nodes are labeled with one of the operators. It can then be shown wit induction upon the number of steps for the generation of the Jackson net Ω

from the Jackson type τ that there is a one-to-one mapping h between the nodes of Ω and the leaves of T_τ such that (A) it maps leaves to nodes with the same atomic-type label and (B) for two distinct nodes n_1 and n_2 in Ω it holds that there is an edge from n_1 to n_2 iff the simple path in T_τ from $h(n_1)$ to $h(n_2)$ satisfies a certain condition C. For this purpose we define a path in T_τ as a non-empty list of pairs $((n_1, n_1'), \ldots, (n_k, n_k'))$ such that for all $1 \leq i < k$ the unordered pair $\{n_i, n_i'\}$ is an edge in T_τ and $n_i' = n_{i+1}$. Moreover, with each pair (n, n') in such a path we associate a string $\lambda(n, n')$ such that:

- if n is a ;-node and n' is its first (second) child then "α" ("β")
- if n is a $\|$-node and n' is its first (second) child then "γ" ("δ")
- if n is a +-node and n' is its first (second) child then "μ" ("ν")
- if n is a #-nodes and n' is its first (second) child then "φ" ("ψ")
- if $\lambda(n', n) =$ "x" then $\lambda(n, n') =$ "x^{-1}"

The *string of a path* $((n_1, n_1'), \ldots, (n_k, n_k'))$ is then defined as $\lambda(n_1, n_1') \cdot \ldots \cdot \lambda(n_k, n_k')$. The condition C then can be defined as saying that the string of the path must be in the language of the regular expression $(\beta^{-1} + \gamma^{-1} + \delta^{-1} + \mu^{-1} + \nu^{-1} + \varphi^{-1})^*(\alpha^{-1}\beta + \varphi^{-1}\psi + \psi\varphi^{-1})(\alpha + \gamma + \delta + \mu + \nu + \varphi)^*$.

That there exists a one-to-one mapping between the leaves of T_t and the nodes of Ω such (A) and (B) hold can be shown with induction upon the size of T_τ. If this size is 1 then (A) and (B) clearly hold. If the size is larger than one then Ω must be generated in more than one step. Let $\Omega' \Rightarrow \Omega$ be the last step in the generation of Ω and let n be the nodes that were replaced in this step. We can take the nodes in the subtree of T_τ that represent the subexpression of τ that n was labeled with in Ω'. It is clear that if we replace these nodes with a single node v labeled with a special atomic type a then (1) this is the abstract syntax tree of a Jackson type τ^a, (2) this type τ^a generates a Jackson net Ω^a that is equal to Ω' except that the label of n is replaced with a and (3) by the induction hypothesis there is a one-to-one mapping between the nodes in Ω^a and the leaves of T_{τ^a} such that C holds. We can then verify for each generation rule that we can extend this mapping to a one-to-one mapping between the nodes of Ω and the leaves of T_τ such that (A) and (B) hold. Note that for this we need to show that C holds for the paths between new leaves, between new leaves and old leaves, but not between old leaves because in T_τ and T_{τ^a} these are the same and also are the edges between the associated nodes in Ω^a and Ω.

From the above it follows that all the Jackson nets that are generated by τ are isomorphic up to the classification of nodes as places and transitions. However, since this classification is uniquely determined by the graph and the choice of the input and output place it follows that all these Jackson nets are completely isomorphic. $\square$

5.2 The variety in Jackson types from which a certain Jackson net is generated

If the relationship between Jackson types and Jackson nets is used to generate a dossier data type then it is important that the generated type can indeed accommodate all the information that is involved in a run of the workflow, i.e., its trace set should contain exactly the traces of the Jackson net. This is established by the following theorem.

Theorem 21 *If the atomic Jackson net Ω is generated by the Jackson type τ then $Tr(\Omega) = Tr(\tau)$.*

Proof. We introduce the notion of *interpreted trace set* of a workflow net Ω labeled with types, $inTr(\Omega) = \{\alpha_1 \cdot \ldots \cdot \alpha_k \mid x_1 \ldots x_k \in Tr(\Omega), \alpha_1 \in Tr(x_1), \ldots, \alpha_k \in Tr(x_k)\}$. Informally the interpreted trace set defines the sets of traces of a workflow net where we associate with an event associated with a place or transition not simply an atomic type, but an element of the trace set of the type that the place or transition is labeled with. Note that if Ω is an atomic net then $Tr(\Omega) = inTr(\Omega)$. Then it can be shown that when we generate Ω_{i+1} from Ω_i with one of the generation rules for Jackson nets then $inTr(\Omega_{i+1}) = inTr(\Omega_i)$. Since for Ω_0 it will hold that $inTr(\Omega_0) = Tr(\tau)$ and by induction for the generated Ω that $inTr(\Omega) = Tr(\Omega)$ it follows that $Tr(\Omega) = Tr(\tau)$. $\square$

Another important issue is whether the generate dossier data type is unique or not. The following theorem shows that it is not, but that all the different Jackson types generated from a certain Jackson net are algebraically equivalent as defined by the algebraic identities in Figure 9.

Theorem 22 *Two Jackson types τ and τ' generate the same Jackson net iff τ and τ' are algebraically equivalent.*

In order to prove Theorem 22 we first prove a simplified lemma for which we need the following definitions. We first define simple types which can be informally described as types with the operators + and $\|$ replaced with the single operator $\oplus$.

Definition 23 (Simple Type) *The set of* simple types *is defined by the following syntax of J^S:*

$$J^S ::= \mathcal{A} \mid (J^S \ ; \ J^S) \mid (J^S \ \oplus \ J^S) \mid (J^S \ \# \ J^S).$$

As for normal types we can similarly define algebraic equivalence.

Definition 24 (Algebraic Equivalence of Simple Types) *The* algebraic equivalence $\equiv^S_{alg}$ *is the smallest equivalence relation on the set of simple types that fulfils the identities of Figure 9.*

$$
\begin{array}{rcl}
(\tau_0 \ ; \ \tau_1) \ ; \ \tau_2 & \equiv^S_{alg} & \tau_0 \ ; \ (\tau_1 \ ; \ \tau_2) \\
(\tau_0 \ \oplus \ \tau_1) \ \oplus \ \tau_2 & \equiv^S_{alg} & \tau_0 \ \oplus \ (\tau_1 \ \oplus \ \tau_2) \\
\tau_0 \ \oplus \ \tau_1 & \equiv^S_{alg} & \tau_1 \ \oplus \ \tau_0 \\
(\tau_0 \ \# \ \tau_1) \ \# \ \tau_2 & \equiv^S_{alg} & \tau_0 \ \# \ (\tau_1 \ \oplus \ \tau_2)
\end{array}
$$

Fig. 9. Defining Identities for the Algebraic Equivalence for Simple Types

The second notion is *input-output graph* which are very similar to the notion of graph of a net.

Definition 25 (Input-Output Graph) *An* input-output graph *is a tuple* (V, E, I, O) *with* (V, E) *a directed graph and* I *and* O *subsets of* V *which are called* input nodes *and* output nodes*, respectively.*

Finally, just like for Jackson nets we introduce rules that associate simple types with input-output graphs. The rules are given in Figure 10. The rules may be applied to any node in the input-output graph and the after each rule the new input and output sets are the same except that

- after S1 if v_1 was an input node then v_2 is an input nodes,
- after S1 if v_1 was an output node then v_3 is an output node,
- after S2 if v_1 was an input node then v_2 and v_3 are input nodes,
- after S2 if v_1 was an output node then v_2 and v_3 are output nodes,
- after S3 if v_1 was an input node then v_2 is an input nodes, and
- after S3 if v_1 was an output node then v_2 is an output nodes.

The class of input-output graphs that can be generated from a simple type is called *simple Jackson graphs.*

The two following properties can be shown for simple Jackson graphs with induction upon their generation:

- It does not contain loops.
- It has at least one input node and at least one output node.
- For every node it holds that (1) it is either an input node or there is a non-empty path to it to from an input node and (2) it is either an output node or there is a non-empty path from it to an output node.

We now set out to prove the following Lemma.

Lemma 26 *If two simple types* τ *and* τ' *generate the same simple Jackson*

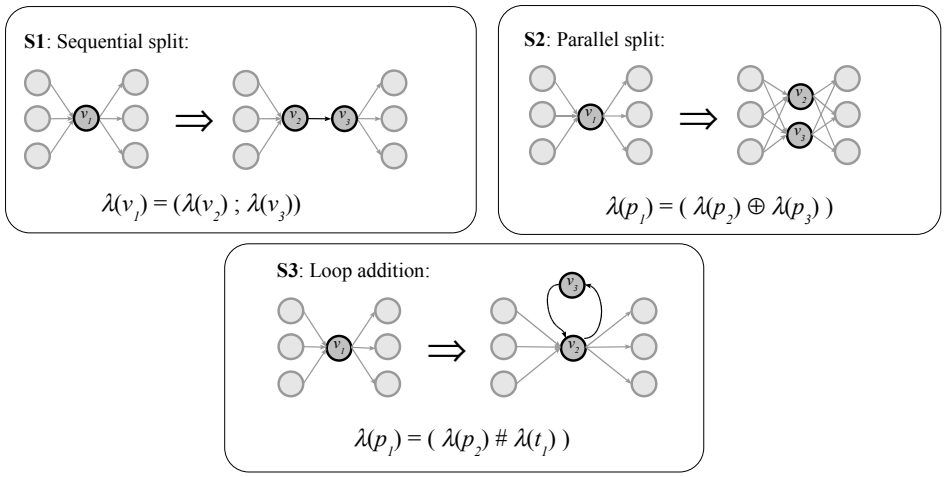


Fig. 10. The generation rules for simple Jackson graphs

graph then τ and τ' are algebraically equivalent.

Proof. The proof proceeds as follows. We consider only so-called *normalized* simple types which means that the algebraic identities are applied as rewrite rules such that (1) all brackets are moved to the right, i.e., we do not allow types of the form $((\tau_1;\tau_2);\tau_3)$, $((\tau_1 \oplus \tau_2) \oplus \tau_3)$ or $((\tau_1 \# \tau_2) \# \tau_3)$ and (2) we assign some kind of Gödel-number $\mathcal{G}(\tau)$ to every simple type τ and allow $(\tau_1 \oplus \tau_2)$ only if τ_2 is of the form $(\tau_3 \oplus \tau_4)$ and $\mathcal{G}(\tau_1) \leq \mathcal{G}(\tau_3)$ or if if τ_2 is *not* of the form $(\tau_3 \oplus \tau_4)$ and $\mathcal{G}(\tau_1) \leq \mathcal{G}(\tau_2)$. Then we show that with each simple Jackson graph there is exactly one such simple type that generates it.

As discussed in the proof of Theorem 20 we can relate subexpressions of a simple type to subgraphs by considering the abstract syntax tree of the type. With this it can be shown that simple Jackson graphs can be decomposed into smaller simple Jackson graphs based on the type they were generated. These decompositions are schematically indicated in Figure 11 where (a) is the decomposition defined by an atomic type, (b) by a sequence type, (c) by a parallel type and (d) by an iteration type. Note that the input nodes and output nodes of the decomposed graph contain I and O, respectively. However, every simple Jackson graph can only be decomposed in one of these ways since with each decomposition certain properties of the graph must hold. For decomposition (a) the graph must contain exactly one node, whereas for all other decompositions there must be more. For decomposition (b) it must hold that from every input node there is a path to every output node, which is not true if decomposition (c) is possible since then there is no path from a node in G_1 to a node in G_2. For decomposition (d) the graph must be strongly connected, which is not the case if (b) or (c) is possible since in both cases there is no path from a node in G_2 to a node in G_1. It follows that only one of the decompositions is possible for a certain simple Jackson graph and hence all the simple types that generate it have the same form, i.e., the root node of

the abstract syntax tree has the same label.

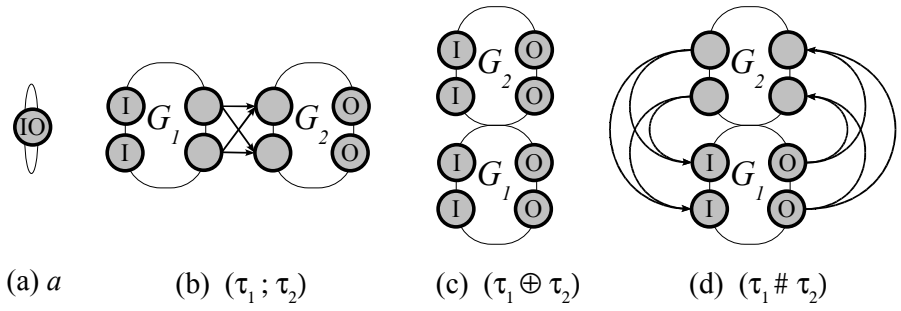
In the following we show with induction on the size of the simple Jackson graph that once we know the type of the root node of the syntax tree and the simple type that generates the simple Jackson graph is a normalized simple type then we can derive (1) what the type of the root node of G_1 is and (2) which part of the input-output graph is G_1 and which part is G_2.

