The Analysis of Data-Flow and Control-Flow in Workflow Processes Using Maude

Oana Otilia Captarencu
Faculty of Computer Science, "Alexandru Ioan Cuza" University, Iasi, Romania

Keywords: Workflow Processes Verification, Data Flow Errors, Petri Nets, Model Checking, Rewriting Logic, Maude.

Abstract: A business process consists of coordinated tasks that take place inside an organization in order to achieve a specific business objective. The process also involves data, that can be used/produced by tasks or used to control the execution of tasks. A workflow is the automation of a business process. Verification of workflow correctness usually focuses on the control-flow of workflows (which regards the tasks and their order of execution). Data anomalies can prevent the correct execution of the process, even if it is correct at the control-flow level, or can produce undesired results. Thus, data information plays an important role in workflow analysis. In this paper we propose an analysis technique for workflows with data, based on Petri nets and on the rewriting logic based language Maude: we use a special class of Petri nets, workflow nets with data, to model workflow processes with data and propose a translation of workflow nets with data into rewrite theories in Maude. This will allow the application of model checking techniques to detect data errors, verify specific properties regarding data and verify the correctness of the workflow.

1 INTRODUCTION

Information systems that allow the design, management, optimization and automation of business processes are critical to the success of businesses and organizations. A business process consists of a set of coordinated tasks organized to achieve a specific business objective. The process also involves resources necessary for executing the tasks and data: data used/produced by tasks or data used to control the execution order of tasks. A workflow is defined as the automation of a business process. A large body of research has been dedicated to the formal modelling and verification of workflow processes. One aspect that has been intensively studied is the verification of the control-flow of workflows, which refers to the tasks and their order of execution. As a result, several notions of workflow correctness have been proposed (Aalst et al., 2010). The data aspect (the data-flow) is an important part of the workflow process: even if the process is correct at the control-flow level, data-flow anomalies can prevent the correct execution of the process or can produce undesired results. Most of the research regarding the data-flow aims either at defining and detecting data-flow errors in workflow processes or at extending the notion of correctness in order to capture the interplay between the control-flow and data-flow in workflows, but do not offer analysis techniques to do both. The current approaches assume the workflow uses a set of data elements and consider a set of data operations that can be performed on data elements during the execution of tasks (read, write, delete). Some approaches also consider data dependent predicates that are used to control the execution order of tasks in the workflow.

In many research approaches that try to identify the data flow errors, the workflow process has various restrictions: it contains only simple loops (Sun et al., 2006), it considers only the read/write operations (Mülle et al., 2019; von Stackelberg et al., 2014; Eshuis, 2006), it does not consider data dependent predicates for controlling the task execution (Eshuis, 2006; Sun et al., 2006; Meda et al., 2007; Liu et al., 2020; Xiang et al., 2017; Xiang et al., 2021a). Even in the approaches in which the process has no such restrictions (Zhao et al., 2022; Xiang et al., 2021b), only some data flow errors are considered or the correctness of the workflow with data restrictions is not studied.

The approaches that discuss the correctness of the workflow with data (Kheldoun et al., 2017; Sidorova et al., 2011; Fan et al., 2007; He et al., 2018) usually do not address the identification of data flow errors. Trčka et al. (Trčka et al., 2009) use temporal logic to

Captarencu, O.
The Analysis of Data-Flow and Control-Flow in Workflow Processes Using Maude.
DOI: 10.5220/0011844800003467
In *Proceedings of the 25th International Conference on Enterprise Information Systems (ICEIS 2023) - Volume 2*, pages 373-384
ISBN: 978-989-758-648-4; ISSN: 2184-4992
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

describe a set of data-flow errors and a notion of correctness for the workflow. They propose a Petri Net model - workflow nets with data (WFD-nets), they define an unfolding of WFD-nets into classical Petri nets and use it to apply model-checking techniques to verify the existence of data-flow errors and the correctness of the workflow. This solution does not allow the explicit representation of data elements and suffers from the state explosion problem, due to the size of the unfolded net.

The existing correctness notions do not impose any conditions on data when the workflow terminates, although workflow processes are often required to produce specific data that should be available at the end of the process. In this paper we propose notions of correctness that permit the specification of data elements that should be present when the process starts and when the processes terminates. It will be possible to specify that all data elements from a given set should be present or at least one element from a given set should be present when the process terminates.

Our approach uses the WFD-nets defined in (Trčka et al., 2009) as a model for workflow processes with data and proposes the specification of WFD-nets as rewrite theories in Maude, providing a formal semantics for WFD-nets. Maude is a declarative programming language and high-performance system based on rewriting logic. Rewriting logic is a computational logic that can naturally deal with state and concurrent computations and it has been used as a semantic framework for a wide range of languages and models of concurrency, including Petri nets (Meseguer, 1992). This approach will allow us to use temporal logic and the efficient Maude model checking tool to verify the existence of data anomalies, specific data properties and correctness properties for workflow processes with data.

2 PRELIMINARIES

This section reviews the terminology and notations related to Petri nets and workflow nets. For details, the reader is referred to (Reisig, 1985), (Aalst, 1998).

A Petri net is a triple $N = (P,T,F)$, where P is a finite set of places, T is a finite set of transitions ($P \cap T = \emptyset$), $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation).

For $x \in P \cup T$, the *preset* of x is $\bullet x = \{y|(y,x) \in F\}$ and the *postset* of x is $x\bullet = \{y|(x,y) \in F\}$. The state (or the marking), is a function: $m : P \rightarrow \mathbb{N}$ ($\mathbb{N}$ denotes the set of natural numbers). A marking can be seen as a multiset of places.

A transition t is said to be enabled (in a marking m) iff: $m(p) \geq 1, \forall p \in \bullet t$. If an enabled transition t fires, it changes the marking m into m': $m'(p) = m(p) - 1, \forall p \in \bullet t$ and $m'(p) = m(p) + 1, \forall p \in t\bullet$. We write $m[t\rangle m'$.

If $\sigma = t_1 t_2 \dots t_n \in T^*$ is a set of transitions such that $m_1[t_1\rangle m_2[t_2\rangle \dots [t_n\rangle m_n$, we also write $m_1[\sigma\rangle m_n$ (or $m_1[*\rangle m_n$). Marking m_n is said to be *reachable* from marking m_1. If (N,m_0) is a marked Petri net, the set of reachable markings of N is denoted by $[m_0\rangle$. Let (N,m_0) be a marked Petri net. A transition t is *quasi-live (not dead)* in (N,m_0) if $\exists m \in [m_0\rangle$ such that $m[t\rangle$.

Workflow nets (WF-nets) are are widely used Petri net model for workflow processes, introduced in (Aalst, 1998).

