Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

RESEARCH OUTPUTS / RÉSULTATS DE RECHERCHE

From Workflow Models to Document Types and Back Again

van Hee, Kees; Hidders, Jan; Houben, Geert-Jan; Paredaens, Jan; Thiran, Philippe

Publication date:
2007

Document Version
Early version, also known as pre-print

Link to publication

Citation for pulished version (HARVARD):
van Hee, K, Hidders, J, Houben, G-J, Paredaens, J & Thiran, P 2007, *From Workflow Models to Document Types and Back Again*. Technische Universiteit Eindhoven, Eindhoven.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 08. Dec. 2024

From Workflow Models to Document Types and Back Again

Kees van Hee Jan Hidders Geert-Jan Houben Jan Paredaens
Philippe Thiran

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.

Contents

1 Introduction **3**

2 Context and motivation **3**
2.1 Historical perspective 3
2.2 The relationship between workflow and document management 5
2.3 Example: Patient Care System 5

3 Jackson Types **7**

4 Jackson Nets **11**
4.1 Petri nets and workflow nets 11
4.2 Jackson nets and soundness 13

5 The Jackson Types of Jackson Nets **17**
5.1 The variety in Jackson nets generated by a certain Jackson types 17
5.2 The variety in Jackson types from which a certain Jackson net is generated . . . 18
5.3 Characterizing the expressive power of Jackson Nets 23

6 Case Study **27**

7 Related work **30**

8 Conclusion **32**

A Proof of Soundness and Generalized Soundness of Jackson Nets **35**

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 [20]. 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 [6] 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. Finally, we discuss related work in Section 7. The conclusion of the paper is given in Section 8.

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 [4]. 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), UML activity diagrams, and BPMN (business process modeling notation). 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 [6] and later the method was lifted to the level of system design, JSD, Jackson’s development method [7]. (In *Software requirements and specifications* [8] 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 [19]. 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 [9]. 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.

2.2 The relationship between workflow and document management

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 [23] 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 [17]. 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.

2.3 Example: Patient Care System

There are many Electronic Patient Record (EPR) systems that are used to record and plan the medical events in the treatment of a patient. 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.

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

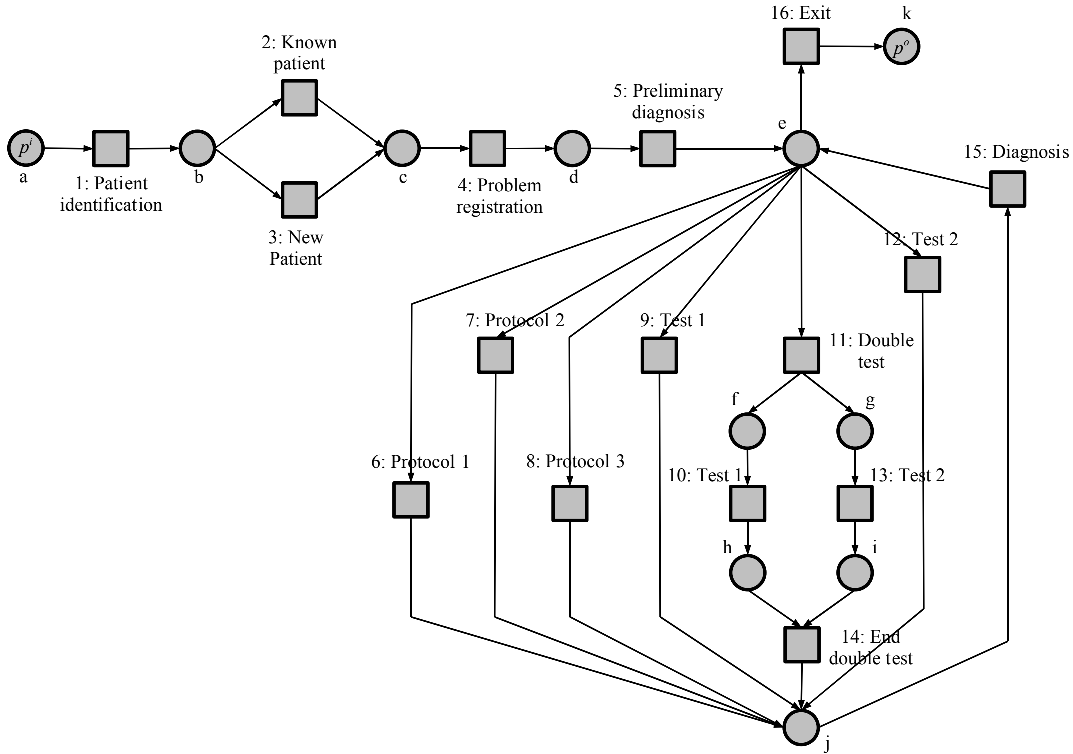

Figure 1: A workflow for handling a medical problem

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.