First we consider the case where the root node of the abstract syntax tree indicates a sequence type. Since the type is normalized there are only three possibilities for the left-hand side and the corresponding decompositions are indicated in Figure 12. We can observe that decomposition (b.1) is characterized by a single input node, which is not possible for the other decompositions in the figure. Moreover, in (b.3) there are paths between all input nodes, which is not possible in (b.2). Once we know which decomposition applies we can derive what G_1 (and therefore also G_2) as follows. For (b.1) G_1 consists of the single input node. For (b.2) G_1 consists of all the nodes that are reachable from at least one of the input nodes but not from all of them. For (b.3) G_1 consists of all the nodes that can be reached from an input node and from which we can reach an input node.

Next we consider the case where the root node of the abstract syntax tree indicates an iteration type. Because the type is normalized we have also here only three possibilities for the left-hand side and the corresponding decompositions are indicated in Figure 13. We can observe that decomposition (d.1) is characterized by a single input node which is also an output node, which is not possible for (d.2) since there input nodes cannot be output nodes and also not for (d.3) since there we have at least two input nodes. Moreover, if we define *internal paths* as paths that, except for the first and last node, only go through nodes that are not input or output nodes, then in (d.2) there is between every input node and output node an internal path, whereas in (d.3) this is not possible. Also here we can derive what G_1 (and therefore also G_2) is since it consists in all cases of the input nodes and all those nodes that can be reached from them with internal paths.

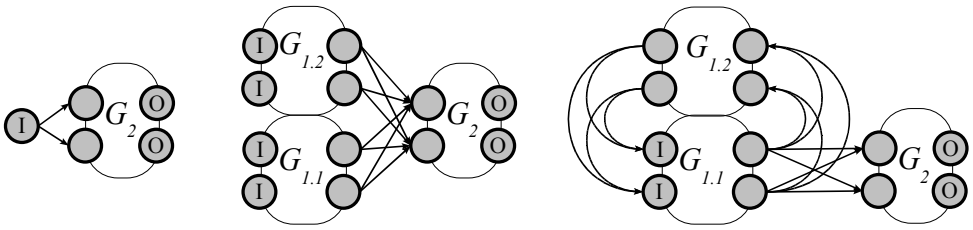
Finally we consider the case where the root node of the abstract syntax tree indicates a parallel type. If we assume that the type that generates the simple graph is $(\tau_1 \oplus (\tau_2 \oplus \ldots \tau_k \ldots))$ with all τ_i not parallel types, then the k corresponding components can be found by taking the finest partition of the nodes such that two nodes connected by an edge are in the same set. By induction we may assume that there is a unique normalized simple type for each component that generates that component, and the component with simple normalized type with the smallest Gödel number has to be G_1.

This concludes the cases to be considered, so it is in all cases uniquely determined how the simple Jackson graph has to be divided into component simple Jackson graphs, and by induction we may assume that for these components there is only one unique normalized simple type that generates them, and hence also only one that generates the complete simple Jackson graph. □



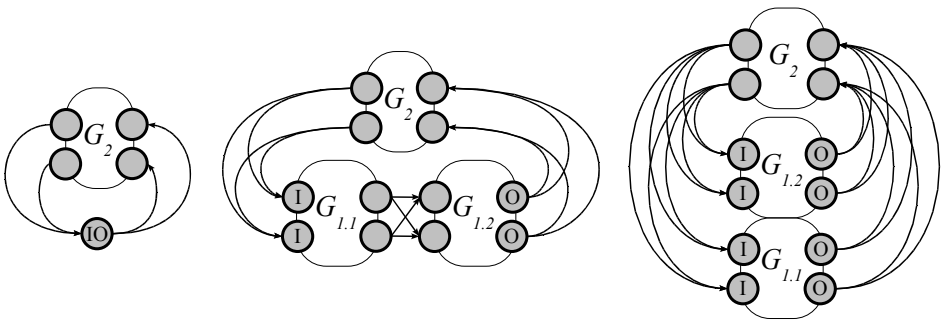
(a) a (b) $(\tau_1 ; \tau_2)$ (c) $(\tau_1 \oplus \tau_2)$ (d) $(\tau_1 \# \tau_2)$

Fig. 11. Decompositions of simple Jackson graphs based on their generating type



(b.1) $(a ; \tau_1)$ (b.2) $((\tau_{1.1} \oplus \tau_{1.2}) ; \tau_2)$ (b.3) $((\tau_{1.1} \# \tau_{1.2}) ; \tau_2)$

Fig. 12. Decompositions based on sequence types



(d.1) $(a \# \tau_2)$ (d.2) $((\tau_{1.1} ; \tau_{1.2}) \# \tau_2)$ (d.3) $((\tau_{1.1} \oplus \tau_{1.2}) \# \tau_2)$

Fig. 13. Decompositions based on iteration types

Using Lemma 26 we can now prove Theorem 22.

Proof. We first show that two Jackson types τ and τ' generated the same Jackson net if τ and τ' are algebraically equivalent. As was shown in the proof of Theorem 20 the graph of the place-expanded net is determined by the abstract syntax tree of the type such that for every leaf there is a node in

the graph and there is an edge between two such nodes if the path between these nodes define a string in a certain regular language. It can then be shown that if an algebraic identity is applied to a syntax tree the string associated with two leaves is in that language iff it was before the identity was applied. It follows that the associated graph of the net stays the same if we apply an identity, and hence the whole Jackson net remains the same.

Next we show that if two Jackson types τ and τ' generate the same Jackson net then τ and τ' are algebraically equivalent. Assume that with an atomic Jackson net Ω we associate two Jackson types τ and τ' and that these are not algebraically equivalent. With the Jackson type we can associate the simple types σ and σ' that are obtained by replacing $||$ and $+$ with $\oplus$. It can be shown that two Jackson types are algebraically equivalent iff the corresponding simple types are algebraically equivalent. It also holds that the graph of a Jackson net that is generated by a Jackson type is identical to the input-output graph that is generated by the simple type that is generated by the Jackson type. So it follows that σ and σ' are not algebraically equivalent and hence that the graphs of the Jackson nets generated by τ and τ' are not isomorphic. But this contradicts the assumption that τ and τ' generate the same Jackson net, so the assumption that they are not algebraically equivalent must be false. □

Summarizing, we can now characterize the ambiguity in the relationship between Jackson types and Jackson nets with the following corollary.

Theorem 27 *If the Jackson nets Ω_1 and Ω_2 are generated by the Jackson types τ_1 and τ_2, respectively, then the following are equivalent:*

(1) Ω_1 *and* Ω_2 *are isomorphic*
(2) $\tau_1 \equiv_{alg} \tau_2$

Proof. Clearly (1) $\Rightarrow$ (2) because if Ω_1 and Ω_2 are isomorphic then τ_2 also generates Ω_1 and so by Theorem 22 it follows that $\tau_1 \equiv_{alg} \tau_2$. It also holds that (2) $\Rightarrow$ (1) because if $\tau_1 \equiv_{alg} \tau_2$ then by Theorem 22 there is a Jackson net Ω_3 generated by both τ_1 and τ_2. Since both Ω_1 and Ω_3 are generated by τ_1, and both Ω_2 and Ω_3 are generated by τ_2 it follows by Theorem 20 that Ω_1 and Ω_3 are isomorphic, and Ω_2 and Ω_3 are isomorphic. Hence Ω_1 and Ω_2 are also isomorphic. □

5.3 Characterizing the expressive power of Jackson Nets

The set of Jackson nets is a proper subset of the set of workflow nets, which raises the question whether the class of workflows that they can express is not too limited. One way of comparing the expressive power of such formalisms is by looking at the sets of traces that can be expressed. These are in both

cases the same, viz., if both places and transitions are labeled then both formalisms can express exactly all sets of trances that can be described by Jackson types and if only places are labeled then both can express all regular languages. There is however a difference if we restrict ourselves to nets where each place and transition has a unique label. In that case the Jackson nets can only express trace sets that can be described by a Jackson type in which every atomic type appears at most once. Consider for example the net in Figure 8 which is not a Jackson net. Its trace set is described by the Jackson type $(\mathsf{a};\mathsf{b};\mathsf{g};\mathsf{h}) + (\mathsf{a};\mathsf{b};(((\mathsf{c};\mathsf{e};\mathsf{d})\#\mathsf{f}) \parallel ((\mathsf{d};\mathsf{f};\mathsf{c})\#\mathsf{e}));\mathsf{g};\mathsf{h}) + (\mathsf{a};\mathsf{b};((\mathsf{d}\#(\mathsf{f};\mathsf{c};\mathsf{e})) \parallel (\mathsf{c}\#(\mathsf{e};\mathsf{d};\mathsf{f})));\mathsf{g};\mathsf{h})$. It can be verified that there is indeed no equivalent Jackson type where all the atomic types appear at most once. As is shown by Theorem 28 this is a characteristic property of trace sets that can be expressed by Jackson nets without duplicate labels, i.e., such Jackson nets can express exactly all trace sets that can be described by Jackson types in which every atomic type appears at most once. Moreover, as is shown in Corollary 32, this Jackson net is completely determined by the trace set, i.e., given a certain trace set there is at most one Jackson net without duplicate labels that represents this trace set. In the same corollary it is shown that it follows that for types in which atomic types appear at most once algebraic equivalence coincides with trace equivalence

Theorem 28 *Let Ω be an atomic sound net without duplicate labels. Ω is a Jackson net iff there is an Jackson type τ in which every atomic type appears at most once and it holds that $Tr(\tau) = Tr(\Omega)$.*

Before we prove this theorem we first prove the following lemmas.

Lemma 29 *Let the atomic Jackson net Ω be generated by the Jackson type τ. All labels of Ω are different iff τ contains no duplicate labels*

Proof. Let us define the number of occurrences of an atomic type a in a labeled Petri net as the sum of the number of times a appears in the label of each of the nodes of the net. It can be easily verified for each generation step $\Omega_i \Rightarrow \Omega_{i+1}$ that an atomic type a occurs once in Ω_i iff a occurs once in Ω_{i+1}. By induction it follows that for any generation sequence $\Omega_0 \Rightarrow \ldots \Rightarrow \Omega_k = \Omega$ the same holds for Ω_0 and Ω_k. If this generation sequence associates τ with Ω then, since Ω_0 consists of a single node labeled with τ, it holds that a occurs once in τ iff it does so in Ω_0 and, as was already shown, the latter is true iff a occurs once in $\Omega_k = \Omega$. $\square$

Note that the fact that for each generation step $\Omega_i \Rightarrow \Omega_{i+1}$ an atomic type a occurs once in Ω_i iff a occurs once in Ω_{i+1}, would not be true if we would use the Kleene-star in our types instead of the # that we use now.

Lemma 30 *If τ is a Jackson type without duplicate labels, Ω is a sound workflow net and $Tr(\tau) = Tr(\Omega)$ then Ω is safe, i.e., in all markings m that are*

reachable from the initial marking m^i it holds that $m(p) \leq 1$ for all places p in Ω.

Proof. It can be shown with induction on the structure of τ that $Tr(\tau)$ does not contain a trace of the form $xaay$ where x and y are strings of atomic types and a is an atomic type:

- If $\tau = B$ with B an atomic type then this clearly holds.
- If $\tau = (\tau_1; \tau_2)$ then we know by induction that aa does not appear in $Tr(\tau_1)$ or $Tr(\tau_2)$. So if there is a trace of the form $xaay$ in $Tr((\tau_1; \tau_2))$ then $Tr(\tau_1)$ contains a trace of the form xa and $Tr(\tau_2)$ contains a trace of the form ay. However, since every atomic type appears only once in τ this is not possible.
- If $\tau = (\tau_1 \parallel \tau_2)$ then we know by induction that aa does not appear in $Tr(\tau_1)$ or $Tr(\tau_2)$. So if there is a trace of the form $xaay$ in $Tr((\tau_1 \parallel \tau_2))$ then $Tr(\tau_1)$ contains a trace with a and $Tr(\tau_2)$ contains a trace with a. However, since every atomic type appears only once in τ this is not possible.
- If $\tau = (\tau_1 + \tau_2)$ then we know by induction that aa does not appear in $Tr(\tau_1)$ or $Tr(\tau_2)$. Since $Tr((\tau_1 + \tau_2)) = Tr(\tau_1) \cup Tr(\tau_2)$ it follows that aa also not appears in $Tr((\tau_1 + \tau_2))$.
- If $\tau = (\tau_1 \# \tau_2)$ then we know by induction that aa does not appear in $Tr(\tau_1)$ or $Tr(\tau_2)$. So if there is a trace of the form $xaay$ in $Tr((\tau_1 \# \tau_2))$ then, because the empty string is not in the trace set of any type, there must be traces of the form $x'a$ and ay' in $Tr(\tau_1)$ and $Tr(\tau_2)$, respectively, or vice versa. However since every atomic type appears only once in τ it holds that a cannot appear in both τ_1 and τ_2 and therefore this is not possible.