A Petri net WF=(P,T,F) is a WF-net iff: (1) PN has a source place i and a sink place o such that $\bullet i = \emptyset$ and $o\bullet = \emptyset$ and (2) if we add a new transition t^* to PN such that $\bullet t^* = \{o\}$ and $t^*\bullet = \{i\}$, then the resulting Petri net is strongly connected.

The marking m with $m(i) = 1$ and $m(p) = 0, \forall p \neq i$ represents the beginning of the process (the initial marking, denoted by i). The marking m with $m(o) = 1$ and $m(p) = 0, \forall p \neq o$, represents the termination of the process (the final marking, denoted by o).

A notion of soundness was defined for WF-nets, expressing the minimal conditions a correct workflow should satisfy (Aalst, 1998).

A workflow net $WF = (P,T,F)$ is sound iff: (1) the final marking o can be reached from every reachable marking m (termination condition): $(\forall m)((i[*\rangle m) \Longrightarrow (m[*\rangle o))$; (2) all the transitions in WF are quasi-live:$(\forall t \in T)(\exists m,)(i[*\rangle m[t\rangle)$.

3 WORKFLOW NETS WITH DATA

Workflow Nets with Data (WFD-nets) have been introduced in (Trčka et al., 2009). WFD-nets extend WF-nets with data elements, three possible operations on data elements (read, write, delete) and guards for transitions.

If D denotes the set of data elements, a predicate is an expression of the form $P(d_1,\dots,d_n)$, where P is the predicate name, $n \geq 0$, $d_i \in D, \forall i \in 1\dots n$. We refer to a predicate $P(d1,\dots,d_n)$ simply by P and we denote by $vars(P)$ the data elements the predicate depends on (i.e., $vars(P) = \{d_1,\dots,d_n\}$).We denote by $Preds$ the set of predicates. A guard for a transition is a predicate or the negation of a predicate (over D). The following definition introduces Workflow Nets with Data (WFD-nets):

A WFD-net is a tuple
$WFD = (P,T,F,D,\mathcal{G},Read,Write,Delete,Guard)$

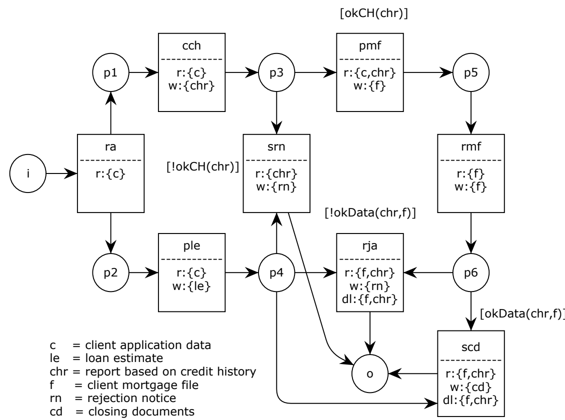

Figure 1: An example of a WFD-net.

- (P,T,F) is a WF-net (the underlying WF-net);
- D is the set of data elements;
- $\mathcal{G}$ is the set of guards over D: $\mathcal{G} = \{P|P \in Preds\} \cup \{!P|P \in Preds\}$;
- $Read : T \to 2^D$ the data elements read by each transition in T
- $Write : T \to 2^D$ the data elements written by the transitions in T
- $Delete : T \to 2^D$ the data elements deleted by the transitions in T
- $Guard : T \to \mathcal{G}$ is a partial function which assigns guards to transitions in T

The set of data elements used by a transition t is $vars(t) = Read(t) \cup vars(Guard(t))$. A transition t in a WFD-net can fire if it is enabled in the marking of the underlying WF-net, if all the data elements it uses (data elements from $vars(t)$) are defined and its guard is true. If some data element is written by a transition, it becomes available /defined. A data element that is deleted by a transition becomes undefined. If a transitions performs all the operations on a data element d, the order of operations is: read, write, delete.

A configuration of a WFD-net is : $< M,\ DS,\ \sigma >$, where M is the marking of the net, DS is the set of defined data elements, $\sigma : Preds \to \{true,\ false\}$ is a function that describes the value of every predicate.

In our approach we will allow the specification of a set of data elements available/defined when the process starts. Thus, the initial configuration is $< i,\ DS,\ \sigma >$, where DS is a set of initial data elements and $\sigma(P) = false, \forall P \in Preds$.

The function $defined : Preds \times 2^D \to Bool$ is defined such that for any $P \in Preds$ and $DS \subseteq D$, $defined(P,DS)$ is $true$ iff $vars(P) \subseteq DS$.

Function $definedPreds : 2^{Preds} \times 2^D \to Bool$ is the extension of $defined$ to sets of predicates.

The example in Figure 1 depicts a WFD-net modelling the processing of a mortgage application in a bank. The data elements involved are: $\mathtt{D} = \{\mathtt{c}, \mathtt{chr}, \mathtt{f}, \mathtt{le}, \mathtt{rn}, \mathtt{cd}\}$. The data element available (defined) when the process starts is the client application data (`c`) - which includes the client personal information, property information, client debts, income etc; Transition `ra` (receive application) performs a read operation on `c`. Afterwords two actions will be executed in parallel: `ple` (provide loan estimate) - the client is provided with a loan estimate (the data element `le` is written) which details the estimated closing costs and the monthly payment; `cch` (check credit history): the loan officer will check the clients' credit history, providing a report (`chr`). After `cch`, if predicate `okCHR(chr)` is `false`, a rejection notice is sent to the client (`srj`), explaining the reason for the rejection of the application. The data element `rn` (rejection notice) is written. If `okCHR(chr)` is `true`, `pmf` (prepare mortgage file) is executed: a loan officer verifies all the information provided by the client, adding additional information (title search, tax transcripts etc) and prepares a file: the data element `f` is written and `r`, `chr` are read. The next action, `rmf` (review mortgage file), is performed by another employee who reviews the file (`f` is read) and updates it with additional data (`f` is written again). The action `scd` (send closing documents) will be executed if the predicate `okData(chr,f)` is `true` (the loan is approved). In this case, a set of closing documents (`cd`), will be written and sent to the client for signing and the workflow terminates. Data elements `f` and `chr` are deleted, as they are no longer needed. If the predicate is `false`, the action `rja` (reject application) is performed: a rejection notice (`rn`) is written and `f` and `chr` are deleted.

4 SPECIFICATION OF WORKFLOW NETS WITH DATA IN MAUDE

Maude is a high-level language and an efficient system based on rewriting logic. Rewriting logic is a computational logic that can naturally deal with state and express both concurrent computations and logical deduction (Meseguer, 1992).

A concurrent system can be specified in Maude as a rewrite theory $(\Sigma; E \cup A; R)$ in the rewriting logic. $(\Sigma; E \cup A)$ is a theory in the equational membership logic, which specifies the static part of the system (its states) and R is the set of possibly conditional rewrite rules describing the dynamic part of the system - the

transitions between the states.