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

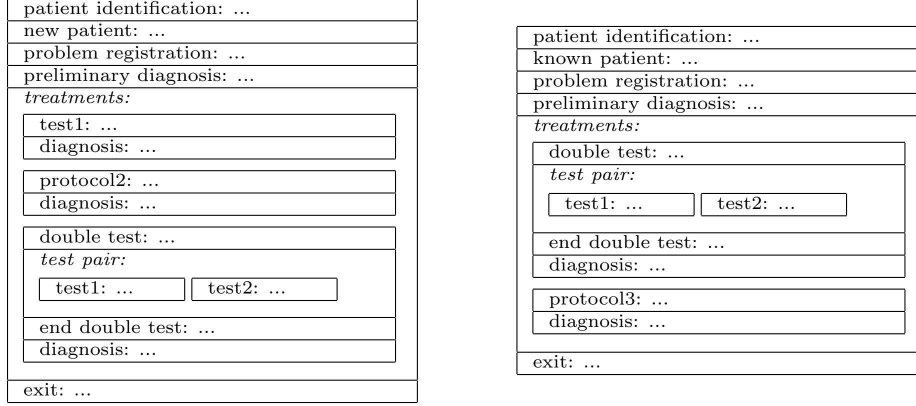

Figure 2: Examples of two patient dossiers

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

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

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 3.2 (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 3.3 (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 3.4. *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 3.5. *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 [12].

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 3.6 (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.

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.

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

Figure 3: Defining identities for algebraic equivalence

Theorem 3.7. *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 $Tr((\tau_0 \ \# \ \tau_1) \ \# \ \tau_2) = \{\alpha_1^1 \cdot \beta_1^1 \ldots \beta_{n_1-1}^1 \cdot \alpha_{n_1}^1 \cdot \gamma_1 \ldots \gamma_{k-1} \cdot {\alpha_1}^k \cdot {\beta_1}^k \ldots \beta_{n_k-1}^k \cdot \alpha_{n_k}^k | \ n_i, k > 0, \alpha_i^j \in Tr(\tau_0), \beta_i^j \in Tr(\tau_1), \gamma_i \in Tr(\tau_2)\} = \{\alpha_1 \cdot \delta_1 \ldots \delta_{m-1} \cdot \alpha_m | \ m > 0, \alpha_i \in Tr(\tau_0), \delta_i \in Tr(\tau_1) \cup Tr(\tau_2)\} = Tr(\tau_0 \ \# \ (\tau_1 + \tau_2))$.

That the converse does not hold follows from Theorem 3.4 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 3.8 (Jackson Type)**.** The set of *Jackson types* is described by the following syntax of J^0:

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

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 4.1 (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 4.2 (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 4.3 (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.

Definition 4.4 (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 4.5 (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) $m_2 = m_1 - \bullet t + t\bullet$.

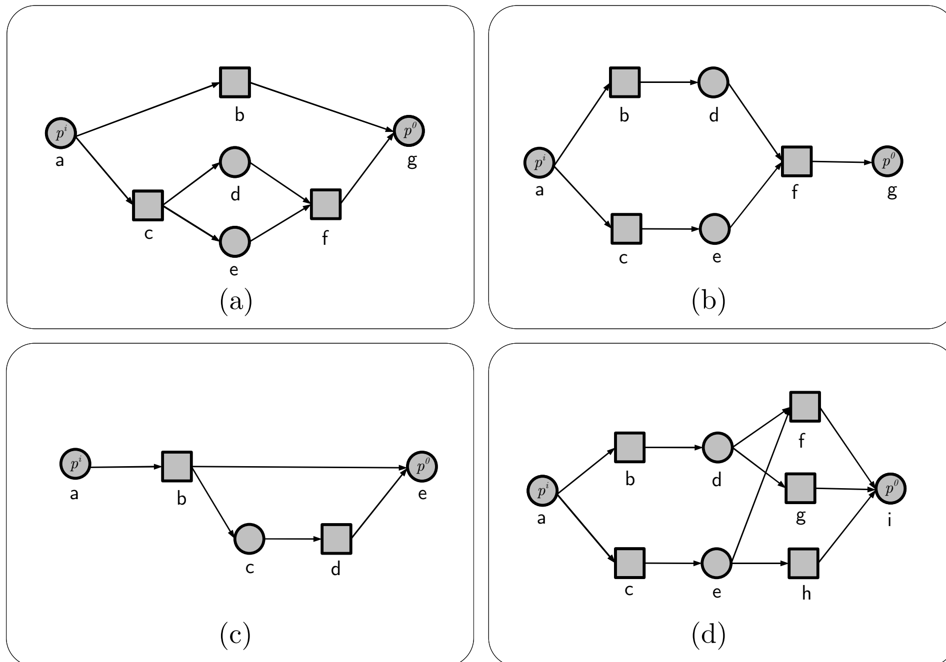

Figure 4: Four workflow nets

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 4.6 (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 4.7 (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.