However, it can also be shown that if Ω is not safe then there is a trace of the form $xaay$ in its trace set. Assume that Ω' is the place-expanded net of Ω With every marking m of Ω we associate an associated marking m of Ω' such that whenever place p is split into input place p'_1 and output place p'_2 then $m'(p_1) = m(p)$ and $m'(p_2) = 0$. Clearly it holds that if a marking m for Ω is reachable from the initial marking of Ω then m' is reachable from the initial marking of Ω'. Moreover, from the fact that Ω is sound as proven in Theorem 18, it can be derived that from every marking reachable from m' we can reach the final marking of Ω'. Since Ω is not safe there must be a marking m_2 that is reachable from the initial marking m^i of Ω and a place p in Ω such that $m_2(p) > 1$. Then it holds for the associated marking m'_2 that $m'_2(p_1) > 1$. If the place p is labeled with a in Ω then it follows that the transition with label a in Ω' can fire at least twice in a row after which we can still reach the final marking of Ω'. So there will be a trace of the form $xaay$ in the trace set of Ω. $\square$

Lemma 31 *Let Ω_1 and Ω_2 be two sound safe atomic workflow nets without duplicate labels. If $Tr(\Omega_1) = Tr(\Omega_2)$ then Ω_1 and Ω_2 are isomorphic.*

Proof. In the following we will describe markings of a place-expanded net Ω' in terms of markings of the original net Ω where if a place p is split into begin place p_1, transition t and end place p_2 then the tokens in p_1 and p_2 under the associated marking m' are described as *inactive* and *active* tokens, respectively, in p under the marking m.

If Ω is a sound safe atomic workflow net and a and b two labels in Ω then we say that a *enables* b iff there is in $Tr(\Omega)$ a trace of the form $xaby$ but not of the form xby'. We will show that in the graph of Ω it holds for two nodes with labels a and b that there is an edge between these two nodes iff a enables b.

First, it can be observed that if a enables b then either a is a label of a place and b of a transition or vice versa. This is shown as follows. Assume that a and b are both labels of places and there is a trace of the form $xaby$. Then after trace x there will be an inactive token in the places for a and b. So there will also be a trace of the form $xbay$, which contradicts the assumption that a enables b. Assume that a and b are both labels of transitions and there is a trace of the form $xaby$. Then after trace x there are active tokens that enable the transition a and active tokens that enable transition b, since the firing of a produces only inactive tokens which cannot enable b. So there will also be a trace of the form $xbay$, which contradicts the assumption that a enables b.

We now consider the two remaining cases: a is the label of place and b is the label of a transition and vice versa.

Assume that a is the label of a place p and b the label of a transition t. We consider the two directions:

if: Assume there is no edge from p to t and there is a trace of the form $xaby$. Then b is also already enabled after x and from the soundness of Ω it follows that there is a trace of the form xby'.

only if: Assume there is an edge from p to t. From the soundness of Ω it follows that there is a trace of the form $xaby$. Since Ω is safe it holds that after the trace x there is one inactive token in p and therefore transition t is not enabled, hence there is no trace of the form xby'.

Assume that a is the label of a transition t and b the label of a place p. We consider the two directions:

if: Assume there is no edge from t to p and there is a trace of the form $xaby$. Then b contains already an inactive after x and from the soundness of Ω it follows that there is a trace of the form xby'.

only if: Assume there is an edge from t to p. From the soundness of Ω it follows that there is a trace of the form $xaby$. Since Ω is safe it holds that after the trace x there is no token in p that can be activated, hence there is no trace of the form xby'.

From the above it follows that the graph of Ω is exactly defined by the trace set. Moreover, since the first and the last label in every trace must be the label of the input and output place, respectively, it follows that it is determined which label belongs to a transition and which label belongs to place. Consequently the net is fully determined by the trace set. $\square$

We now proceed with the proof of Theorem 28:

Proof. The only-if part of the theorem follows from the definition of Jackson net, Lemma 29 and Theorem 21.

By definition of Jackson net and Theorem 21 it holds that we can derive from the Jackson type τ a Jackson net Ω' such that $Tr(\Omega') = Tr(\tau)$. We know that Ω' is sound and safe by Theorem 18 and Lemma 30. Since $Tr(\Omega') = Tr(\tau) = Tr(\Omega)$ it follows by Lemma 31 that Ω and Ω' are isomorphic. Since Ω' is a Jackson net, it follows that Ω is also a Jackson net. $\square$

With this result we can now extend Theorem 27 as follows.

Corollary 32 *If the Jackson nets Ω_1 and Ω_2 have no duplicate labels and are generated by the Jackson types τ_1 and τ_2, respectively, then the following are equivalent:*

(1) Ω_1 *and* Ω_2 *are isomorphic*
(2) $\tau_1 \equiv_{alg} \tau_2$
(3) $Tr(\Omega_1) = Tr(\Omega_2)$
(4) $Tr(\tau_1) = Tr(\tau_2)$

Proof. That (1) $\Leftrightarrow$ (2) was already established in the proof of Theorem 27. That (3) $\Leftrightarrow$ (4) follows from Theorem 21. That (2) $\Rightarrow$ (3) follows from Theorem 7. That (3) $\Rightarrow$ (1) follows from Lemma 31 and Lemma 30. $\square$

Observe that this corollary implies that for Jackson types in which every atomic type appears at most once it holds that algebraic equivalence is the same as trace equivalence, and hence, for these types we can indeed axiomatize algebraically trace equivalence. This is in contrast with the set of all types, which cannot be axiomatized in that way, as was shown in Theorem 4.

6 Case Study

Recall the model of section 2.3 in particular Figure 1. The places have an atomic type denoted by a single character label, while the transitions represent tasks or activities in the process and they labeled with a number and a name.

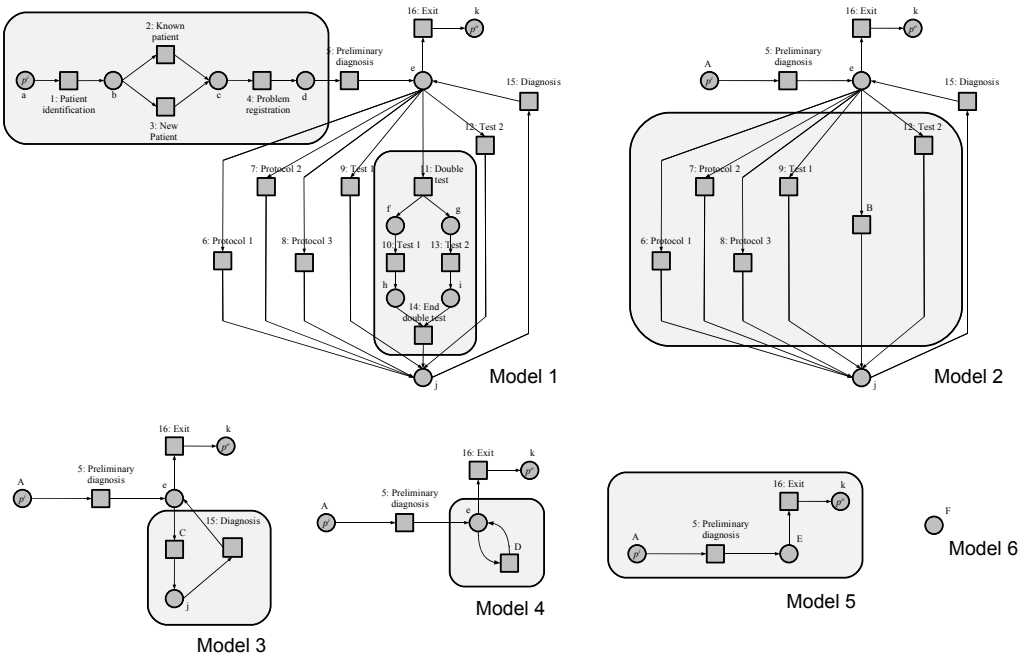


Fig. 14. Generation of the singleton net

To derive the document type for the cases handled by this process, we start with a reduction process according to the Murata rules R1,...,R5. In Figure 14 we show the reduction process. Note that several steps are aggregated into single steps. We start with the original model, model 1. We cluster one box into a place with label A using rules R5 and R1 twice. The other box is reduced to a transition with label B using rules R1 and R4 twice. This gives model 2. The definitions of these labels are: $A = (a; 1; b; (2 + 3); c; 4; d)$ and $B = (11; ((f; 10; h)\|(g; 13; i)); 14)$. Note that the transitions are represented by their numbers only. In model 2 we reduce everything in the box, using rule R5 five times. This leads to model 3 where one transition with label C represents the cluster. Label C is defined by $C = (6\|7\|8\|9\|B\|12)$. Note that we use here the associativity (algebraic equivalence rules) of the parallel composition constructor $\|$. This brings us to model 4. Here we apply rule R2 to obtain label D defined by $D = (C; j; 15)$. Next we reduce the loop with rule R3, leaving us a place with label E where $E = (e\#D)$. So we obtain model 5. The last step is the application of rule R1 twice. This leads us to model 6, a singleton net with label F, where $F = (A; 5; E; 16; k)$. Hence we have verified that we have indeed a Jackson net and we can derive a tree structure by expanding type F. This is in fact the Jackson type (see Figure 15). This tree structure is the basis for the document type for the case data, which can be considered as an *electronic patient record*.

Note that in this tree the non-leaf nodes are labeled with a constructor (sequential, parallel, choice or loop constructors) and that the leaf nodes represent places or transitions in the Jackson net. To make a useful document type of

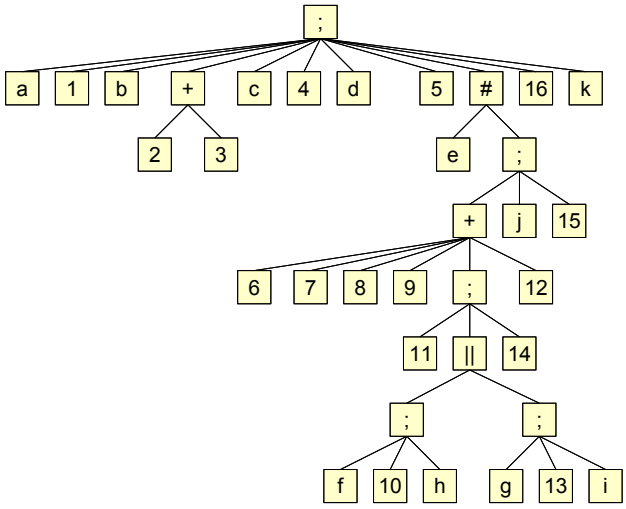


Fig. 15. Jackson type as tree

this we have to modify the tree in two ways: we have to delete the leaves that refer to a place in the net and we have to add data elements to the transition leaves. The reason is that in our approach the places are only used as the "glue" between the transitions and that the tokens in the places only carry references to the case document. The transitions represent the tasks and in each task new data may be created that should be recorded in the case document. The reduced tree is displayed in Figure 16. Note that the loop has only one child node and that some sequences are reduced to a single node. Since tasks 11 and 14 are in fact only control flow tasks, we can also omit them which makes the sequence constructor between tasks 9 an 12 superfluous. The sequence constructors containing tasks 10 and 13 can also be omitted since they both have only one remaining child.

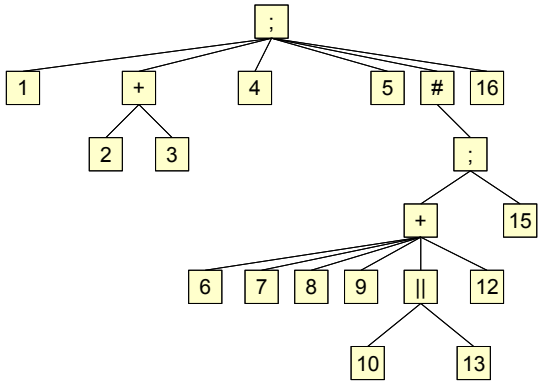


Fig. 16. Reduced tree

Next we add *data elements* to the task nodes. In principle they can belong to any kind of data type, however we choose a record type here. The augmented tree is displayed in Figure 17.

In task 1 we identify a patient by an identity card and we create a new case,

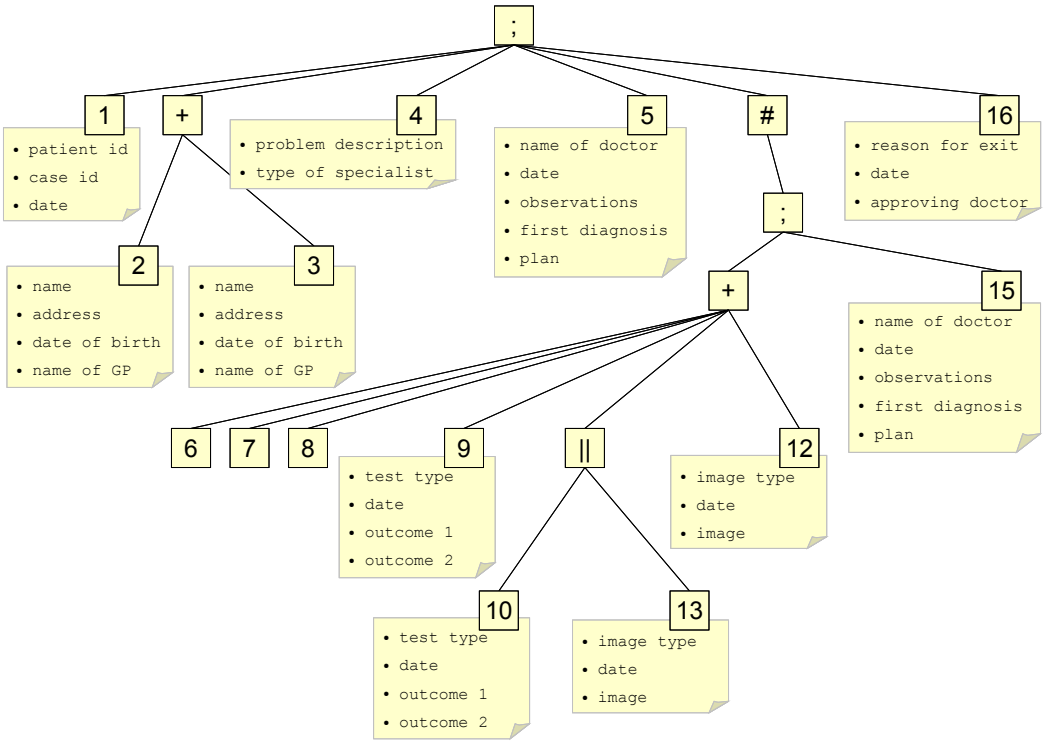


Fig. 17. Case tree

with a case identity. If the patient is known then its patient identity number is registered otherwise a new patient identity number is created. In task 3: "New patient", the relevant patient data, such as name, address, day of birth and identity of the general physician are entered to the document. In task 2: "Known patient" the same data elements are retrieved from earlier cases and updated if necessary. In task 4: "Problem registration" the problem is described based on an interview with the patient or a letter of the general physician. Also the type of medical specialist for the preliminary diagnosis is selected. In task 5: "Preliminary diagnosis" a doctor describes the observations of the physical examination, the first diagnosis and the plan for next steps. In tasks 9 and 10 we perform Test 1 and so they have the same record type. We assume that Test 1 is a laboratory test and that Test 2 is image generation. For task 15: "Diagnosis" we have the same structure as for task 5. For task 16: "Exit" we register the reason for the exit, and the doctor who approved it. For tasks 6, 7 and 8 we have not detailed these records. This could be a record but also a subtree for the processes belonging to the protocols.