Σ, called the signature, specifies the type structure: sorts, subsorts, kinds, and operators. E is the collection of memberships and equations between terms and A is the collection of equational attributes (assoc, comm, and so on) declared for the different operators.

The (possibly conditional) rewrite rules have the form $t \rightarrow t'$ `if` C where t and t' are terms in Σ and C is a condition. A rule $t \rightarrow t'$ can be seen as a local state transition, stating that if a portion of a system's state matches the pattern described by t, then that portion can change to the corresponding instance of t'. The rule can be applied only if the condition C is satisfied. Furthermore, such a local state change can take place concurrently with any other nonoverlapping local state changes. Rewriting logic is therefore a logic of concurrent state change.

A Maude program containing only a theory in the membership equational logic is called a functional module. Computation in a functional module is accomplished by using the equations as rewrite rules: each step of rewriting is a step of replacement of *equals by equals* until a canonical form is found.

A Maude program containing the specification of a rewrite theory is called a system module. Computation in such a module is rewriting logic deduction, in which rewriting in the membership equational logic with the axioms $E \cup A$ is combined with rewriting computation with the rules R.

A concurrent system can be specified using a functional module which contains the equational theory describing its states and a system module which includes the functional module and specifies the set of rewrite rules describing the transitions between states.

In order to specify WFD-nets in Maude, we will first describe the structural aspects of a WFD-net, as well as the markings and configurations of a WFD-net using a theory in the equational logic.

The sorts of the equational theory are: `Places` (the sort that represents the places of a WFD-net); `Transitions` (the sort that represents the transitions of the net); `Marking` (the markings of a WFD-net); `Config` (the configurations of a WFD-net); `Data` (the set of data elements); `DataOp` represents a set of operations on data elements (for any $\mathtt{d} \in \mathtt{Data}$, `DataOp` will contain elements `w(d)`, `r(d)`, `dl(d)` which correspond to the possible data operations on `d`); `PredValue` (a set of predicate values).

We will also use a predefined data type in Maude - a parametrized set, in order to represent sets of `Data` elements (`Set{Data}`), sets of predicates (`Set{Preds}`), sets of data operations (`Set{DataOp}`) and sets of predicate values (`Set{PredValue}`).

We define the subsort relation: `subsort Places < Marking` (which describes that any element of `Places` is also an element of `Marking`).

The set of operators related to the sort `Marking` are declared in Maude as follows:

```
op null  : -> Marking .
op __ : Marking Marking -> Marking
        [ctor assoc comm id: null] .
```

According to this declaration, `null` is a constant (representing the empty marking). A union operator constructs a marking from two existing markings. The attribute "assoc" (resp. "comm") is used to declare that the operator is associative (resp. commutative).

The constructor operators `w`, `r` and `dl`, with the domain sort `Data` and the result sort `DataOp`, construct the elements of `DataOp`:

```
ops w r dl :  Data -> DataOp [ctor] .
```

The operator $\{_ : _\}$ is used to construct predicate values which can be assignments of the form $\{\mathtt{P} : \mathtt{value}\}$, where `P` is a predicate and $\mathtt{value} \in \{\mathtt{true}, \mathtt{false}\}$:

```
 op {_:_} : Preds Bool -> PredValue [ctor] .
```

We also define operators that will describe several functions on the data types (together with equations that will define their semantics) and a constructor operator for configurations:

```
op vars : Preds -> Set{Data} .
op defined : Preds Set{Data} -> Bool .
op definedPreds : Set{Preds} Set{Data}
                -> Bool .
op changeValues : Set{PredValue} Set{Preds}
                -> Set{PredValue} .
ops readOps writeOps deleteOps : Set{Data}
                              -> Set{DataOp} .
ops read write delete dops : Transitions
                      -> Set{Data} .
var Pr : Set{PredValue} .
var b : Bool .
vars W pn : Preds .
var pset : Set{Preds} .
var DS : Set{Data} .
var E : Data .
eq defined(pn,DS) = intersection(vars(pn),DS)
                  == vars(pn) .
eq definedPreds(empty,DS) = false .
eq definedPreds(pn,DS) = defined(pn,DS) .
eq definedPreds ((pset,pn), DS) =
   definedPreds(pset,DS) and defined(pn,DS) .

eq changeValues(empty, pset) = empty .
eq changeValues (Pr, empty) = Pr .
eq changeValues((Pr,{W : b}), W) =
              Pr,{W : not b} .
eq changeValues ((Pr,{W : b}), (pset,W) ) =
   changeValues(Pr,pset),{W : not b} .
```

```
eq changeValues (Pr, pset)  = Pr [owise] .

eq readOps(empty) = empty .
eq readOps(E) = r(E) .
eq readOps ((DS,E)) = readOps(DS),r(E) .
eq writeOps(empty) = empty .
eq writeOps(E) = w(E) .
eq writeOps ((DS,E)) = writeOps(DS),w(E) .
eq deleteOps(empty) = empty .
eq deleteOps(E) = dl(E) .
eq deleteOps ((DS,E)) = deleteOps(DS),dl(E) .

var t : Transitions .
eq dops(t) = readOps(read(t)),
             writeOps(write(t)),
             deleteOps(delete(t)) .

op <_,_,_,_> : Marking Set{Data}
               Set{PredValue}
               Set{DataOp} -> Config [ctor].
op initConfig : -> Config .
```

The operators `vars`, `defined` and `definedPreds` will be used to describe the functions with the same name defined in section 3.

The operator `changeValues` will describe a function which, given a set of elements from `PredValue` (a set $\{\{\mathtt{P}:\mathtt{value}\}|\mathtt{P}\in\mathtt{Preds}\}$) and a set of predicate names `PSet`, will produce the following set: $\{\{\mathtt{P}:\mathtt{value}\}|\mathtt{P}\in\mathtt{Preds}\setminus\mathtt{PSet}\}\cup\{\{\mathtt{P}:\neg\mathtt{value}\}|\mathtt{P}\in\mathtt{PSet}\}$ (the value of the predicates from `PSet` will be changed).

The operators `readOps`, `writeOps` and `deleteOps` will produce a set of `DataOp` elements corresponding to the data elements in the argument (DS): $\mathtt{readOps}(\mathtt{DS})=\{\mathtt{r}(\mathtt{d})|\mathtt{d}\in\mathtt{DS}\}$, $\mathtt{writeOps}(\mathtt{DS})=\{\mathtt{w}(\mathtt{d})|\mathtt{d}\in\mathtt{DS}\}$, $\mathtt{deleteOps}(\mathtt{DS})=\{\mathtt{dl}(\mathtt{d})|\mathtt{d}\in\mathtt{DS}\}$.