Definition 4.8 (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.

[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.

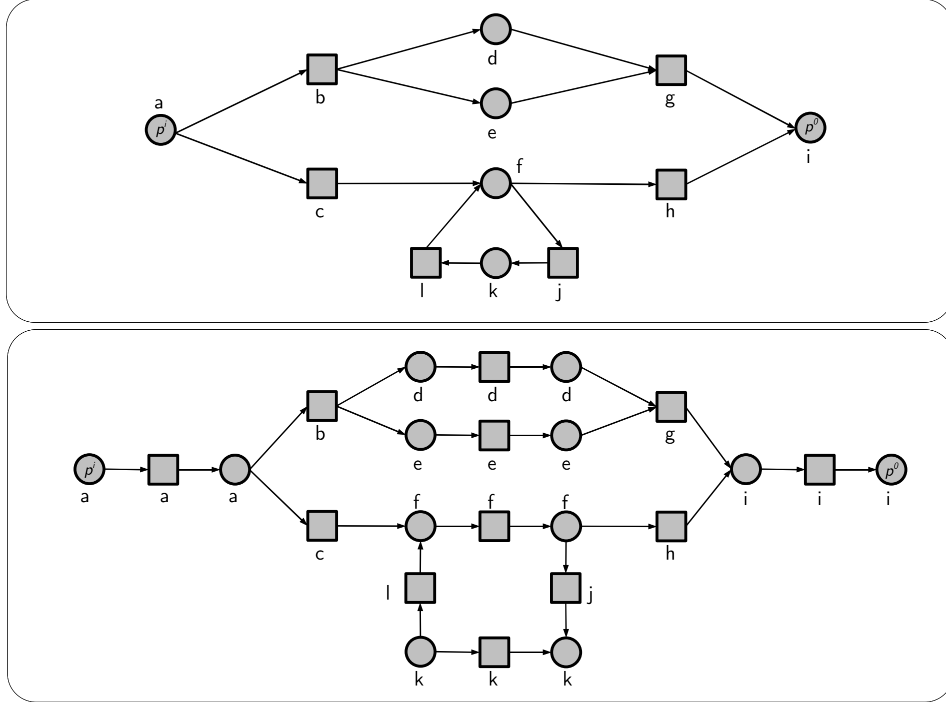

Figure 5: A net with its place-expanded net

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 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.

Definition 4.9 (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 4.10 it holds that every Jackson net is a sound net, but the converse does not hold as is demonstrated in Theorem 4.11 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 holds for the reverse *is-generated-from* relationship, which can be assumed to be defined by the same