Now we have for each new case of a patient a new document. It is often preferable to have one patient document that contains all the cases of a patient. This is easy to construct out of the data structure we have made so far by making a new tree structure with a leave node "Basic patient data" sequentially coupled to the case tree in Figure 18. We also created one root node on top of this with a loop construct, to collect all patient documents into one document. Finally

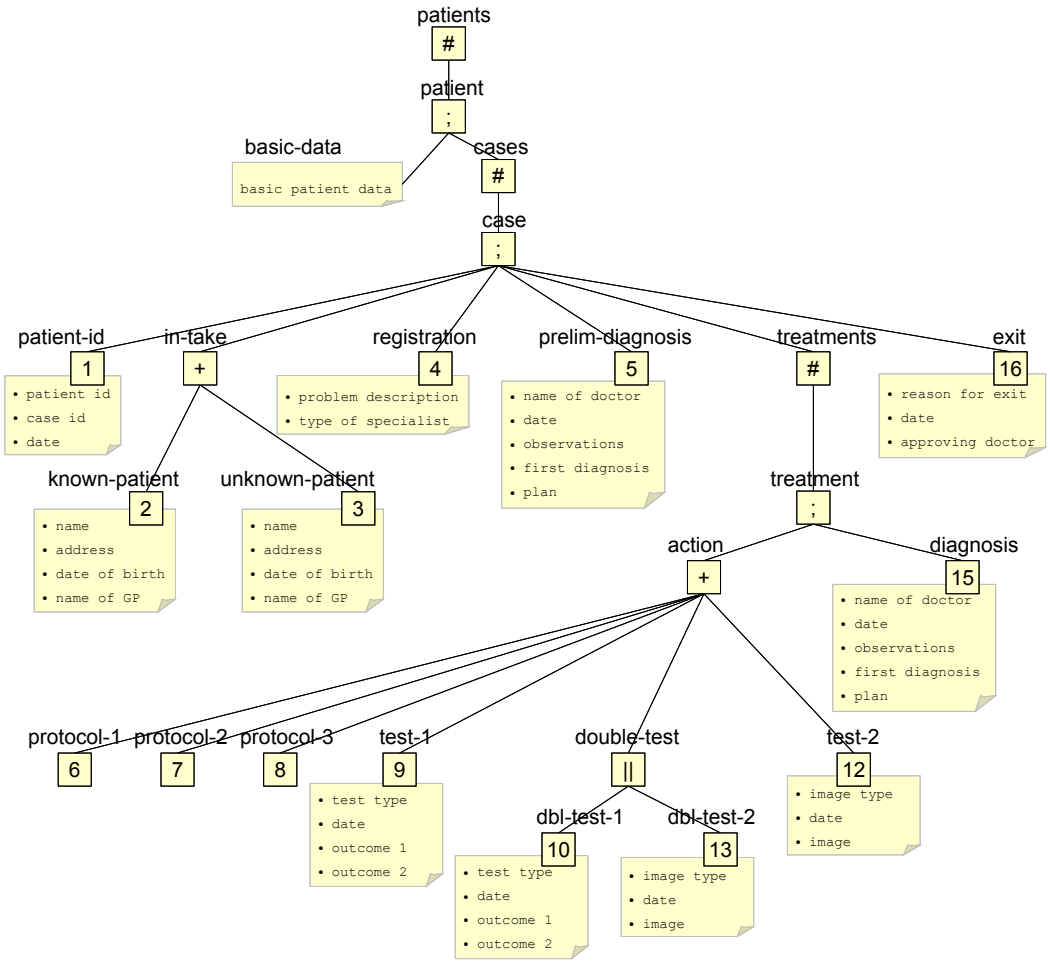


Fig. 18. Patients tree

we added a unique label to each node in the tree. From this tree structure we can easily derive a DTD (Document Type Definition) for XML documents. The DTD of the case tree of Figure 17 is given in Figure 19. Note that the fields of records such as in `case` are optional. This allows the representation of runs of cases that have not already completed. Also note, however, that the DTD does not capture that the fields need to be added in a certain order, e.g., it allows the addition of the field `registration` even if the field `intake` has not been defined.

From this document type we can formulate queries on the trees in XPath or XQuery. So we can ask questions concerning the business process. For example we may ask:

- How many patients have more than two unfinished cases?

```
fn:count(/patients/patient[fn:count(cases/case[not(exit)]) > 2])
```

- In how many cases with a heart problem, two tests were taken in parallel (tasks

```
<!DOCTYPE patients [
  <!ELEMENT patients      ( patient* )>
  <!ELEMENT patient       ( basic-data, cases )>
  ....
  <!ELEMENT cases         ( case* )>
  <!ELEMENT case          ( patient-id, intake?, registration?,
                            prelim-diagnosis?, treatments?, exit? )>
  <!ELEMENT in-take       ( known-patient | unknown-patient )>
  ....
  <!ELEMENT treatments    ( treatment* )>
  <!ELEMENT treatment     ( action, diagnosis? )>
  <!ELEMENT action        ( protocol-1 | double-test | protocol-2 |
                            protocol-3 | test-1 | test-2 )>
  ....
  <!ELEMENT double-test   ( test-1, test-2? )>
  ....
]>
```

Fig. 19. A partial DTD for the Patients tree

10 and 13)?

```
fn:count(/patients/patient/cases/case[
           fn:contains(registration/problem, "heart") and
           treatments/treatment/action/double-test ])
```

- What is the percentage of cases where the patient is new?

```
fn:count(/patients/patient/case[in-take/unknown-patient]) * 100
```

So far we have seen how we could derive a data model from a process model. The opposite is also possible. We will not show this in detail, but in principle it is possible to traverse the derivation we have made here in opposite direction. It means that we first have to look for the subtree that describes the cases instead of the whole patient population. Then we have to strip the data elements. Next we have to add leave nodes for places and sometimes a level with a constructor in order to obtain a Jackson type. Deriving a Jackson net from a Jackson type can be done automatically.

7 Possibilities and limitations of Jackson types

In the preceding sections it is shown that there is a close relationship between certain process specifications, viz., Jackson nets, and certain document types, viz., Jackson types. In this section we briefly discuss in more detail how and

when we think this relationship can be useful in practice.

7.1 Generating dossier data types from process specifications

The most straightforward use of the relationship is the generation of a data type for dossiers that contain all the information that is involved in a certain run of a workflow, as is demonstrated in the preceding section. Such a data structure is especially useful for case-based information systems where, as it is often required for audit reasons, data is only added to the database and never removed. We claim that the generated data type organizes the data in a way that closely corresponds to the structure of the workflow, and therefore makes it easy to formulate queries over workflow runs that involve both the control flow and the data flow, as illustrated in Section 6.

The generated data structure is a conceptual view that contains and organizes all the data associated with a run. By a conceptual view we mean a data model that describes the data at the conceptual level and therefore is not necessarily directly related to how it is stored at the physical level, i.e., at the level of the file system or at the physical level in a DBMS. There are many ways to map a hierarchical data-structure to a file system or a relational DBMS, and if it is stored in a DBMS that supports hierarchical data-structures then this DBMS will often have several ways of mapping the data structure to the physical level. Also note that the Jackson type describes only a high-level view in the sense that it treats the types associated with the places and transitions as atomic. For a full description of the dossier data structure these need to be specified as well.

The generated data structure is not the only or even necessarily the best possible conceptual view for all purposes. For example, for a data-centric application where the control flow is of less concern, a view that organizes the data by class (e.g., for the use case presented in the previous section by the classes Patient, Diagnosis, Protocol and Test) might be more compact and appropriate. However, as was shown in related research [14] it is possible to transparently integrate such different views.

The generation of the dossier data type requires that the process specification is a Jackson net, and the class of Jackson nets is a proper subclass of the class of sound, safe, choice-free workflow nets. This raises the question whether this class is big enough to be of practical interest.

An important point of reference is the industrial standard BPEL, a process specification language. In BPEL workflows are essentially specified as hierarchically nested activities; at the lowest level we find the *basic activities* and at each level activities of lower levels can be combined into *sequence activi-*

ties, *flow activities*, *switch activities*, *pick activities* and *while activities*. Here a sequence activity is an activity that consists of a sequence of activities that are executed in the specified sequence. A flow activity consists of a bag of activities that are executed in parallel. A switch activity chooses from a set of activities one activity to execute on the basis of certain specified conditions. A pick activity does the same but makes the choice based on external events or time outs. Finally, a while activity repeats a certain activity while a certain specified condition holds.

As is shown in [34], the basic behavior of these constructs can be described with Jackson nets. In fact, these constructs correspond roughly to the constructors of Jackson types: sequence activities correspond to the sequence constructor (;), flow activities correspond to the parallelism constructor ($\|$), switch and pick activities correspond to the choice constructor (+), and the while activity corresponds to the loop constructor (#). However, the basic behavior of these constructs can be altered by additional BPEL constructs such as *control links* and *scopes*.

Control links allow the specification of additional control flow dependencies between activities, which is hard to describe with Jackson types. However, such links have no information associated with them, so a Jackson type that is generated while ignoring these links can still accommodate all the information involved in a workflow run. Another problematic construct in BPEL is the *scope* construct which allows the association of several types of handlers with a certain activity which is called the *primary activity*. There are several types of handlers, such as *event handlers*, *fault handlers* and *compensation handlers*. With some exceptions, such as event handlers for events that happen at most one time, these are hard to represent with Jackson types. For example, describing event handlers for events that can happen more than once probably requires an extra type constructor that describes the parallel execution of an arbitrary number of instances of a sub-process. Finally we wish to mention the notion of *exit activity* which, when executed, stops all running activities in the scope. Also this behavior is currently hard to describe with Jackson types, but as a data type Jackson types are able to accommodate the information involved in such an aborted run.

Given the above considerations we can summarize the applicability of Jackson types for BPEL specifications as follows. For specifications that only use the basic behavior of structured activities and control links, the Jackson types seem appropriate and straightforwardly lead to suitable dossier data types. For specifications that use handlers and special termination behavior this approach seems less appropriate, although future research might lead to extensions of Jackson types that can deal with some of these BPEL features.

Another use of the relationship between Jackson nets and Jackson types is the design of a workflow specification based on a data type that represents all information and artifacts that have to be generated by the workflow. This process can start with a normal data type consisting of recursively nested tuple, union and list types. In a next step this type can then be extended with some annotation to indicate which Jackson type corresponds to each nested type. For example, a record type is marked as either a sequence constructor or a parallelism constructor. As already discussed, the resulting process specification can be readily mapped to industry standard specifications such as BPEL.

Generation of the process specification from a data type is only interesting if the structure of the workflow process is largely determined by the structure of the product of the workflow and if the control flow that has to be added is minimal. It can be that the control flow that has to be added is more complex than Jackson types can describe, in which case more powerful formalisms such as data types with instance-dependent access rules [5] can be more appropriate. It can also be that the structure of the workflow is mostly determined by other factors such as the data model that describes involved artifacts and data, and subprocesses or life cycles that are associated with the described entities in this data model. For these cases there are alternative approaches which are discussed in more detail in the next section on related work.

8 Related work

The problem that we addressed in this paper is the integration of the data and the process views, and the possible generation of one from the other. In the existing literature, we can find many different approaches to achieve this integration and to generate workflow specifications from data models or other models. In the following we will attempt to give an overview of these and compare them to our approach.

The first type of approach is the one where a process is specified based on a hierarchical data or product structure which describes the product of the workflow. An example of this is the Product-Based Workflow Design approach (PBWD) which was presented by Reijers, Limam and van der Aalst in [36] and [41]. It aims at determining the workflow process in product manufacturing and generates a process specification from a produce specification in the form of a Bill-of-Material (BOM) [33]. For an example see Appendix B.1. This work can be considered an early predecessor of the work in this paper, and the main

differences are that it only considers generation in one direction and does not consider list types and iteration.