The operators `read`, `write` and `delete` will describe the data elements read, written or deleted by a transition. The operator `dops` will produce the set of data operations for a given transition: $\mathtt{dops}(\mathtt{t})=\{\mathtt{r}(\mathtt{d})|\mathtt{d}\in\mathtt{Read}(\mathtt{t}))\}\cup\{\mathtt{w}(\mathtt{d})|\mathtt{d}\in\mathtt{Write}(\mathtt{t})\}\cup\{\mathtt{dl}(\mathtt{d})|\mathtt{d}\in\mathtt{Delete}(\mathtt{t})\}$.

The configuration of a WFD-net is given by the sort `Config`, defined using the constructor operator `op <_,_,_,_>` : the first three arguments of the operator correspond to the elements from the definition of a configuration from section 3, while the forth argument will be a set of data operations performed in the configuration. The constant operator `initConfig` will describe the initial configuration.

Places of a WFD-net will be defined as constants of sort `Places`, transitions as constants of sort `Transitions`, data elements as constants of sort `Data` and predicates as constants of sort `Preds`. The sets of variables used by each predicate are described using equations. An equation will describe the initial configuration of the net.

For the example in Figure 1, we have the following constant definitions and equations (we only included the equations for transition `rja`):

```
ops i o p1 p2 p3 p4 p5 p6 : -> Places  .
ops cch ple pmf ra rja rmf scd srn :
                             -> Transitions .
ops c chr cd le rn f : -> Data .
ops okCH okData : -> Preds .
eq vars(okCH) = chr .
eq vars(odData) = chr , f .
eq read(rja) = f, chr .
eq write(rja) = rn .
eq delete(rja) = f,chr .
.....................
eq initConfig = <i, (c), ({ okCH : false },
               { okData : false }),empty > .
```

In what follows we will describe the behavior of WFD-nets using a rewrite theory in a system module of Maude. Every rewrite rule will describe the dynamic part of WFD-nets, i.e. when a transition `t` is enabled in a given configuration and the resulted configurations after the firing of `t`.

Let $\mathtt{depPreds}(\mathtt{t})$ be the set of predicates which depend on data elements that are written by `t`:

$$\mathtt{depPreds}(\mathtt{t})=\{\mathtt{P}\in\mathtt{Preds}|\ \mathtt{vars}(\mathtt{P})\cap\mathtt{Write}(\mathtt{t})\neq\emptyset\}$$

The firing of a transition t in a configuration can potentially produce a set of configurations:

$$\bigcup_{\mathtt{S}\in 2^{\mathtt{depPreds}(\mathtt{t})}}\{<\mathtt{M}',\mathtt{DS}',\sigma',\mathtt{DO}'>|\sigma'(\mathtt{P})=\neg\sigma(\mathtt{P}),\ \mathtt{P}\in\mathtt{S},\sigma'(\mathtt{P})=\sigma(\mathtt{P}),\mathtt{P}\notin\mathtt{S}\}$$

Each configuration in this set has the same marking, data elements and data operations, but can have different values for predicates. This set describes all the ways in which the write operations of `t` could influence the value of the dependent predicates. For each transition `t` a set of rewrite rules is introduced to describe all possible resulting configurations. The system module includes the functional module which describes the types and the static aspects of the WFD-net and also defines the following variables:

```
var M  : Marking .
var DS : Set{Data} .
var PS : Set{PredValue} .
var DO : Set{DataOp} .
```

For each $\mathtt{S}\in 2^{\mathtt{depPreds}(\mathtt{t})}$, $\mathtt{S}\neq\emptyset$ a conditional rewrite rule will be added:

$$\mathtt{crl}[\mathtt{label}]:\mathtt{C}_1 => \mathtt{C}_2\ \mathtt{if\ condition}$$

Such a rule has a unique label and will describe when transition `t` is enabled in configuration $\mathtt{C}_1$ and a resulting configuration $\mathtt{C}_2$, assuming that the write operations of `t` change the values of the predicates in set

Table 1: Rewrite rules corresponding to transition cch.

```
crl[cch1]: <M p1, (DS,c), PS, DO> =>   <M p3, (DS,c,chr), changeValues(PS, okData), dops(cch)>
                                       if definedPreds(okData, (DS,c,chr)).
crl[cch2]: <M p1, (DS, c), PS, DO> =>  <M p3, (DS,c,chr), changeValues(PS, okCH), dops(cch)>
                                       if definedPreds(okCH, (DS,c,chr)).
crl[cch3]: <M p1, (DS,c), PS, DO> =>   <M p3, (DS,c,chr), changeValues(PS, (okCH,okData)),
                                        dops(cch)> if definedPreds((okCH,okData), (DS,c,chr)).
rl[cch4]: <M p1, (DS,c), PS, DO> =>     <M p3, (DS,c,chr), PS, dops(cch)>
```

S. The left side of the rule is:

$$\mathtt{C}_1 = <\mathtt{M}\ \mathtt{inPlaces},\ \mathtt{DS},\ \mathtt{rData},\ \mathtt{prValues},\ \mathtt{DO}>$$

M, DS, DO are the variables defined in the module; inPlaces is the set of all the input places of t; $\mathtt{rData} = \mathtt{vars}(\mathtt{t})$ (the set of data elements read by t or involved in the guard of t); prValues is variable PS if the transition has no guard, otherwise the set $(\mathtt{PS}, \{\mathtt{P} : \mathtt{value}\})$, where P is the predicate in the guard of t and value is false if the guard is the negation of predicate P, otherwise true; $\mathtt{C}_2$ is:

$$\mathtt{C}_2 = <\mathtt{M}\ \mathtt{outPlaces}, (\mathtt{DS}, \mathtt{outData}),\ \mathtt{changeValues}(\mathtt{prValues}, S),\ \mathtt{dops}(\mathtt{t}) >$$

outPlaces is the set of all the output places of t and M outPlaces represents the new marking of the underlying WF-net; $\mathtt{outData} = \mathtt{Read}(\mathtt{t}) \cup \mathtt{Write}(\mathtt{t}) \setminus \mathtt{Delete}(\mathtt{t})$ (the set of data elements available after the operations in t have been performed); (DS, outData) represents the new set of defined data elements: the union between the set represented by the variable DS (the existing data elements before the transition fires) and the data elements produced by the execution of t; changeValues gives the new predicate values after the execution of the transition, it has as arguments the previous values, prValues, and the set of predicates that will change their value. dops(t) is the set of data operations performed by t.

The condition of the rule (condition) is definedPreds(S, (DS, outData)): all the predicates that could potentially change their value (S) must have all the dependent data defined in the new data configuration (the set (DS, outData)). Otherwise, the change of values would be useless (a predicate with undefined variables is considered false, because its transition cannot execute anyway due to missing data).