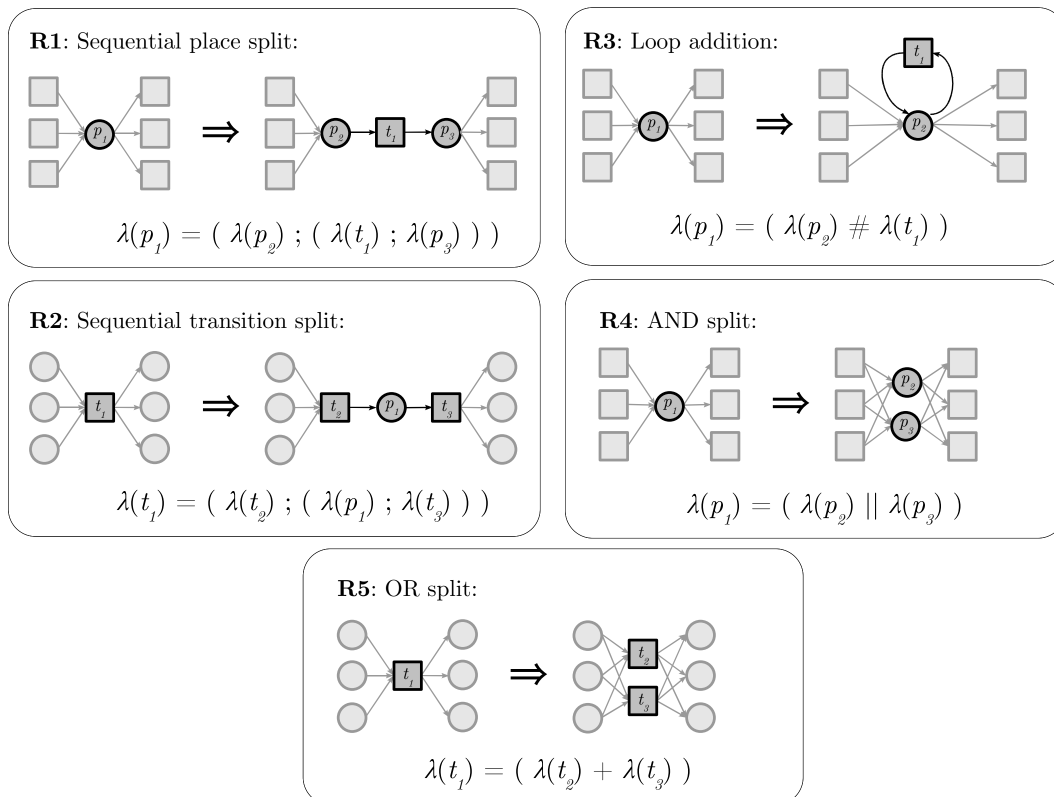

Figure 6: The generation rules for Jackson nets

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 [2] and Murata in [13] 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 [16] and Chrzastowski-Wachtel et al. in [3] 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 4.10. *Every Jackson net is a sound net.*

It is well-known that the Murata rules preserve liveness and boundedness of Petri nets (see [13]) 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 [22] it is shown that soundness is equivalent with liveness and boundedness

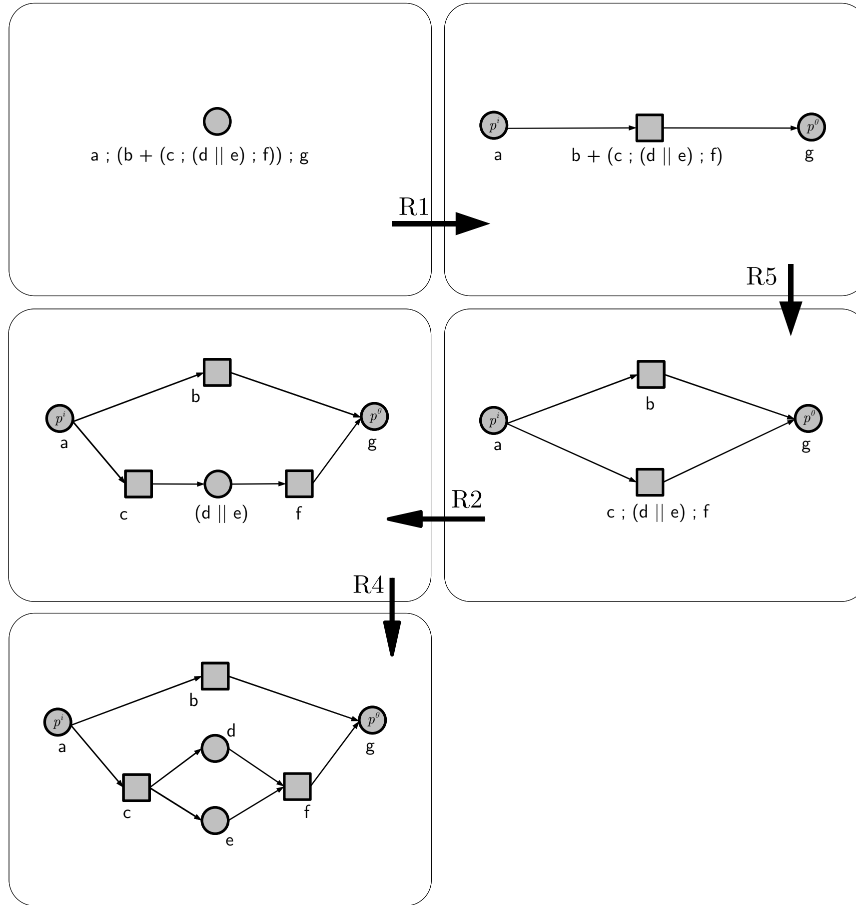

Figure 7: Generation of a net

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* [25] 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 4.11. *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. □