Another type of approach is where the process is specified based on a fixed set of objects with each a certain object life cycle (OLC), which are essentially finite state machines, and certain types of synchronization between these life cycles. An example of this is the *Team Automata* formalism [2], which was introduced by Ellis to describe coordination, collaboration and cooperation in groupware. It describes processes in terms of finite automata where transitions describe both internal and external events, and these automata are synchronized on external events. A similar approach based on life cycles are *Interaction Expressions and Graphs* [13], introduced by Heinlein, where several operators are defined to compose and synchronize life cycles in different ways. Finally, a similar recent approach for generating a process specification is introduced in [22] where Küster, Ryndina and Gall propose a technique for generating a business process description from a given set of objects and their reference OLCs. This is illustrated by an example in Appendix B.2. These approaches differ from the one presented in this paper in that they do not start from a product structure, although the initial set of objects can be interpreted as such, and they do not establish a link between the data that is used in the process and the process specification. The interaction expressions and graphs approach, however, is very similar in that some of its composition operations directly correspond to the Jackson types, but it does not analyze formally the relationship between the expressions and workflow graphs.

Another type of OLC-based approach is the one where next to the objects and their OLCs also a global workflow is specified in which these objects flow. An example of this are *Object/Behavior Diagrams* as introduced by Kappel and Schrefl in [21] and extended for business processes and workflows in [4]. A specification consists of an *object diagram* that defines the data model, *life cycle diagrams* that define the life cycles of the objects, and a *workflow diagram* that describes the global workflow in terms of stores and workflow steps. The stores contain objects that are in certain states and satisfy certain conditions, and the workflow steps move objects between these stores. The workflow diagram is linked to the life cycle diagrams by *workflow realization diagrams* that specify for a particular step in the workflow in more detail when which transitions in the OLCs of the involved objects happen. The involved objects are in this case those that stream into and out of the workflow step. A similar approach based on *business artifacts* is the *Operational Specification* approach (OpS) presented by Nigam and Caswell in [28]. Also here, objects, called *key artifacts*, and their OLCs are specified in terms of business tasks and repositories. In addition the collection of the OLCs of all the artifacts and the interaction between them, can be aggregated into a model that specifies the operational model for the whole business. Although in these approaches there is usually a data model that describes the structure of the objects, there is no

notion of a global data structure that contains all the information concerning a certain case.

Another OLC-based approach where the process specification also depends on a data model and its instance, is described in [26] by Müller, Reichert and Herbst. They propose a framework where a workflow is specified by a data structure and a description of the behavior of the elements in this structure. The data structure is defined by an ER-like schema with classes and binary relationships, and a particular instance of that schema. The behavior is specified by associating with each class an OLC that describes the behavior of objects in this class, and by indicating with relationships in the schema OLC dependencies, i.e., synchronization links between states in the OLCs of the classes that are connected by the relationship. For an example see Appendix B.3. This approach in many ways generalizes the PBWD approach, and is appropriate for data-driven workflows, i.e., workflows where the process structure is largely determined by the structure of the data model. It is especially useful if this data model, i.e., the data schema, is fairly stable and the data structure, i.e., the instance of the schema, often changes, because then only the new instance needs to be specified. It is similar to our work in that it allows the specification of processes based on a data structure, and in that sense it is even more general because it can deal with a more general class of data structures and more complex types of coordination, but it largely ignores the data aspect of the involved objects, i.e., their attribute values etc., and establishes the relationship between data model and process structure only in one direction. An important feature of this work is that it allows multiple instantiations of a subprocess based on the number of objects in a certain class [10]. This cannot be represented by Jackson types except by simply repeating the type for each instantiation. For example, if a process produces a car with four wheels which can be produced in parallel, then the parallel production of the four wheels can be described by a type $(\tau_w \parallel \tau_w \parallel \tau_w \parallel \tau_w)$ where τ_w describes the production of a single wheel. In the described approach it is also possible to have intermediate synchronization between the different instantiations, which is harder and sometimes even impossible to specify with Jackson types.

In some approaches the emphasis is not on object and their life cycles, but more on processes and activities which may be combined into networks by connecting them through data flow and control flow edges, and nesting such networks as subprocesses in other processes. An example of such an approach is *STATEMATE* which is presented by Harel et al in [12] and was designed for the integrated process modeling of reactive systems. It proposes a notation and semantics for three types of related diagrams: *module charts*, *activity charts* and *state charts* that roughly describe the physical structure of the system, the decomposition the system into activities, and the coordination of these activities. For an illustrative example see Appendix B.4. The process specification consists roughly of a hierarchy of activities and sub-activities, and the

behavior of an activity can be described by a state chart. An approach similar to STATEMATE is the *Object Process Methodology* (OPM) which was introduced by Dori in [7]. It features a full-blown object-oriented data model and has a more elaborate graphical notation with a corresponding textual counterpart. In these approaches, the link between the involved data and the process is established by assigning data types to data flows between the activities and data stores within the activities. There is no further relationship between data structure and process structure, but in related later work such as [11] by Harel, Kugler and Pnueli, it is shown how such process specifications can themselves be generated from, and checked against, scenario-based requirements.

Many approaches for adding data flow to process specifications are some variant of high-level Petri nets. Examples of these models are the case-handling workflows [37], the NR/T-nets [30] and the XML Nets [24]. In these models, places are interpreted as containers for data (relations in NR/T nets, XML documents in XML Nets). The flow of data is defined by occurrences of transitions which manipulate (create, change, delete) data of their adjacent places. The firing of a transition may depend on conditions over the consumed tokens that appear as labels of the adjacent edges. For illustration, an example of XML Nets is given in Appendix B.5. These approaches can all be used in conjunction with our techniques, if their types of Petri nets are restricted to Jackson nets.

An approach that emphasizes a more integrated modeling of dataflow in processes is PPM (Process and Product Modeling) [23] by Lee, Eastman and Sacks. This provides a formal framework, called Requirements Collection and Modeling (RCM), that allows the specification of the data flows in a process, their integration and the validation of their consistency. A process is defined as an *act of processing data items* [9]. A process receives, generates, updates, deletes or distributes a data item. The model consistency is based on the interaction and interdependence of the processes with regard to the availability or unavailability of data. Data items are categorized into input and output data and subcategorized in five types. The input data is subdivided into remaining data (data not transferred) and the rest of the input data. The output data comprises the data modified, the data passed-through (not modified but transferred) and the data generated (newly generated in the process). Rules that define the relationship between these data types are used for checking the model consistency. Examples of such rules are: *"output is the union of passed-through, remaining and modified data"* or *"intersection of input and generated data is an empty set"*.

Another approach comes from the domain of software engineering. AHEAD [19] by Jäger, Schleicher and Westfechtel, offers tools for supporting the management and execution of (software) development process. In this approach tasks and data flow are modeled by graph-based specifications. Control flow

relationships define the order of subtask enactment. Data flow relationships connect input and output parameters of tasks and allow data exchange between them. With respect to dynamic task nets, AHEAD distinguishes between process definitions and process instances. A process instance represents a specific development process being executed while a process definition describes a whole class of processes. AHEAD supports the generation of process instances (i.e., program code). This work is similar to ours in that it proposes a unified specification of data and processes. It differs from our work as it does provide a generation from model to instance but not from one model to another one.

Finally we wish to mention the work on workflow data patterns [38] by Russell, ter Hofstede and van der Aalst, where a list of important use cases and patterns are given of usage and representation of data in workflow management systems and workflow specification languages. Since the purpose of dossier data structures is to record all data usage it is important to determine which patterns can be covered by the Jackson type approach. Recall that the approach assumes that all data usage is captured by the types associated with the places and transitions. Also recall that Jackson nets do not allow places that model data stores that are both read and written by a transition. So the patterns (1) *task data* and (8) *data interaction between tasks* are straightforwardly covered by the place and transition types, but the other visibility patterns such as (2) *block data*, (3) *scope data*, etc., can only be dealt with by including the interaction with this data into the type associated with the transition. For the internal data interaction patterns that deal with decompositions, patterns (9) and (10), can be dealt with by generating the type for the flattened net. The patterns for multiple instance tasks, (11) and (12), cannot be dealt with directly because Jackson nets do not support multiple instances of tasks. Finally, pattern (13), *data interaction between different cases*, is not covered directly since Jackson types only describe data used within a case, but also here a simulation is possible by including the interaction with the data in the type of the transition in question. The considerations for the remaining patterns are similar and straightforward.

9 Conclusion

In this paper we have presented a framework in which it is possible to establish a straightforward relationship between the process model and the data model of a case-based information system such as a workflow system. To this end we introduced a special class of Petri nets, called Jackson nets, to model the business processes, and a document type, called Jackson types, to represent the data model. We have shown that there is a one-to-one correspondence between Jackson nets and Jackson types with several interesting theoretical

properties. Finally, we have illustrated the use of the framework by an example in which it is shown that the resulting data model allows the straightforward formulation of queries over runs of the system.

In future research we intend to extend the presented framework to address the problem of constructing and integrating the different data models that are associated with each event in a run of the system. Some preliminary work on this was presented in [14] but no attempt was made yet to integrate it into the framework that is presented here. In addition we will develop tools to support the design of workflows based on Jackson types.

Acknowledgment: The authors would like to thank the anonymous referees whose remarks have helped considerably to improve the quality and readability of this paper.

References

[1] Luca Aceto, Willem Jan Fokkink, and Anna Ingólfsdóttir. On a question of A. Salomaa: The equational theory of regular expressions over a singleton alphabet is not finitely based. *Theoretical Computer Science*, 209(1–2):141–162, December 1998.

[2] Maurice H. Ter Beek, Clarence A. Ellis, Jetty Kleijn, and Grzegorz Rozenberg. Synchronizations in team automata for groupware systems. *Comput. Supported Coop. Work*, 12(1):21–69, 2003.

[3] Gèrard Berthelot. Checking properties of nets using transformation. In *Advances in Petri Nets 1985, covers the 6th European Workshop on Applications and Theory in Petri Nets-selected papers*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40, London, UK, 1986. Springer-Verlag.

[4] Peter Bichler, Günter Preuner, and Michael Schrefl. Workflow transparency. In *CAiSE '97: Proceedings of the 9th International Conference on Advanced Information Systems Engineering*, pages 423–436, London, UK, 1997. Springer-Verlag.

[5] Toon Calders, Stijn Dekeyser, Jan Hidders, and Jan Paredaens. Analyzing workflows implied by instance-dependent access rules. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 100–109, New York, NY, USA, 2006. ACM.

[6] Piotr Chrzastowski-Wachtel, Boualem Benatallah, Rachid Hamadi, Milton O'Dell, and Adi Susanto. A top-down petri net-based approach for dynamic workflow modeling. In Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 336–353. Springer, 2003.

[7] Dov Dori. Object-process methodology as a business-process modelling tool. In *Proceedings of the 8th European Conference on Information Systems, Trends in Information and Communication Systems for the 21st Century, ECIS 2000*, Vienna, Austria, July 3–5 2000.

[8] Marlon Dumas, Wil M. van der Aalst, and Arthur H. ter Hofstede. *Process-aware information systems: bridging people and software through process technology.* John Wiley & Sons, Inc., New York, NY, USA, 2005.

[9] Charles M. Eastman. Managing integrity in design information flows. *Computer-Aided Design*, 28(6-7):551–565, 1996.

[10] Adnene Guabtni and François Charoy. Multiple instantiation in a dynamic workflow environment. In Anne Persson and Janis Stirna, editors, *CAiSE*, volume 3084 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[11] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In Hans-Jörg Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Formal Methods in Software and Systems Modeling*, volume 3393 of *Lecture Notes in Computer Science*, pages 309–324. Springer, 2005.

[12] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. Software Eng.*, 16(4):403–414, 1990.

[13] Christian Heinlein. Workflow and process synchronization with interaction expressions and graphs. In *Proceedings of the 17th International Conference on Data Engineering*, pages 243–252, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Jan Hidders, Jan Paredaens, Philippe Thiran, Geert-Jan Houben, and Kees van Hee. Non-destructive integration of form-based views. In Johann Eder, Hele-Mai Haav, Ahto Kalja, and Jaan Penjam, editors, *9th East European Conference on Advances in Databases and Information Systems (ADBIS 2005)*, number 3631 in Lecture Notes in Computer Science, pages 74–86. Springer, September 2005.

[15] Healthcare Information and Management Systems Society (HIMSS). Electronic health records: A global perspective. `http://www.himss.org/content/files/DrArnold20011207EISPresentationWhitePaper.pdf`, January 2007.

[16] Michael A. Jackson. *Principles of Program Design.* Academic Press, 1975.

[17] Michael A. Jackson. *System Development.* Prentice Hall Publishing, 1983.

[18] Michael A. Jackson. *Software requirements & specifications: a lexicon of practice, principles and prejudices.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[19] Dirk Jäger, Ansgar Schleicher, and Bernhard Westfechtel. Ahead: A graph-based system for modeling and managing development processes. In Manfred Nagl, Andy Schürr, and Manfred Münch, editors, *AGTIVE*, volume 1779 of *Lecture Notes in Computer Science*, pages 325–339. Springer, 1999.

[20] Kurt Jensen. Coloured petri nets: a high level language for system design and analysis. In G. Rozenberg, editor, *APN 90: Proceedings on Advances in Petri nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[21] Gerti Kappel and Michael Schrefl. Object/behavior diagrams. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 530–539, Washington, DC, USA, 1991. IEEE Computer Society.