One additional un-conditional rewrite rule will describe the situation in which all the predicate values remain unchanged after the write operations of t are performed:

$$\mathtt{rl}[<\mathtt{label}>] : \mathtt{C}_1 => \mathtt{C}_2\ .$$

$\mathtt{C}_1$ is obtained as in the case of conditional rules. The right side of the rule, $\mathtt{C}_2$, is:

$$\mathtt{C}_2 = <\mathtt{M}\ \mathtt{outPlaces}, (\mathtt{DS}, \mathtt{outData}),\ \mathtt{prValues}, \mathtt{dops}(\mathtt{t}) >$$

The only difference from the conditional rules is that operator changeValues is not needed, as the predicate values are not changed by the execution of t.

In the WFD-net in Figure 1, $\mathtt{Preds} = \{\mathtt{okCH}, \mathtt{okData}\}$.

Transition ra (receive application) does not have any write operation, so it has one corresponding rewrite rule in Maude:

```
rl[ra] : < M i, (DS,c), PS, DO > =>
         < M p2 p1 , (DS,c), PS, dops(ra) > .
```

In order to obtain the conditional rules for transition cch (check credit history), we compute $\mathtt{modifiedPreds}(\mathtt{cch}) = \{\mathtt{okData}, \mathtt{okCH}\}$ (the values of these predicates could be changed by the write operation). All the rules have the same left side (see Table 1). The right side differs only in the arguments for changeValues and in the condition of the rule.

For instance, rule cch3 models the case in which the write operation changes the values of both okCH and okData. This rule applies only if both predicates have all the data elements defined in the resulting configuration.

In the example in Figure 1, the initial configuration is:

$$<\mathtt{i},\ \mathtt{c},\ (\{\mathtt{okCH} : \mathtt{false}\}, \{\mathtt{okData} : \mathtt{false}\}),\ \mathtt{empty}>$$

The rewrite rule rl[ra] can be applied: the term Mi can be matched with i (variable M in the rule will be matched to the empty set null) and i will be rewritten to p2 p1. (DS, c) will be matched to c (variable DS will be matched to the empty set, empty); PS in the left-side of the rule can be matched to the term $(\{\mathtt{okCH} : \mathtt{false}\}, \{\mathtt{okData} : \mathtt{false}\})$; DO in the rule can be matched with empty and this term will be rewritten to r(c) (representing the operations performed). After using with this rule, the resulting configuration is:

$$<\mathtt{p2}\ \mathtt{p1},\ \mathtt{c},\ (\{\mathtt{okCH} : \mathtt{false}\}, \{\mathtt{okData} : \mathtt{false}\}),\ \mathtt{r}(\mathtt{c})>$$

The rewrite with rule cch1 is not possible because definedPreds(okData, (DS, c, chr)) is false (data element f, on which okData depends, is not defined

in the data set (DS, c, chr), as DS is matched to the empty set). The same holds for rule cch3. The rewrite with rule cch2 changes the configuration into:

$$< \texttt{p2 p3},\ \texttt{c},\ (\{\texttt{okCH}:\texttt{true}\},\{\texttt{okData}:\texttt{false}\}),\ \texttt{r(c)},\texttt{w(chr)} >$$

The complete Maude modules for the example in Figure 1 can be found at:
https://github.com/cylonx/WFDtoMaudeExample.

5 VERIFICATION OF DATA FLOW ERRORS AND SOUNDNESS IN MAUDE

In this section we will formalize some common data-flow errors and the correctness properties using temporal logic (LTL and CTL temporal logic) (Clarke et al., 1999). The LTL properties can than be verified using the LTL Model Checker in Maude, while the CTL properties can be verified using the umaudemc utility for Maude (Rubio et al., 2021).

The LTL model checker in Maude allows to associate a Kripke structure to the rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$ specified by a Maude system module M, if one specifies the intended kind k of states in the signature Σ, and the relevant state predicates, that is, the relevant set AP of atomic propositions. In our case, we declare the intended states and the state predicates in a separate system module which protects the functional module where the data types and the structure of net are specified and includes the Maude predefined module SATISFACTION. For the example in Figure 1, this module will be called wfd-preds.

The subsort declaration subsort Config < State ensures that the states of the Kripke structure are the configurations described by the sort Config. The state predicates are declared as operators having the sort Prop (a predefined sort in the module SATISFACTION). Module wfd-preds has the following structure:

```
mod wfd-preds is
protecting wfd-nettypes .
including SATISFACTION .
subsort Config < State .

ops enabledTrans predTrans : Transitions
                             -> Prop .
op concurrentlyFireable : Transitions
                          Transitions
                        -> Prop .
ops final availableData : Set{Data} -> Prop .
ops read deleted written : Data -> Prop .

var M : Marking .
vars DS DS1 : DataSet .
var PS : PredSet .
var DO : DataOps .
var C : Config .
var P : Prop .
var x : Data .
--- equations
endm
```

The semantics of the state predicates is defined using equations eq $\texttt{config} \models \texttt{proposition} = \texttt{true}$.

Given a set of data elements DS, the proposition final(DS) is satisfied in any configuration in which the final marking is the final marking of the WF-net, o, and the set of defined data elements includes DS. This is described in Maude using the equation:

```
eq <o, (DS1,DS), PS, DO> |= final(DS) = true .
```

If t is a term of sort Transitions (representing a transition t of the WFD-net), enabledTrans(t) is satisfied in a configuration, if t is enabled at the marking of the net, i.e. the input places of the t belong to the multiset representing the marking:

```
eq < M input_places , DS , PS , DO > |=
   enabledTrans(t) = true .
```

For instance, in the example in Figure 1, the equation for the state predicate corresponding to transition rja (reject application) is:

```
eq < M p4 p6 , DS , PS , DO > |=
   enabledTrans(rja) = true .
```

If t is a term of sort Transitions and represents a transition which has in its guard a predicate P, then predTrans(t) is satisfied in a configuration, if the guard of t is true in that configuration:

```
eq <M, DS, (PS,{P : value}), DO> |=
   predTrans(t) = true .
```

P is a constant (representing an actual predicate in a guard for the given net). Term (PS, {P : value}) means that predicate P must appear with the value value in the configuration (PS represents the set of predicate values for the other predicates). value is false if the predicate P appears negated in the guard and true otherwise. For example in Figure 1, the equation for the state predicate predTrans(rja), corresponding to transition rja (reject application) is:

```
eq < M , DS , (PS , {okData : false}) , DO >
   |= predTrans(rja) = true .
```

If transition t has no guard, the following equation is used (C is a variable of sort Config):

```
eq C |= predTrans(t) = true .
```

If DS is a set of data elements, the proposition availableData(DS) is true in a configuration, if DS is included in the set of defined/available data elements in the configuration:

```
eq < M , (DS1,DS) , PS , DO > |=
   availableData (DS) = true .
```

If t1 and t2 are two transitions of the net, concurrentlyFireable(t1,t2) is true in a configuration if: t1 and t2 are concurrently enabled at the marking of the net (i.e. the multisets of input places of the transitions are included in the current marking of the configuration), all data elements read by both t1 and t2 are available (i.e. they are included in the set of defined data elements of the configuration and the guards of both transitions are true:

```
eq <M input_places, (DS,read_data),
   (PS,predicates), DO>
   |= concurrentlyFireable(t1,t2) = true .
```

M input_places is the current marking in the configuration, where input_places represents the multiset containing the input places of the transitions; (DS, read_data) is the set of defined data elements in the configuration and read_data is the set of data elements read by the two transitions; predicates is the set of predicate values that make the guards of both the transitions true.