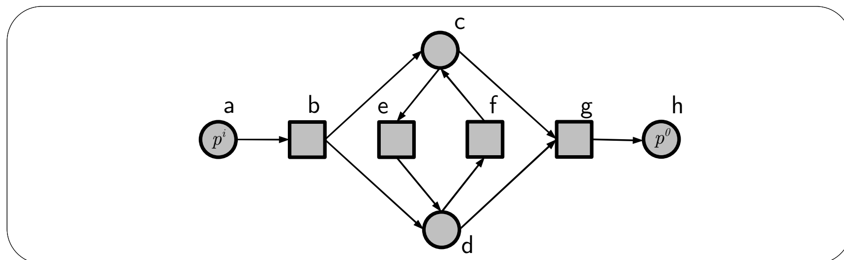

Figure 8: Not a Jackson net

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 5.1) 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 5.8). 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 5.1. *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 3.1. 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. □

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 5.2. *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 $Tr(\Omega) = Tr(\tau)$. □

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

In order to prove Theorem 5.3 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 5.4 (Simple Type)**.** The set of *simple types* is defined by the following syntax of J^S:

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

As for normal types we can similarly define algebraic equivalence.

Definition 5.5 (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.

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

Figure 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 5.6 (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.*

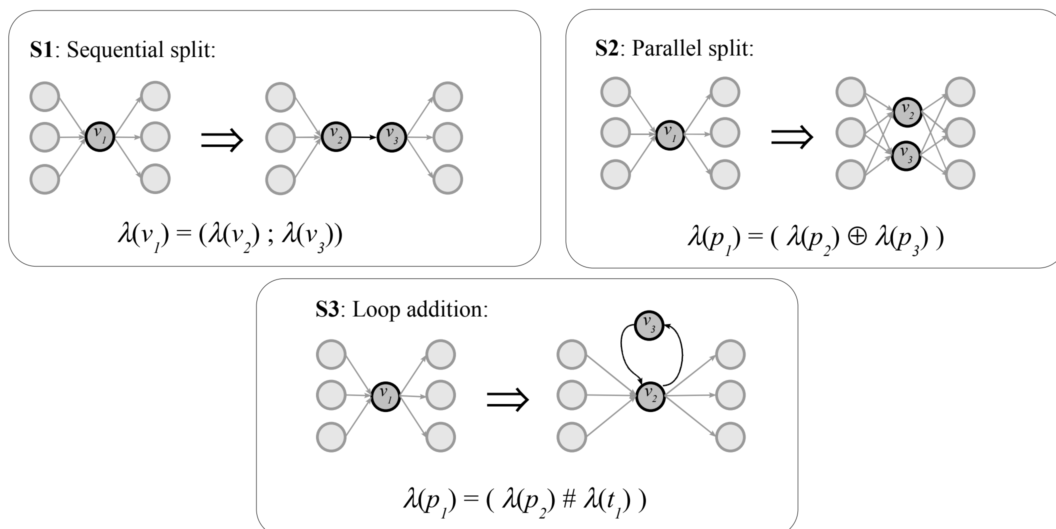

Figure 10: The generation rules for simple Jackson graphs

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 5.7. *If two simple types τ and τ' generate the same simple Jackson 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 5.1 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. □

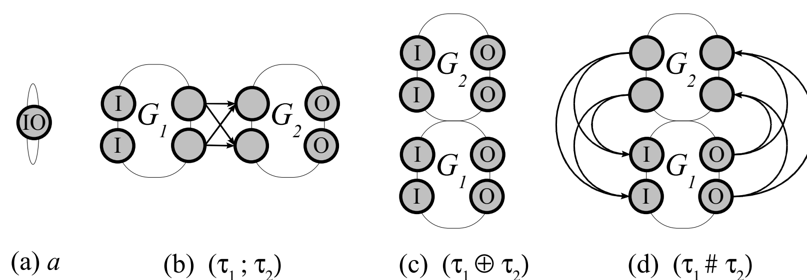

Figure 11: Decompositions of simple Jackson graphs based on their generating type

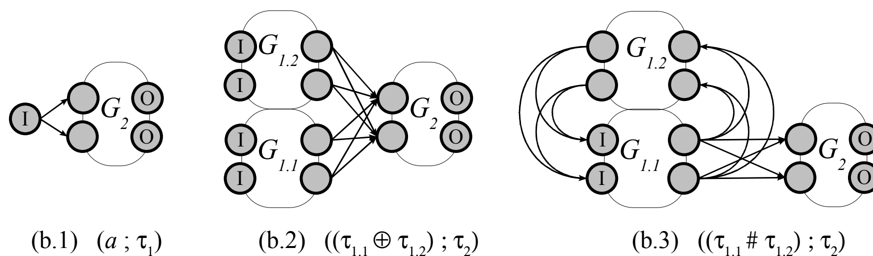

Figure 12: Decompositions based on sequence types

Using Lemma 5.7 we can now prove Theorem 5.3.