[22] Jochen Malte Küster, Ksenia Ryndina, and Harald Gall. Generation of business process models for object life cycle compliance. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 165–181. Springer, 2007.

[23] Ghang Lee, Charles M. Eastman, and Rafael Sacks. Eliciting information for product modeling using process modeling. *Data Knowl. Eng.*, 62(2):292–307, 2007.

[24] Kirsten Lenz and Andreas Oberweis. Inter-organizational business process management with XML nets. In Hartmut Ehrig, Wolfgang Reisig, Grzegorz Rozenberg, and Herbert Weber, editors, *Petri Net Technology for Communication-Based Systems*, volume 2472 of *Lecture Notes in Computer Science*, pages 243–263. Springer, 2003.

[25] Alain J. Mayer and Larry J. Stockmeyer. The complexity of word problems – this time with interleaving. *Inf. Comput.*, 115(2):293–311, 1994.

[26] Dominic Müller, Manfred Reichert, and Joachim Herbst. Data-driven modeling and coordination of large process structures. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4803 of *Lecture Notes in Computer Science*, pages 131–149. Springer, 2007.

[27] Tadao Murata. Petri nets, properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[28] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Syst. J.*, 42(3):428–445, 2003.

[29] OASIS. Web services business process execution language version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, April 2007.

[30] A. Oberweis. An integrated approach for the specification of processes and related complex structured objects in business applications. *Decision Support Systems*, 17:31–53, 1996.

[31] Object Management Group. Unified modeling language specification, version 1.4.2. `http://www.omg.org/docs/formal/05-04-01.pdf`, January 2005.

[32] Object Management Group. Business process modeling notation, v1.1. `http://www.omg.org/spec/BPMN/1.1/PDF/`, January 2008.

[33] A. Orlicky. Structuring the bill of materials for mrp. *Production and Inventory Management*, pages 19–42, 1972.

[34] Chun Ouyang, Eric Verbeek, Wil M. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, July 2007.

[35] Hajo A. Reijers. *Design and Control of Workflow Processes: Business Process Management for the Service Industry*. Number 2617 in Lecture Notes in Computer Science. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[36] Hajo A. Reijers, Selma Limam, and Wil M. P. van der Aalst. Product-based workflow design. *Journal of Management Information Systems*, 20(1):229–262, 2003.

[37] Hajo A. Reijers, J. H. M. Rigter, and Wil M. P. van der Aalst. The case handling case. *Int. J. Cooperative Inf. Syst.*, 12(3):365–391, 2003.

[38] Nick Russell, Arthur ter Hofstede, David Edmond, and Wil van der Aalst. Workflow data patterns. Technical report, Queensland University of Technology, Brisbane, 2004.

[39] K. T. Sridhar and C. A. R. Hoare. JSD expressed in CSP. In Malcolm P. Atkinson, Peter Buneman, and Ronald Morrison, editors, *Data Types and Persistence, Informal Proceedings of the First Workshop on Persistent Objects*, pages 49–82, August 1985.

[40] W. M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[41] W. M. P. van der Aalst. On the automatic generation of workflow processes based on product structures. *Comput. Ind.*, 39(2):97–111, 1999.

[42] Wil M. P. van der Aalst. Verification of workflow nets. In *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426, London, UK, 1997. Springer-Verlag.

[43] Wil M. P. van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[44] Kees M. van Hee, Natalia Sidorova, and Marc Voorhoeve. Soundness and separability of workflow nets in the stepwise refinement approach. In Wil M. P. van der Aalst and Eike Best, editors, *ICATPN*, volume 2679 of *Lecture Notes in Computer Science*, pages 337–356. Springer, 2003.

[45] Kees M. van Hee, Natalia Sidorova, and Marc Voorhoeve. Generalised soundness of workflow nets is decidable. In Jordi Cortadella and Wolfgang Reisig, editors, *ICATPN*, volume 3099 of *Lecture Notes in Computer Science*, pages 197–215. Springer, 2004.

A Proof of Soundness and Generalized Soundness of Jackson Nets

In this appendix we prove that all Jackson nets are sound and in addition satisfy the property of generalized soundness. We start with defining the latter property more formally.

Definition 33 (Generalized Sound Net) *A net $\Omega = (P, T, F, p^i, p^o, \lambda)$ is said to be* generalized sound *if it holds in the reachability graph of Ω for every natural number k that from every marking reachable from $k \cdot m^i$, we can reach $k \cdot m^o$.*

Recall that the property of soundness is defined in Definition 14 is defined such that a net $\Omega = (P, T, F, p^i, p^o, \lambda)$ is said to be *sound* if it holds in the reachability graph of Ω that

(1) from every marking reachable from m^i, we can reach m^o and
(2) for every transition $t \in T$ there is a run with an edge (m_i, t, m_j).

It will be clear that the property of generalized soundness is a generalization of the first requirement for soundness. It was introduced by van Hee, Sidorova and Voorhoeve in [44] and is motivated by both practical and theoretical considerations. The practical argument is that this type of soundness is relevant for batch processing systems where multiple cases are processed at the same time and where generalized soundness guarantees that in spite of possible interference between the different cases the system will always terminate with all cases correctly processed. The theoretical argument is that conventional soundness is not compositional with respect to refinement. For example, it is not the case that if in a sound net we replace a place with another sound net that then the result is necessarily also sound. However, this does hold for generalized soundness and this allows for the verification of soundness in a compositional way.

Since general soundness implies the second property of soundness we will prove Theorem 18 by proving the following stronger theorem.

Theorem 34 *Every Jackson net is a sound net and a generalized sound net.*

Proof. It easy to show by induction upon the number of generation steps for generating the Jackson net that it is a labeled Petri net. In the same fashion it can be shown that it satisfies the properties of a workflow net using the fact that rules R3 and R4 cannot be applied to the input place p^i and output place p^o. Finally, also with induction on the number of steps, we show that each generated workflow net is both sound and generalized sound. For this it is sufficient to show that in the reachability graph of the net it holds that (1) from every marking reachable from the initial marking $k \cdot m^i$ we can reach

the final marking $k \cdot m^o$ and (2) for every transition there is a run from m^i to m^o in which t is fired. We consider for this each of the generation rules and assume that $\tilde{\Omega}$ was constructed from Ω by applying that rule:

R1 Consider the two soundness properties:

- Assume that there is a transition path r_1 from $k \cdot m^i$ to a marking m' for the net $\tilde{\Omega}$. We define a mapping of transition paths of $\tilde{\Omega}$ to transition paths of Ω as follows. First, we define a mapping of markings m of $\tilde{\Omega}$ to markings $\hat{m}$ of Ω such that $\hat{m}$ is equal to m except that $\hat{m}(p_1) = m(p_2) + m(p_3)$. Then we define the mapping of transition paths r of $\tilde{\Omega}$ to a transition path $\hat{r}$ of Ω by mapping each edge (m'_i, t, m'_j) to the edge $(\hat{m}'_i, t, \hat{m}'_j)$ if $t \neq t_1$, and removing the edge if $t = t_1$. It is easy to verify that the resulting list of edges is indeed a transition path of Ω. It then follows that there is a transition path $\hat{r}_1$ from $k \cdot m^i$ to the marking $\hat{m}'$ for the net Ω. By induction it then holds that there is for Ω a transition path r_2 from $\hat{m}'$ to $k \cdot m^o$.

  We now define a reverse mapping from transition paths of $\tilde{\Omega}$ to transition paths of Ω as follows. First, we define a mapping of markings of Ω to markings of $\tilde{\Omega}$ such that m is mapped to $\overline{m}$ where $\overline{m}$ is equal to m except that $\overline{m}(p_2) = 0$ and $\overline{m}(p_3) = m(p_1)$. Then we define the mapping of a transition path r of Ω to a transition path $\overline{r}$ of $\tilde{\Omega}$ by mapping each edge (m_i, t, m_j) to $(\overline{m}_i, t, \overline{m}_j)$ if $t \notin {\bullet p_1}$ and to the two consecutive edges $(\overline{m}_i, t, m'_i)$ and $(m'_i, t_1, \overline{m}_j)$ where $m'_i = \overline{m}_i - \overline{\bullet} t + t\overline{\bullet}$ otherwise, where $\overline{\bullet} t$ and $t\overline{\bullet}$ denote ${\bullet t}$ and $t{\bullet}$ in $\tilde{\Omega}$. Note that in the latter case it indeed holds that in m'_i transition t_1 is enabled in $\tilde{\Omega}$ and $\overline{m}_j = m'_i - \overline{\bullet} t_1 + t_1\overline{\bullet}$ in $\tilde{\Omega}$. In m'_i transition t_1 is enabled because $t \in {\bullet p_1} = \overline{\bullet} p_2$. It follows that $\overline{r}$ is indeed a transition path of $\tilde{\Omega}$.

  With this mapping we can then show that there is a transition path $\overline{r_2}$ from $\overline{\hat{m}'}$ to $\overline{k \cdot m^o}$ for $\tilde{\Omega}$. Since there is also a transition path of $\tilde{\Omega}$ from m' to $\overline{\hat{m}'}$ which consists of $m'(p_2)$ times firing t_1, there is a transition graph path for $\tilde{\Omega}$ from m' to $\overline{k \cdot m^o}$ and in addition it holds that for $\tilde{\Omega}$ that $\overline{k \cdot m^o} = k \cdot m^o$.
- Consider a transition t in $\tilde{\Omega}$. Either t is a transition in Ω or $t = t_1$. We first consider the case where t is a transition in Ω. By induction we know there is run for Ω from m^i to a marking m such that t is enabled in m. As was shown in the previous point it then holds that for $\tilde{\Omega}$ there is also a run from m^i to $\overline{m}$ and by the way that $\tilde{\Omega}$ is constructed it holds that t is enabled in $\overline{m}$. Next we consider the case where $t = t_1$. Either p_2 is the input place of $\tilde{\Omega}$, in which case t is enabled in m^i, or it is not and then there is at least one transition $t' \in {\bullet p_2}$ and, as shown in the previous case, a marking m' that is reachable from m^i and in which t' is enabled. It follows that after firing t' we obtain a marking that is reachable from m^i and in which t is enabled.

R2 Consider the two soundness properties:

- Assume that there is a transition path r_1 from $k \cdot m^i$ to a marking m' for the

net $\tilde{\Omega}$. We define a mapping of transition paths of $\tilde{\Omega}$ to transition paths of Ω as follows. First, we define a mapping of markings m of $\tilde{\Omega}$ to markings $\hat{m}$ of Ω such that $\hat{m}$ is equal to m except that $\hat{m}(p) = m(p)+m(p_1)$ if $p \in t_1\bullet$. Then we define the mapping of transition paths r of $\tilde{\Omega}$ to a transition path $\hat{r}$ of Ω by mapping each edge (m'_i, t, m'_j) to the edge $(\hat{m}'_i, t, \hat{m}'_j)$ if $t \neq t_2$ and $t \neq t_3$, to $(\hat{m}'_i, t, \hat{m}'_j)$ if $t = t_2$ and removing the edge if $t = t_3$. It is easy to verify that the resulting list of edges is indeed a transition path of Ω. It then follows that there is a transition path $\hat{r}_1$ from $k \cdot m^i$ to the marking $\hat{m}'$ for the net Ω. By induction it then holds that there is for Ω a transition path r_2 from $\hat{m}'$ to $k \cdot m^o$.

We now define a reverse mapping from transition paths of $\tilde{\Omega}$ to transition paths of Ω as follows. First, we define a mapping of markings of Ω to markings of $\tilde{\Omega}$ such that m is mapped to $\hat{m}$ where $\hat{m}$ is equal to m except that $\hat{m}(p) = m(p) + m(p_1)$ if $p \in t_1\bullet$. Then we define the mapping of a transition path r of Ω to a transition path $\overline{r}$ of $\tilde{\Omega}$ by mapping each edge (m'_i, t, m'_j) to the edge $(\hat{m}'_i, t, \hat{m}'_j)$ if $t \neq t_2$ and $t \neq t_3$, to $(\hat{m}'_i, t, \hat{m}'_j)$ if $t = t_2$ and removing the edge if $t = t_3$. It can be verified that $\overline{r}$ is indeed a transition path of $\tilde{\Omega}$.

With the latter mapping we can then show that there is a transition path $\overline{r_2}$ from $\overline{\hat{m}'}$ to $\overline{k \cdot m^o}$ for $\tilde{\Omega}$. Since there is also a transition path from m' to $\overline{\hat{m}'}$ which consists of $m'(p_1)$ times firing t_3, there is a transition path from m' to $\overline{k \cdot m^o}$ and in addition it holds that for $\tilde{\Omega}$ that $\overline{k \cdot m^o} = k \cdot m^o$.

- Consider a transition t in $\tilde{\Omega}$. Either $t \notin \{t_2, t_3\}$, $t = t_2$ or $t = t_3$. We first consider the case where $t \notin \{t_2, t_3\}$. By induction we know there is run for Ω from m^i to a marking m such that t is enabled in m. As was shown in the previous point it then holds that for $\tilde{\Omega}$ there is also a run from m^i to $\overline{m}$ and by the way that $\tilde{\Omega}$ is constructed it holds that t is enabled in $\overline{m}$. Next we consider the case where $t = t_2$. By induction we know there is run for Ω from m^i to a marking m such that t_1 is enabled in m. As was shown in the previous point it then holds that for $\tilde{\Omega}$ there is also a run from m^i to $\overline{m}$ and since $\bullet t_2$ in $\tilde{\Omega}$ is equal to $\bullet t_1$ in Ω it holds that t_2 is enabled in $\overline{m}$. The case for $t = t_3$ is similar except after t_2 is enabled we fire it once such that t_3 becomes enabled.