For instance, in order to describe that transitions pmf and scd are concurrently fireable in a certain configuration, we use the equation:

```
eq <M p3 p4 p6, (DS,chr,f,c),
   (PS,{okData : true},{okCH : true}), DO>
   |= concurrentlyFireable(pmf,scd) = true .
```

The proposition written(x) is satisfied in any configuration in which a write operation has been performed (i.e. the set of operations contains w(x)):

```
eq <M, DS, PS, (DO,w(x))>
   |= written(x) = true.
```

Similar equations are introduced for read(x) and delete(x).

We will define a system module (for the example in Figure 1, it will be called wfd-check), in which we introduce the formulas that describe the data errors we need to check. This formulas are operators with the sort Prop.

```
mod wfd-check is
including wfd-preds .
including MODEL-CHECKER .
including LTL-SIMPLIFIER .
op enabledWithData : Transitions Data
                   -> Prop .
ops notmissing notredundant notoverwritten
consistentData delwrite access : Data
                              -> Prop .
ops fireable : Transitions -> Prop .
ops end : -> Prop .
var t : Transitions .
var x : Data .
var s: Set{Data}
eq enabledWithData(t,s) = enabledTrans(t) /\
   predTrans(t) /\ availableData (s) .
eq fireable (t) = enabledWithData(t,read(t)) .
.....
```

Proposition fireable(t) is satisfied in a configuration iff transition t is fireable in that configuration.

Next we will present and formalize the most common data flow errors that can occur in workflows with data, even if the underlying WF-net is sound. These data-flow errors are also called anti-patterns (Trčka et al., 2009).

The data flow errors can be described using formulas in CTL* temporal logic, whose negations are formulas in the LTL logic. The temporal operator F is denoted by <> in Maude notation, G is denoted by [], $\neg$ by $\sim$, the next operator (X) is denoted by O.

The **missing data** anti-pattern describes the situation where some data element needs to be accessed (used in the guard, read or deleted), but either it has never been written or it has been deleted without having been written again. Let Data(t) be the data elements read, deleted or used in a guard by t. A configuration in which a data element d needs to be accessed is a configuration $<\mathtt{M},\mathtt{DS},\mathtt{PS},\mathtt{DO}>$ such that there exists a transition t with $\mathtt{d} \in \mathtt{Data}(\mathtt{t})$ and $\mathtt{M}[\mathtt{t}\rangle$. If $\mathtt{d} \notin \mathtt{DS}$, then we have a missing data error.

Using the atomic proposition enabledTrans(t), we define formula access(d), which is satisfied in any configuration in which the data element d needs to be accessed: $\mathtt{access}(\mathtt{d}) = \bigvee_{\{\mathtt{t}|\mathtt{d}\in\mathtt{Data}(\mathtt{t})\}} \mathtt{enabledTrans}(\mathtt{t})$.

The following CTL* formula describes a missing data error:

$$\mathtt{EF}(\mathtt{access}(\mathtt{d}) \vee \neg\mathtt{availableData}(\mathtt{d}))$$

The negation of this formula is a LTL formula, which can be described in Maude as:

```
eq notmissing(d) =
[] ~(access(d) /\ ~availableData(d))
```

A data element is **lost** if there is an execution sequence in which, after the element is written, it is deleted or written again without being read first. In the example in Figure 1, if we remove the read operation on data element f for transition rmf, we would obtain a lost data error: in the execution sequence (ra, cch, pmf,rmf) the data element f is overwritten.

Let $\mathtt{delwrite}(\mathtt{d}) = \mathtt{deleted}(\mathtt{d}) \vee \mathtt{written}(\mathtt{d})$. In CTL*, the formula for lost data would be:

$$\mathtt{EF}(\mathtt{written}(\mathtt{d}) \wedge \neg\mathtt{end} \wedge \mathtt{X}(\neg\mathtt{read}(\mathtt{d})\,\mathtt{U}\,(\mathtt{delwrite}(\mathtt{d}) \wedge \neg\mathtt{read}(\mathtt{d}))))$$

Proposition end is satisfied when the workflow execution is completed.

The corresponding LTL formula for the negation of the CTL* formula above is described by the following equation in Maude:

```
eq notlost(x) = [] ~(written(x) /\ ~end /\
O (~read(x) U (delwrite(x) /\ ~read(x)))) .
```

The semantic of proposition end is given by the equation:

```
eq end = final(empty) .
```

A data element is **redundant** if there is some execution scenario in which it is written but never read before the workflow execution is completed. In the example in Figure 1, the data element le is redundant (after the firing of ple, no other transition reads this data element). This behavior does not always represent an error (in our example, le is not read by any other transition because it is a document for the client, it needs to appear in the final data of the workflow).

The CTL* formula for checking this property is:

$$\mathtt{EF}(\mathtt{written}(\mathtt{d}) \wedge \mathtt{X}(\neg\mathtt{read}(\mathtt{d})\mathtt{U}(\mathtt{final} \wedge \neg\mathtt{read}(\mathtt{d}))$$

The LTL formula describing the negation of this formula can be expressed by the following equation in Maude:

```
eq notredundant(x) = [] ~(written(x) /\
O (~read(x) U (end /\ ~read(x)))) .
```

Inconsistent data: A data element d is inconsistent if some transition t that writes or deletes d and some transition t′ that uses d (reads, has d in its guard, writes or deletes d) can be executed concurrently. In the example in Figure 1, all data elements are consistent. If we modify the example such that $\mathtt{Write}(\mathtt{ple}) = \{\mathtt{le}, \mathtt{chr}\}$, since transitions cch and ple can be executed concurrently, the data element le would be inconsistent.

Let Changing(d) be the set of transitions that write or delete d:
$\mathtt{Changing}(\mathtt{d}) = \{\mathtt{t} | \mathtt{d} \in \mathtt{Write}(\mathtt{t}) \cup \mathtt{Delete}(\mathtt{t})\}$

Let Using(d) be the set of transitions that write, delete or use d in a predicate:
$\mathtt{Using}(\mathtt{d}) = \mathtt{Changing}(\mathtt{d}) \cup \{\mathtt{t} | \mathtt{d} \in \mathtt{Vars}(\mathtt{t})\}$.

If $\mathtt{Changing}(\mathtt{d}) = \emptyset$, then d is consistent.