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 5.1 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

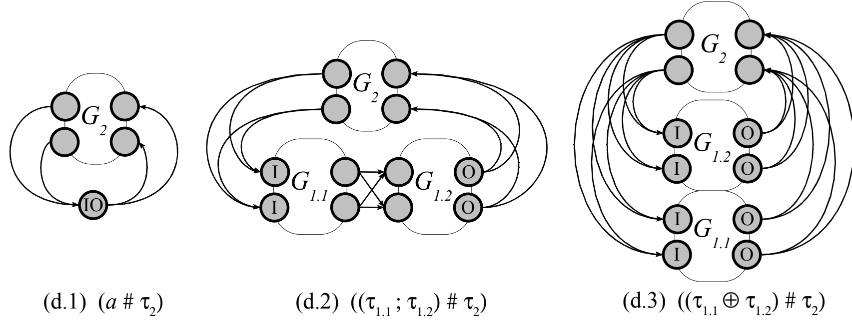

Figure 13: Decompositions based on iteration types

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 5.8. *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 5.3 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 5.3 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 5.1 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 5.9 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 5.13, 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 5.9. *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 5.10. *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 $\Omega_k = \Omega$. □

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 5.11. *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 4.10, 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. □

Lemma 5.12. *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. □

We now proceed with the proof of Theorem 5.9:

Proof. The only-if part of the theorem follows from the definition of Jackson net, Lemma 5.10 and Theorem 5.2.

By definition of Jackson net and Theorem 5.2 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 4.10 and Lemma 5.11. Since $Tr(\Omega') = Tr(\tau) = Tr(\Omega)$ it follows by Lemma 5.12 that Ω and Ω' are isomorphic. Since Ω' is a Jackson net, it follows that Ω is also a Jackson net. □

With this result we can now extend Theorem 5.8 as follows.

Corollary 5.13. *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 5.8. That (3) $\Leftrightarrow$ (4) follows from Theorem 5.2. That (2) $\Rightarrow$ (3) follows from Theorem 3.7. That (3) $\Rightarrow$ (1) follows from Lemma 5.12 and Lemma 5.11. □

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 3.4.

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.