R3 Consider the two soundness properties:

- Assume that marking m' is reachable from $k \cdot m^i$ for the net $\tilde{\Omega}$. We can then construct a transition path for Ω that ends in the marking m' by omitting all edges for transition t_1. By induction we know that there is for Ω a transition path from m' to $k \cdot m^o$. Clearly this path is also a transition path from m' to $k \cdot m^o$ for $\tilde{\Omega}$.
- Consider a transition t in $\tilde{\Omega}$. Either $t \neq t_1$ or $t = t_1$. We first consider the case where $t \neq t_1$. By induction we know there is run for Ω from m^i to a marking m such that t is enabled in m. The same path will be a run for $\tilde{\Omega}$ and in its final marking t will be enabled. Next we consider the case where $t = t_1$. If p_2 is the start place of $\tilde{\Omega}$ the t will be enabled after an empty run.

If p_s is not the start place of $\tilde{\Omega}$ then there is a transition $t' \in \bullet p_2$ and as was shown in the previous case there is a run after which t' is enabled. If we extend this run by firing t' we end in a marking in which t_1 is enabled.

R4 Consider the two soundness properties:

- Assume that marking m' is reachable from $k \cdot m^i$ for the net $\tilde{\Omega}$. We define a mapping of markings of $\tilde{\Omega}$ to markings of Ω such that m is mapped to $\hat{m}$ where $\hat{m}$ is equal to m except that $\hat{m}(p_2) = \hat{m}(p_3) = m(p_1)$. Observe that a transition t is enabled in m for $\tilde{\Omega}$ iff it is enabled in $\hat{m}$ for Ω. Also observe that if for a marking m_1 transition t is enabled for $\tilde{\Omega}$ and after the firing of t in $\tilde{\Omega}$ we arrive in marking m_2 then if we fire t for a marking $\hat{m}_1$ in Ω then we arrive in marking $\hat{m}_2$. We can then construct a transition path for Ω that ends in the marking m' by firing the same transitions. By induction we know that there is for Ω a transition path from m' to $k \cdot m^o$. Clearly the path that fires the same transitions is also a run from m' to m^o for $\tilde{\Omega}$.
- Consider a transition t in $\tilde{\Omega}$. By induction we know there is run for Ω from m^i to a marking m such that t is enabled in m. As was shown in the previous point then there is a similar path for $\tilde{\Omega}$ that fires the same transitions in the same order and ends in $\hat{m}$ in which t is enabled.

R5 Consider the two soundness properties:

- Assume that marking m' is reachable from $k \cdot m^i$ for the net $\tilde{\Omega}$. We can then construct a run for Ω that ends in the marking m' by replacing all edges for transitions t_2 and t_3 with edges for t_1. Note that this will still be a run since $\bullet t_1 = \bullet t_2 = \bullet t_3$ and $t_1\bullet = t_2\bullet = t_3\bullet$. By induction we know that there is for Ω a transition path from m' to $k \cdot m^o$. We obtain a transition path from m' to $k \cdot m^o$ for $\tilde{\Omega}$ by replacing all edges for t_1 with edges for t_2.
- Consider a transition t in $\tilde{\Omega}$. Either $t \in \{t_2, t_3\}$ or t is a transition in Ω and not t_1. We first consider the case where $t \in \{t_2, t_3\}$. By induction we know there is run for Ω from m^i to a marking m such that t_1 is enabled in m. As shown in the previous point we obtain a run from m^i to m for $\tilde{\Omega}$ by replacing edges for t_1 with edges for t_2, and in m the transitions t_2 and t_3 are both enabled in $\tilde{\Omega}$. Next we consider the case where t is a transition in Ω that is not equal to t_1. By induction we know there is run for Ω from m^i to a marking m such that t is enabled in m. As shown in the previous point we obtain a run from m^i to m for $\tilde{\Omega}$ by replacing edges for t_1 with edges for t_2, and in m the transition t is enabled in $\tilde{\Omega}$.

Summarizing we have shown that if $\tilde{\Omega}$ is generated from Ω by one of the rules R1, R2, R3, R4 and R5, and Ω is sound and generalized sound then $\tilde{\Omega}$ is also sound and generalized sound. Since singleton nets are clearly sound and generalized sound it the follows by induction upon the number of generation steps that these properties hold for all Jackson nets. $\square$

B Examples of Workflow Modeling Approaches

In this section we illustrate some of the process modeling approaches that are mentioned in Section 8 on related work.

B.1 Product-Based Workflow Design

The Product-Based Workflow Design approach (PBWD) was introduced by Reijers, Limam and van der Aalst in [36] and [41]. The approach aims at determining the workflow process in product manufacturing. The workflow process is obtained from the structure and characteristics of a product. The Bill-of-Material (BOM) [33] is used for capturing the structure of the products to be produced. The BOM is a tree-like structure with the end product as root and raw materials and purchased products as leaves. The edges are used to specify composition relation. They can have a cardinality to indicate the number of products needed. In PBWD, the classical BOM is extended with options and choices. This extension allows specifying sequencing, parallelism and choice. A workflow is generated as a Petri net from a BOM and the resulting Petri net is proved to be sound.

An example of PBWD is given in Figure B.1. In this example, the BOM consists of an end product (P_1) and 5 components. A black dot indicates that the sub-product is a mandatory component while a circle indicates that a choice is made between several components (P_5 or P_6). The resulting workflow of the given BOM is given in the same figure. The end product P_1 corresponds to a net responsible for the production of P_1 and the components needed to produce it. The net starts with a transition, $prep(P_1)$. This transition triggers the activities needed to produce P_1. Transition $prep(P_1)$ starts the production of P_2, P_3 (optional) and P_4. The possibility to refrain from P_3 is modeled by the by-pass via transition $skip(P_3)$. For the product P_4, the choice between P_5 and P_6 is modeled by the place $in(P_5, P_6)$. The actual production of a P_i is modeled by transition, $prod(P_i)$.

B.2 Integrating Object Life Cycles

In [22] Küster, Ryndina and Gall propose a technique for generating a business process from a set of given reference object life cycles (OLCs) for the involved objects. The generation technique also requires the specification of synchronization between transitions in the OLCs. The composition of the OLC is computed and used to generate a process model in which dependencies between input and output object states are preserved. The result of the genera-

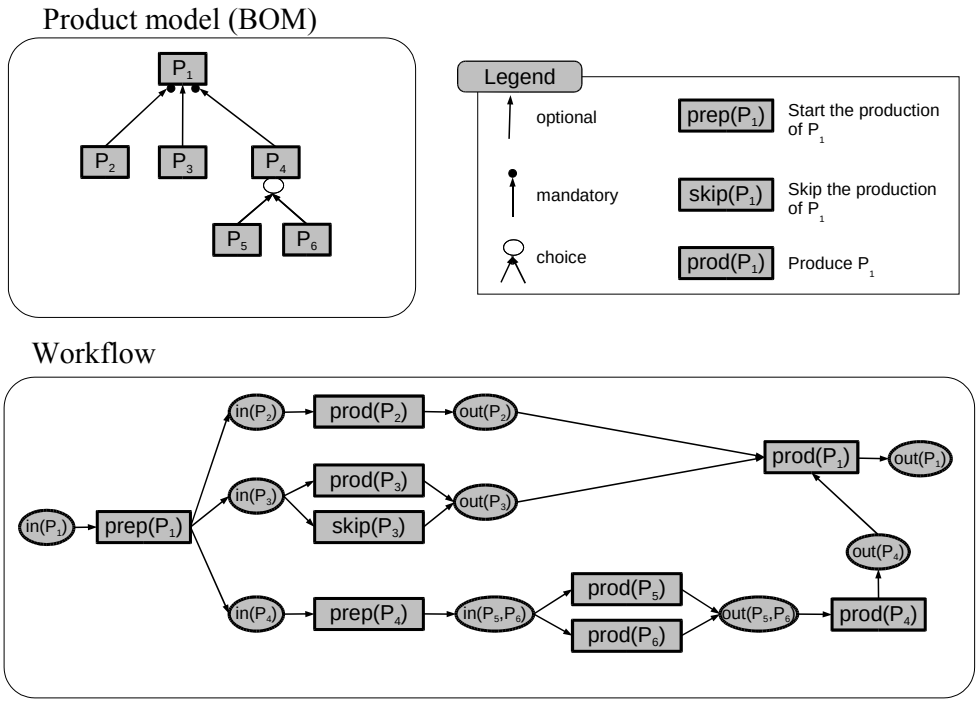


Fig. B.1. Generation of a workflow net from a given BOM

tion is proved to be compliant with the reference OLCs through the notions of *conformance* and *coverage* which informally state that within the complete business process the individual objects can do no more and no less than the transitions specified in the reference OLCs.

In Figure B.2 we give an example illustrating the technique for two OLCs with one synchronization point between transitions `b` and `e`. From these OLCs, the product automation is determined, but it is formulated in terms of transitions. These transitions are defined by a transition from one of the OLCs and a set of *input pins* and *output pins* that indicate for each of the objects in which state it is before the transition and after the transition, respectively. If an object is still in the state of the start node then its pin is omitted to indicate that the object has not yet been really activated. The resulting transitions are ordered with respect to their input and output pins, and a *start node*, a *final node*, and additional *decision nodes* and *merge nodes* are added to make the structure of the resulting process more explicit.

B.3 Integrating object life cycles of objects in a data model

Another OLC-based approach is described in [26] by Müller, Reichert and Herbst. They propose a framework that distinguishes model and instance levels when creating data-driven process structures. It is based on a data model that describes classes of objects that the process has to deal with and bi-

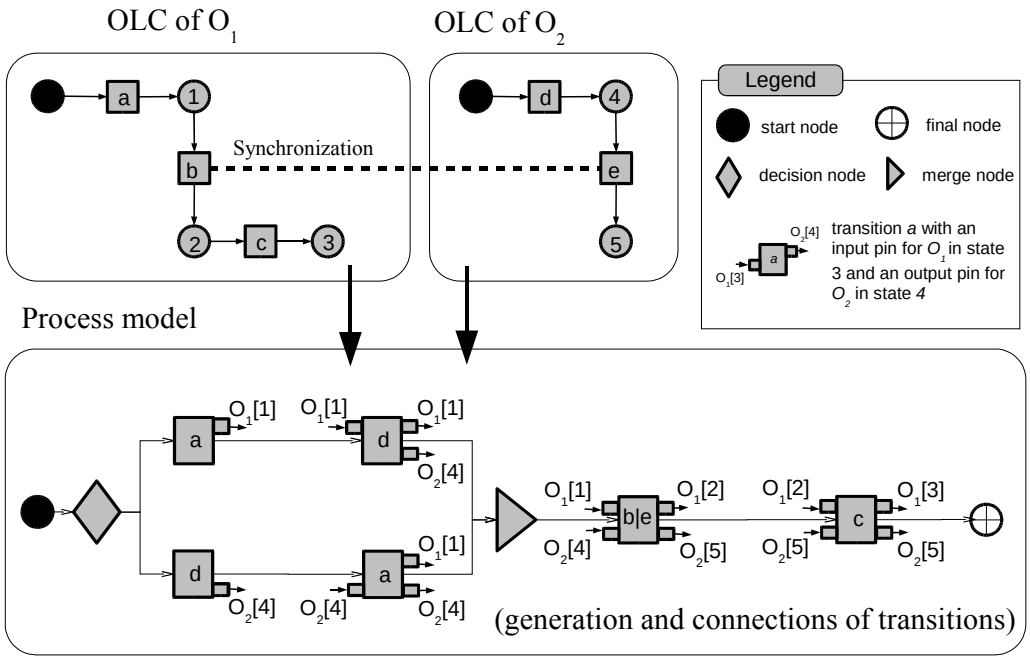


Fig. B.2. Generation of a process model from 2 OLCs with a synchornization point

nary relationships between these classes which typically represent composition relations between the objects. For an example consider the data model in Figure B.3 where the classes T and ST are specified with a relationship Has SubSys. For each class, an OLC is given that describes the behavior of the objects in this class. In addition, for the relationships *OLC dependencies* can be specified that model *external transitions* between the states of the objects in the classes that are involved in the relationship. Together this forms the *life cycle coordination model*, of which an example is given in Figure B.3, where an x transition is specified from state 2 to state 4 and a y transition from state 6 to state 3. Then, if an instance of the data model is given, as for example in the data structure in Figure B.3 where we see an object O of class T and objects O_1 and O_2 of class ST, then this defines a complete process specification which is called the *life cycle coordination structure*, which is also represented in the figure. It consists of a life cycle for each object in the data structure and all states [2] are connected by external transitions as defined by the specified OLC dependencies.