The CTL* formula for checking this property is:

$$\mathtt{EF}(\bigvee_{\substack{t_i \in Changing(d)\\ t_j \in Using(d)}} \mathtt{concurrentlyFireable}(\mathtt{t_i}, \mathtt{t_j}))$$

As in the previous case, we can obtain an LTL formula in Maude (consistentData(d)) for the negation of the CTL* formula above, by replacing EF with the Maude operators `[]~`.

The complete wfd-preds and wfd-check modules for the example in Figure 1 can be found at https://github.com/cylonx/WFDtoMaudeExample.

The LTL formulas can be checked directly in the Maude environment. For the example in Figure 1, checking if data element le is not redundant, lost and missing, can be done using the Maude LTL model checker, as in Figure 2.

In order to describe the correctness of workflows when both the control-flow and the data-flow are considered, we define two variants of soundness. The properties we propose in this paper extend the classical soundness defined in (Aalst, 1998) with information about data. A workflow with data is sound if: (1) there are no dead transitions: for any transition t, there exists a configuration reachable from the initial configuration such that t is fireable in that configuration ; (2) the proper termination condition holds.

We can check if a transition t is dead using the following LTL formula: `[] ~fireable(t)`.

We define two possible termination conditions: given an initial configuration and a set of final data elements, Df, from any configuration reachable form the initial configuration, a final configuration $< \mathtt{o}, \mathtt{DS}, \mathtt{PS}, \mathtt{DO} >$ can be reached and:

1. Termination with optional data:
   DS should contain at least one element from Df;
2. Termination with mandatory data:
   DS should contain all the elements from Df;

The two properties can be expressed in the CTL temporal logic with the following formulas:

1. $\mathtt{AGEF}(\bigvee_{\mathtt{d} \in \mathtt{Df}} \mathtt{final}(\mathtt{d}))$
2. $\mathtt{AGEF}(\bigwedge_{\mathtt{d} \in \mathtt{Df}} \mathtt{final}(\mathtt{d}))$

The CTL formulas can be checked using the unified Maude model-checking utility, umaudemc, which is a uniform command-line, graphical, and programming interface to different model checkers operating on Maude specifications.

For instance, for the example in Figure 1, the soundness with optional data, if the set of final data is $\mathtt{Df} = \{\mathtt{cd}, \mathtt{rn}\}$ can be expressed with the following formula in umaudemc:

```
A [] E <> final(cd) \/ final(rn)
```

Soundness with the set of optional data $\mathtt{Df} = \{\mathtt{cd}, \mathtt{rn}\}$ is satisfied: it is always possible to get to a configuration in which the marking is o and either the closing documents (cd) or the rejection note (rn) is defined.

The mandatory soundness with the data set $\mathtt{Df} = \{\mathtt{cd}\}$ does not hold for our example, but mandatory soundness holds with $\mathtt{Df} = \{\mathtt{le}, \mathtt{c}\}$ (the loan estimate is always provided and the initial client data is always available, as no action deleted it).

Our approach permits the verification of properties regarding data elements that cannot be specified directly in other approaches that do not model data explicitly (Trčka et al., 2009):

- a data element is defined in all possible executions of the workflow:
  `[] availableData(d)`

```
Maude> red in wfd-check : modelCheck(initConfig, notmissing(le)) .
rewrites: 646 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
Maude> red in wfd-check : modelCheck(initConfig, notlost(le)) .
rewrites: 699 in 10ms cpu (0ms real) (69900 rewrites/second)
result Bool: true
Maude> red in wfd-check : modelCheck(initConfig, notredundant(le)) .
rewrites: 166 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult: counterexample({< i,c,{okCH : false}, {okData : false},empty >,'ra-1}
{< p1 p2,c,{okCH : false}, {okData : false},r(c) >,'cch-1} {< p2 p3,chr, c,{okCH :true}, {okData : false},w(chr), r(c) >,'ple-1}
{< p3 p4,chr, le, c,{okCH : true}, {okData : false},w(le), r(c) >,'pmf-1}
{< p4 p5,chr, f, le, c,{okCH : true}, {okData : true},w(f), r(chr), r(c) >,'rmf-1}
{< p4 p6,chr, f, le, c,{okCH : true}, {okData : false},w(f), r(f) >,'rja-1},
{< o,le, c, rn,{okCH : true}, {okData : false},w(rn), r(chr), r(f), dl(chr), dl(f) >,deadlock})
```

Figure 2: Verification of data-flow errors in the Maude environment.

- a certain data element is never written during the execution of the workflow:
  `[] ~written(d)`
- a data element is deleted at least once - the negation of this property is:
  `[] ~deleted(d)`

6 RELATED WORK

There exist several approaches that study the correctness of the workflow process in the case when both data-flow and control-flow are considered, but do not focus on data-flow errors: Fan et al. (Fan et al., 2007) introduces dual workflow nets in order to model the workflow process with data and extends the notion of classical soundness defined in (Aalst, 1998) for these nets; In (Sidorova et al., 2011) workflow processses are modelled using Workflow Nets with Data. Two notions of soundness (may-soundness and must-soundness) are defined for these nets and hyper-transition systems are used to characterize these properties; In (Kheldoun et al., 2017), Kheldoun et al. transform BPMN models of business processes into to Recursive ECATNets (RECATNets) and use the Maude LTL model checker to check the proper termination of a process. The authors in (He et al., 2018) propose Workflow Nets with Data Constraints, in which data constraints are represented by propositional formulas, and verify the may-soundness and must-soundness for this model.

Most of the approaches that focus on detecting data-flow errors in workflows do not discuss a notion of correctness: in (Sun et al., 2006; Meda et al., 2007), the authors propose an extended UML notation for describing workflow processes and present verification algorithms for some of the data flow errors. Eshuis (Eshuis, 2006) proposes model checking techniques to detect violations of data consistency constraints of workflows modelled by UML activity diagrams. In (Liu et al., 2020; Xiang et al., 2021a), the authors propose extensions of Petri Nets for modelling workflow processes with data and introduce algorithms for detecting data-flow errors. Xiang et al. (Xiang et al., 2017) use unfolding techniques to detect data inconsistency in workflows modelled by workflow nets with data operations. None of these approaches allows the modelling of complex workflow processes in which the execution of tasks is conditioned by guards.

In (Xiang and Liu, 2020), the authors define a a guard-driven reachability graph (CRG) of a workflow net with data and use it to detect data-flow errors.

Zhao et al. (Zhao et al., 2022) detect data-flow errors by analyzing all the transition sequences in a state space of a WFD-net, but the influence of the write operations on the guards of transitions is not considered when computing the states.

In (von Stackelberg et al., 2014), the authors propose a tool that maps BPMN 2.0 process models into Petri nets and detect data-flow anti-patterns using model-checking techniques. Data deletion is not permitted in this approach. Also, in order to model the complex data usage scenarios in BPMN 2.0, the proposed approach produces very large Petri Net models, which will lead to the state explosion problem. In order to solve this problem, the authors in (Mülle et al., 2019) propose reduction techniques to obtain a reduced Petri net model. Although these techniques preserve the data-flow properties of the model, they can affect the study of the proper termination of the process, when both the control-flow and the data-flow are considered.