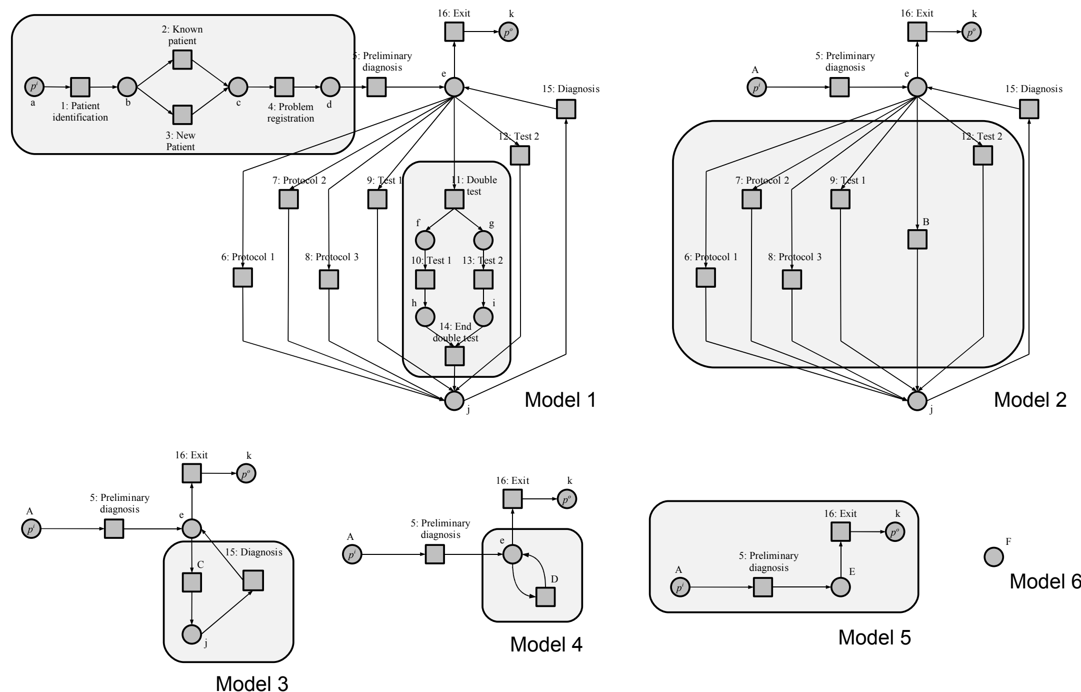

Figure 14: Generation of the singleton net

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,

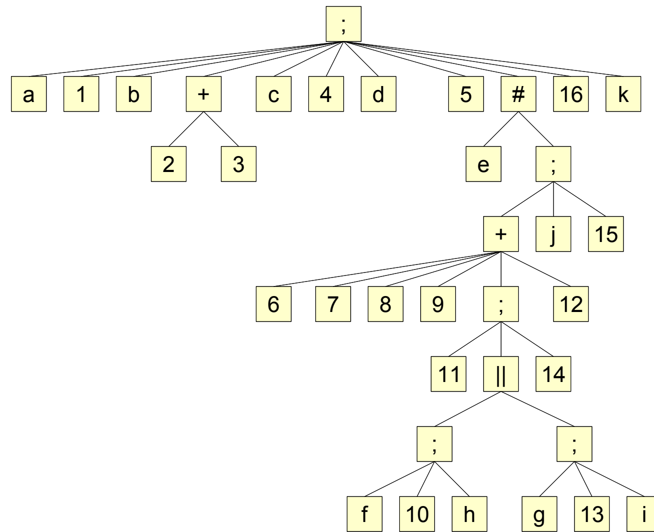

Figure 15: Jackson type as tree

choice or loop constructors) and that the leaf nodes represent places or transitions in the Jackson net. To make a useful document type of 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.

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, 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

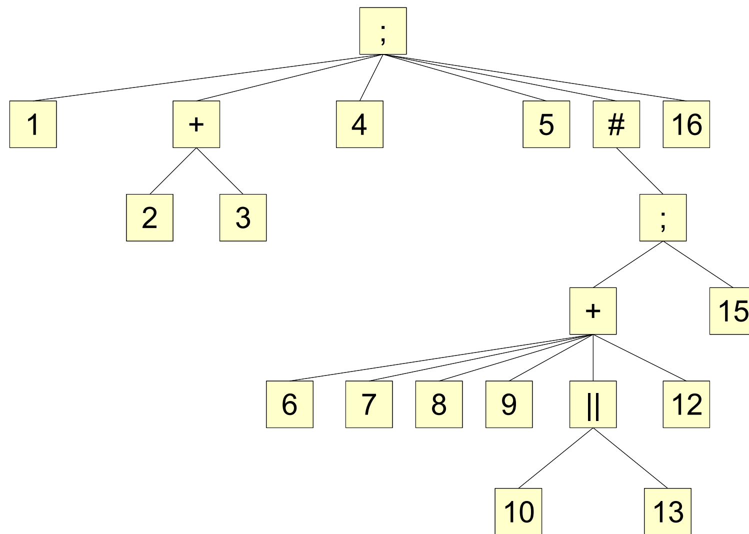

Figure 16: Reduced tree

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 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 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?