The semantics of the life cycle coordination structure are that all OLCs are executing in parallel, but are synchronized by the external transitions. The latter means that an OLC with a state that has incoming external transitions can only become the current state if the OLCs from which the transitions depart are all in the source state of these external transitions. For example,

[2] We have for simplicity omitted the special global begin and end state that are included in [26].

because of the y transitions, object O can only go to state 3 if O_1 and O_2 are both in state 6. In the other direction, because of the x transitions, objects O_1 and O_2 can only go to state 4 if O is in state 2. The process as a whole begins with each object in a special state in which it is not yet active and which has a silent transition to the begin state of the OLC. The process can end if all objects are in a final state.

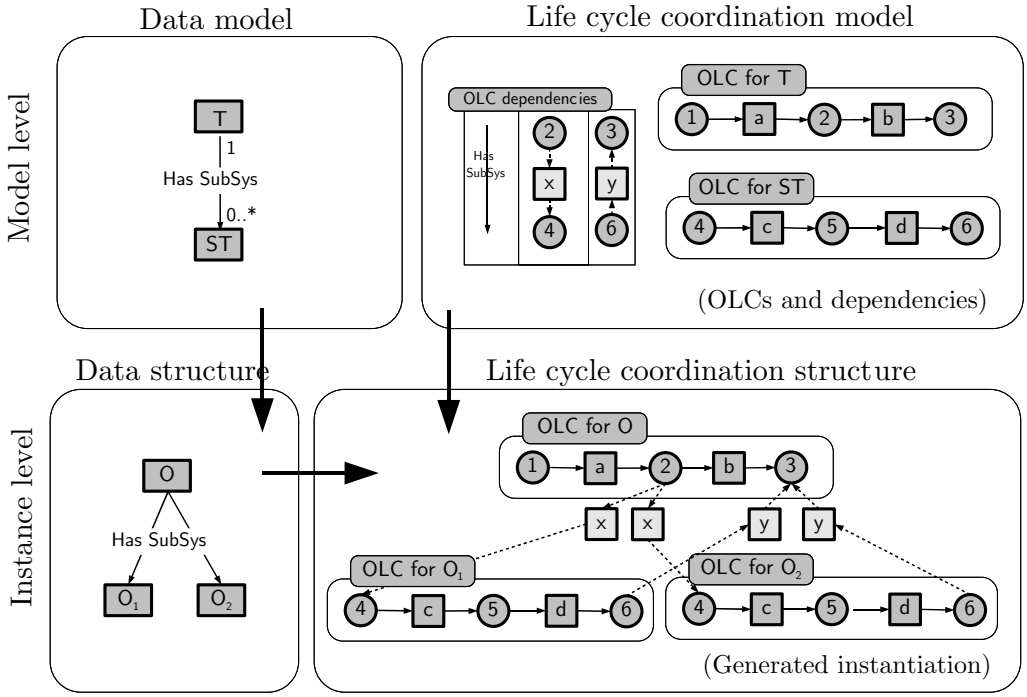


Fig. B.3. Generation of the life cycle coordination structure of three objects of two classes for which an OCL and the OCL dependencies are given

B.4 STATEMATE

The STATEMATE formalism for integrated process modeling of reactive systems is presented by Harel et al in [12]. It proposes a notation and semantics for three types of related diagrams: *module charts*, *activity charts* and *state charts*. Examples of these charts are given in Figure B.4 and we briefly explain their meaning in the following.

The module chart describes the physical structure of the system, i.e., how it is composed of *modules* which are usually physical components, but can also be more logical components such as software modules. It specifies the external modules outside the system, here EM_1, EM_2 and EM_3, and internal modules, here M_1, M_2 and SM_3, of which SM_3 is a storage module that stores information or physical objects. The modules are connected through directed *flow lines* that indicate communication between the modules. The

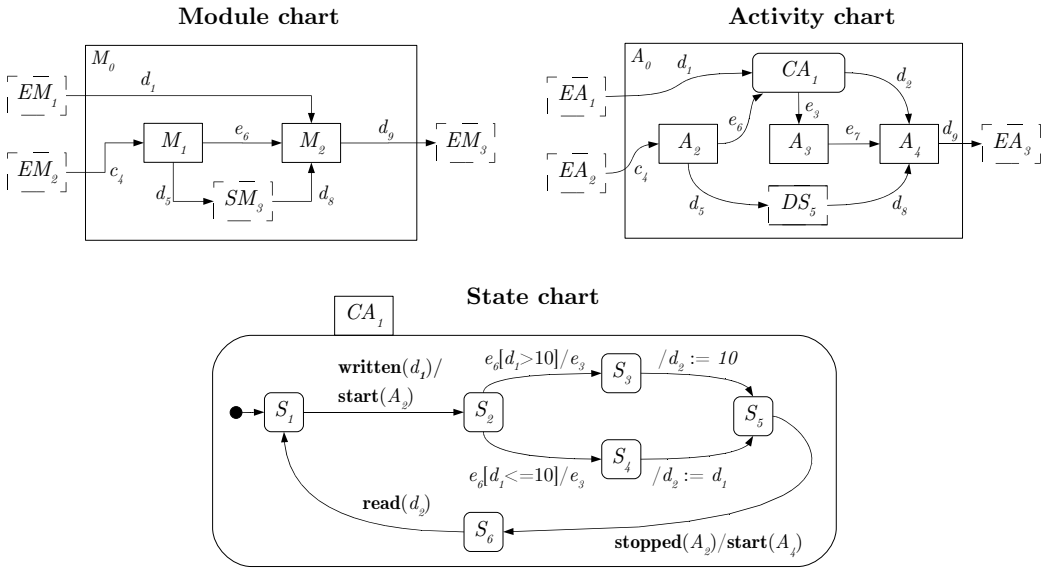


Fig. B.4. A module chart, activity chart and state chart in STATEMATE

labels indicate whether the flow lines transport data items (d_1, d_5 and d_9), conditions / booleans (c_4) or atomic events (e_6). In general module charts can be recursively nested and flow lines may cross the boundaries of such nested modules, which is an important aspect of the STATEMATE charts, but for simplicity we do not consider such nesting here.

The activity chart describes the functional structure of the system, i.e., it describes the activities and tasks that have to be performed by the system. In the example the external activities are EA_1, EA_2 and EA_3, and the internal activities are CA_1, A_1, A_2, A_3 and DS_5. Of the latter, CA_1 is a *control activity* of which there is always exactly one in each activity chart and which coordinates all the other activities. The activity DS_5 represents a data store which is an activity that only stores data items. As in the module chart the activities are connected by labeled flow lines which again indicate data flow and control flow between the activities. The relationship between the activity chart and the module chart is simply an assignment of each of the activities to one of the modules, where external activities must be mapped to external modules, internal activities must be mapped to internal modules, and data stores must be mapped to storage modules. In the example we can map the external activities EA_1, EA_2 and EA_3 to EM_1, EM_2 and EM_3, respectively, the activities CA_1, A_3 and A_4 to M_2, the activity A_2 to M_1 and the data store DS_5 to storage module SM_3. Observe that this mapping also implies which flow lines there should be in the module chart.

The state chart specifies for a control activity, in this case CA_1, how it coordinates all the activities within its activity chart. As such it defines together with the flow lines in the activity chart the complete control flow at that level.

The nested activities such as A_2, A_3 and A_4, will either be basic or have their own activity chart with a control activity for which a state chart is given. Although the notation and semantics of state charts is more general, the given example can be understood as a finite automaton with states $S_1, \ldots, S_6$ where the transitions are in general of the form $e[c]/a$ where e is a triggering event, c a condition and a an action, but all three parts are optional. Such a transition can only happen if both e is happening and c holds, and if it does then the action a is taken.

Possible triggering events include basic events arriving on incoming flow lines, e.g., an event on flow line e_1, special starting and stopping events of activities in the activity chart, e.g., **started**(A_1) and **stopped**(A_2), entering or leaving certain states in the state chart (it is in general possible to be in more than one state of the state chart), conditions on incoming flow lines becoming true or false, data-items on on incoming flow lines being written, e.g., **written**(d_1), and data-items on outgoing flow lines being read, e.g., **read**(d_2). The conditions can be boolean combinations of basic conditions specified for conditions and data items on incoming flow lines, e.g., $d_1 > 10$, for being in a certain state in the state chart, and for being active or suspended for activities in the activity chart. Finally, the possible actions include events on outgoing flow lines, e.g., an event on flow line e_3, control actions on activities in the activity chart such as starting, stopping, suspending and resuming, and writing data items and truth values to outgoing flow lines, e.g., $d_2 := 10$.

We end this treatment of state charts with a brief informal description of the semantics of the state chart in Figure B.4. We start in the initial state S_1. When a data item is written on flow line d_1, activity A_2 is started. After this we wait in state S_2 until in flow line e_6 an event appears and, depending on whether the data item d_1 is larger then 10 or not, we write either the number 10 or this data item to the flow line d_2. Then we wait in state S_5 until A_2 has stopped and start A_4. Then we wait in state S_6 until the data item written on d_2 has been read by A_4, and then we return to state S_1.

B.5 XML Nets

As an illustration of the High level Petri net approaches, we present XML nets [24]. XML nets model the data flow through a Petri net as XML documents.

XML schemas are defined in GXSL, a graphical specification language that represents a document type definition DTD by XML schema diagrams. In GXSL, element types are represented by UML classes and hierarchies are structured by aggregation. An example of an XML schema in GXSL and its equivalent DTD are given in the Figure B.5. Manipulation and querying of

XML documents are specified by an extension of GXSL, namely XManiLa, a GXSL-based document manipulation language. In XManiLa, an XML schema definition is interpreted as a template for a set of XML documents that specifies the structure of matching documents.

In XML nets, the static components (i.e., the places) are typed by GXSL specifications, each of them representing a DTD. Places are interpreted as containers of XML documents which are valid for the corresponding DTD. An edge between a place and a transition is labeled with XManiLa which specifies the prerequisite for the firing of a transition. An edge from transition to place specifies the result of a document manipulation by the transition. The initial marking assigns to each place a (possible empty) set of valid XML documents. Examples of insert and delete operations of documents are given in Figure B.5. In order to express in XManiLa that the matching documents may contain within E_1 any element preceding E_4, the rectangle inscribed with an "Any" is used.

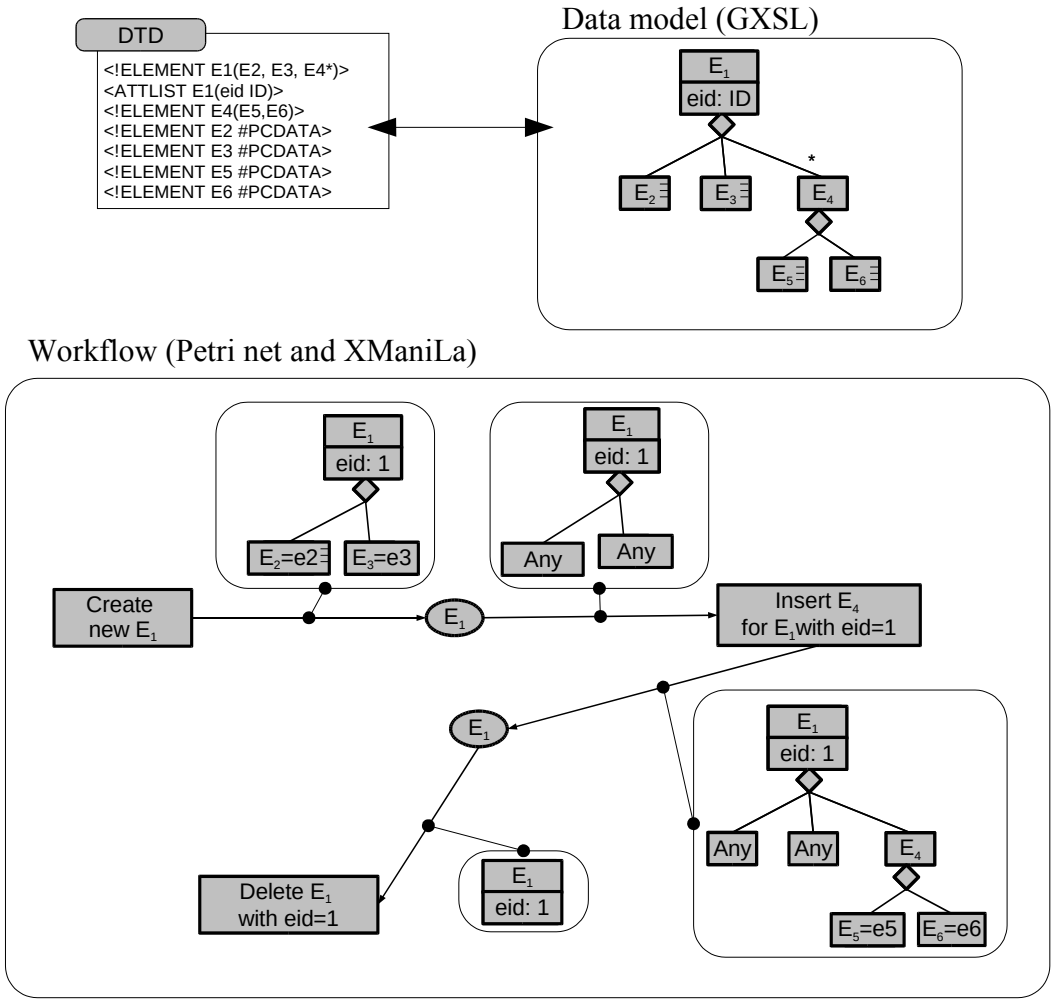


Fig. B.5. An XML net and its related XML schema diagram