In (Trčka et al., 2009), WFD-nets are introduced in order to describe conceptual workflows with data. The authors describe the most common data-flow errors and the correctness property using CTL* formulas. The authors use the reachability graph of an unfolding of the WFD-net (which is also a Petri net) in order to obtain the Kripke structure used for model checking. Due to the size of this unfolding, the proposed technique suffers from the state space explosion

problem. No specific implementations exists for this approach. (Xiang et al., 2021b) proposes a tool which allows the detection of data inconsistency of workflow processes based on unfolding techniques similar to the ones in (Xiang et al., 2017). The authors use the guard-driven reachability graph from (Xiang and Liu, 2020) to apply model-checking techniques for detecting deadlocks and proper-termination, but no other data-flow errors are detected. Because the configuration graph does not take into consideration all the possible values of guards, incorrect results can be obtained regarding the proper termination.

Our approach uses the WFD-net proposed in (Trčka et al., 2009), allowing read, write, delete operations and guards for transitions. Instead of transforming a model of the process into Petri nets in order to apply model-checking techniques (as in (Trčka et al., 2009; von Stackelberg et al., 2014; Mülle et al., 2019)), we provide a formalization of workflows with data in Maude, by specifying WFD-nets as theories in the rewriting logic. This approach permits the specification of configurations as states of the Kripke structure that will be used by the LTL checker in Maude. The proposed configuration embeds all the data information needed to describe the data anti-patterns.

The existing approaches do not permit the specification of data in the initial state of the process or the specification of conditions on data when the workflow terminates, as we propose in this paper.

7 CONCLUSIONS

In this paper we have proposed the specification of workflow nets with data in Maude, providing a formal semantics for WFD-nets. We have introduced two notions of soundness to describe the correctness of workflows with data. Unlike other notions of correctness defined for workflows with data, in which the termination condition only requires that the final marking (o) should be reached, these properties require that at least one data element or all data elements from a given set should be present when the workflow terminates. We have formalized these correctness properties using the CTL temporal logic. We have also formalized the data-flow errors (redundant data, missing data, lost data and inconsistent data) using the LTL temporal logic. The verification of properties can be done using the LTL model checker from Maude and the `umaudemc` utility for Maude.

In ongoing work, we plan to identify and formalize other data flow errors, study soundness with other types of termination requirements and also consider the modelling of security constraints over data elements. We intend to develop a tool for editing WFD-nets, that will also automate the transformation from WFD-nets to Maude specifications and will permit the verification of properties.

REFERENCES

Aalst, W. M. P. (1998). The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 08(01):21–66.

Aalst, W. M. P., Hee, K. M., Hofstede, A. H. M., Sidorova, N., Verbeek, H. M. W., Voorhoeve, M., and Wynn, M. T. (2010). Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363.

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model checking*. MIT Press, London, Cambridge.

Eshuis, R. (2006). Symbolic model checking of UML activity diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38.

Fan, S., Dou, W., and Chen, J. (2007). Dual Workflow Nets: Mixed control/data-flow representation for workflow modeling and verification. In *Advances in Web and Network Technologies, and Information Management*, pages 433–444, Berlin, Heidelberg. Springer Berlin Heidelberg.

He, Y., Liu, G., Xiang, D., Sun, J., Yan, C., and Jiang, C. (2018). Verifying the correctness of workflow systems based on workflow net with data constraints. *IEEE Access*, 6:11412–11423.

Kheldoun, A., Barkaoui, K., and Ioualalen, M. (2017). Formal verification of complex business processes based on high-level Petri nets. *Information Sciences*, 385-386:39–54.

Liu, C., Zeng, Q., Duan, H., Wang, L., Tan, J., Ren, C., and Yu, W. (2020). Petri net based data-flow error detection and correction strategy for business processes. *IEEE Access*, 8:43265–43276.

Meda, H. S., Sen, A. K., and Bagchi, A. (2007). Detecting data flow errors in workflows: A systematic graph traversal approach. In *Workshop on Information Technology & Systems (WITS-2007)*.

Meseguer, J. (1992). Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155.

Mülle, J., Tex, C., and Böhm, K. (2019). A practical data-flow verification scheme for business processes. *Information Systems*, 81:136–151.

Reisig, W. (1985). *Petri Nets: An Introduction*. Springer-Verlag, Berlin, Heidelberg.

Rubio, R., Martí-Oliet, N., Pita, I., and Verdejo, A. (2021). Strategies, model checking and branching-time properties in Maude. *Journal of Logical and Algebraic Methods in Programming*, 123:100700.

Sidorova, N., Stahl, C., and Trčka, N. (2011). Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. *Information Systems*, 36(7):1026–1043. Spe-

cial Issue: Advanced Information Systems Engineering (CAiSE'10).

Sun, S. X., Zhao, J. L., Nunamaker, J. F., and Sheng, O. R. L. (2006). Formulating the data-flow perspective for business process management. *Info. Sys. Research*, 17(4):374–391.

Trčka, N., van der Aalst, W. M. P., and Sidorova, N. (2009). Data-flow anti-patterns: Discovering data-flow errors in workflows. In van Eck, P., Gordijn, J., and Wieringa, R., editors, *Advanced Information Systems Engineering*, pages 425–439, Berlin, Heidelberg. Springer Berlin Heidelberg.

von Stackelberg, S., Putze, S., Mülle, J. A., and Böhm, K. (2014). Detecting data-flow errors in BPMN 2.0. *Open J. Inf. Syst.*, 1:1–19.

Xiang, D. and Liu, G. (2020). Checking data-flow errors based on the guard-driven reachability graph of wfd-net. *COMPUTING AND INFORMATICS*, 39(1-2):193–212.

Xiang, D., Liu, G., Yan, C., and Jiang, C. (2017). Detecting data inconsistency based on the unfolding technique of Petri nets. *IEEE Transactions on Industrial Informatics*, 13(6):2995–3005.

Xiang, D., Liu, G., Yan, C., and Jiang, C. (2021a). A guard-driven analysis approach of workflow net with data. *IEEE Transactions on Services Computing*, 14(6):1650–1661.

Xiang, D., Zhao, F., and Liu, Y. (2021b). DICER 2.0: A new model checker for data-flow errors of concurrent software systems. *Mathematics*, 9(9).

Zhao, F., Xiang, D., Liu, G., Jiang, C., and Zhu, H. (2022). Detecting and repairing data-flow errors in WFD-net systems. *Computer Modeling in Engineering & Sciences*, 131(3):1337–1363.

SCIENCE AND TECHNOLOGY PUBLICATIONS