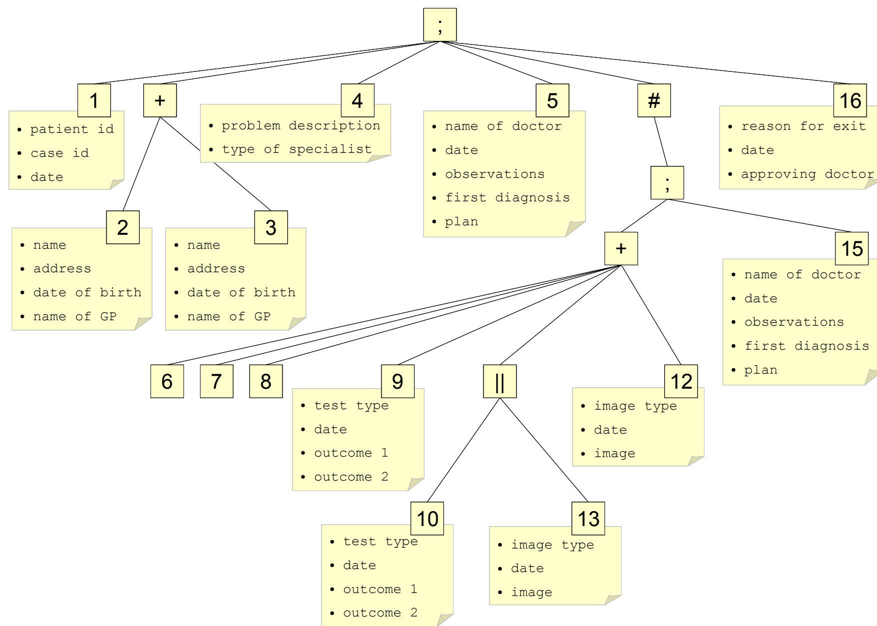

Figure 17: Case tree

```
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 Related work

The problem that we addressed in this paper is the integration of the data and the process views. In the existing literature on this problem we can find two different approaches to achieve this integration.

In the first approach, data and process are loosely coupled in that data and processes are modelled in two linked but separate models. They allow the specification of links between process models and data models but these models remain essentially independent. Their aim is to explicitly describe the data that is used in a process. They therefore do not support the derivation of one model from the other. Case-handling workflows [18], NR/T-nets [14], XML Nets [11] are examples of such models. They can be described as a kind of high-level Petri nets. The 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

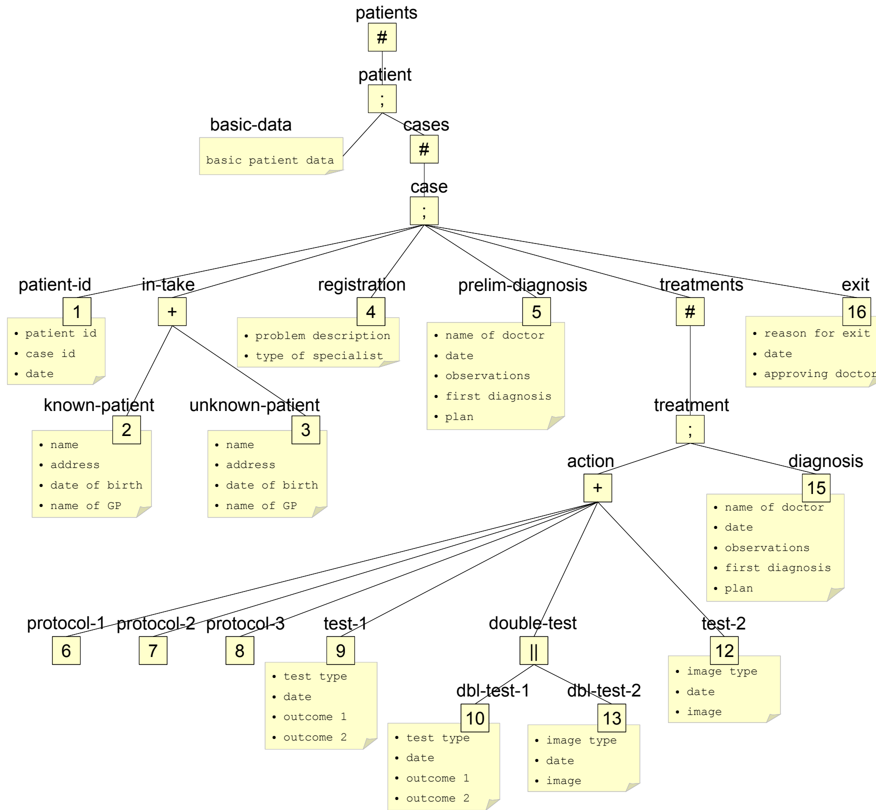

Figure 18: Patients tree

over the consumed tokens that appear as labels of the adjacent edges. Another example of loosely coupled approach is PPM (Process and Product Modeling) [10]. This approach provides a formal framework that allows defining the data flow in a process and validate its consistency.

In contrast, the second approach supports the derivation of the proces model from the data model or vice versa. Data and process models are here *tightly coupled* in that data and processes structures are closely related and changes in one are likely to cause changes in the other. As an example of such models, the Product-Based Workflow Design approach ([17], [21]) 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) [15] 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 [17], [21], 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.

Our paper is closely related to the Product-Based Workflow Design approach and extends it in two ways. First, our data model includes the iteration operator and hence allows modeling loops in the Petri net. Second, we define the correspondence in the two directions: not only from the data model to the process model but also in the other direction.

```
<!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? )>
  ....
]>
```

Figure 19: A partial DTD for the Patients tree

8 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 [5] but no attempt was made yet to integrate it into the framework that is presented here.

Acknowledgment: The authors would like to thank the anonymous referees whose remarks have helped to improve considerably the 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] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

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

[9] 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.

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

[11] 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.

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

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

[24] 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.

[25] 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 A.1 (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 4.6 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 [24] 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 4.10 by proving the following stronger theorem.

Theorem A.2. *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 generallzed 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. □