How to Guarantee Compliance between Workflows and Product Lifecycles?

Zhaoxia Wang[1,2,3,4], Arthur H.M. ter Hofstede[5,6,7,⋆], Chun Ouyang[5,6], Moe Wynn[5,6], Jianmin Wang[1,2,3], and Xiaochen Zhu[1,2,3]

[1] School of Software, Tsinghua University, Beijing, China
{wang-cx06,zhu-xc10}@mails.tsinghua.edu.cn,jimwang@tsinghua.edu.cn
[2] Key Lab for Information System Security, Ministry of Education, Beijing, China
[3] National Laboratory for Information Science and Technology, Beijing, China
[4] Logistical Engineering University, Chongqing, China
[5] Queensland University of Technology, Brisbane, Australia
{a.terhofstede,c.ouyang,m.wynn}@qut.edu.au
[6] NICTA, Queensland Research Laboratory, Brisbane, Australia
[7] Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. Product Lifecycle Management (PLM) systems are widely used in the manufacturing industry. A core feature of such systems is providing support for versioning of product data. As workflow functionality is increasingly used in PLM systems the possibility emerges that versioning policies as encapsulated in process models are inconsistent with respect to their actual lifecycles. In this paper we define compliance of object versioning lifecycles with respect to process models and provide a solution to automatically checking whether compliance holds.

1 Introduction

Facing the manufacturing industry are challenges dealing with achieving customer satisfaction and staying ahead of the competition. To achieve a competitive edge innovative IT solutions can be leveraged, e.g. to facilitate collaboration and to improve product development as well as product improvement. Among such IT solutions are Product Lifecycle Management (PLM) systems [31] which play a pivotal role in managing product data in an electronic manner. The main functionality offered by such systems concerns the management of product data and the management of processes. Functionality for management of product data in a PLM tends to build on the functionality offered by Product Data Management (PDM) [30] systems, which preceeded PLM systems. Version control is at the core of product data management. According to [12], version control mechanisms are concerned with support for design reuse and concurrent design.

Version control mechanisms make a predefined set of object version operations available (e.g. *check in*, *check out*, *release*). The application of these version operations may change the state of the objects to which they are applied. These

⋆ Senior Visiting Scholar of Tsinghua University.

state changes may be subject to constraints, which are enforced by the version control mechanism. Design concurrency issues can thus be addressed by maintaining the state of objects and restricting the application of version operations on objects that are in a certain state as well as by maintaining access privileges for the various users of the system. For example, an object that has been checked-out by some user cannot be modified by other users of the system till it has been checked-in again.

Contemporary PLM systems typically use workflow technology to provide support for process management. Many common business processes in the manufacturing industry, e.g. in areas such as accounting, engineering design, product release, process planning, and production control, involve the use of object version operations. The use of these operations in the context of tasks is subject to access control restrictions. In addition to that, the order in which these operations may be used may be governed by a lifecycle model wherein it can be specified that certain operations can only be applied when the object is in a certain state. As ordering relations between version operations are also implicitly enforced by the ordering relations between tasks in the workflow model, *compliance* issues may rear their head. Specifically, on the process side, one can specify for tasks which version operations are permitted during the execution of these tasks and how they may progress the state of objects, while on the access control side the access privileges of users are stored and control is maintained over the order in which object version operations may be applied. The order of tasks may thus contradict the order in which version operations may be applied. Determining whether this is the case is nontrivial however, as task ordering relations can be complex.

The focus of this paper is to comprehensively address the issues involved in determining compliance between product data management and process management in PLM systems. First a formal definition of compliance between a versioning-aware workflow model on the one hand and object versioning lifecycles on the other hand is given (Section 2). Subsequently, an approach based on a well-known analysis technique from the field of workflow management for automatically determining compliance, is outlined, and the correctness of this approach is then proven (Section 3). Based upon the formal foundation, a tool that implements the proposed approach is presented (Section 4). This tool is used to apply the approach to a number of real-life models in order to provide some insight into its potential practical applicability (Section 5). Section 6 discusses the related work and Section 7 concludes the paper.

2 Fundamentals

In this section background information on version management and workflow management is provided in order to be able to precisely characterise the problem and its proposed solution.

2.1 Version Management

Version management (or version control) [39, 13, 51, 6] is widely used in the management of engineering data [49, 25, 48, 14]. Version management is concerned with maintaining different versions of objects and configurations and with providing support for operations on these versions. The scope of object version management is a single object, e.g. a specific car design, while configuration version management deals with the ways component designs can be combined to create more complex design artifacts. As many business processes supported by PLM systems are concerned with version operations on objects only, we will not discuss configuration version management any further.

In the field of engineering-data management, there may be multiple versions of a design object (see for example Figure 1(a)). A *version* of a design object represents a meaningful stage in its evolution. Traditionally, versions are classified as *revisions* or *variants*. Revisions are versions that are ordered sequentially in time and which represent improvements to or changing requirements with respect to earlier versions. Variants on the other hand are versions that may exist concurrently and represent design alternatives within the same revision.

A revision can be successfully completed when the design object is checked and approved by a designated authority. In this case a new revision number is created and the design object can be officially released. During the design process, design objects may be submitted to supervisors for approval. If there are errors, a new variant is created (with a new variant number) that requires further attention of a designer. The variants thus reflect the design history of an object design before its official release.

The evolution of versions of an object design can be represented by means of a graph. This is illustrated in Figure 1(b) and Figure 1(c), which show a linear version graph and a tree version graph respectively. The latter type of graph does not only show the history of the various revisions, as the former type of graph does, but also the various variants that were produced as part of these revisions.

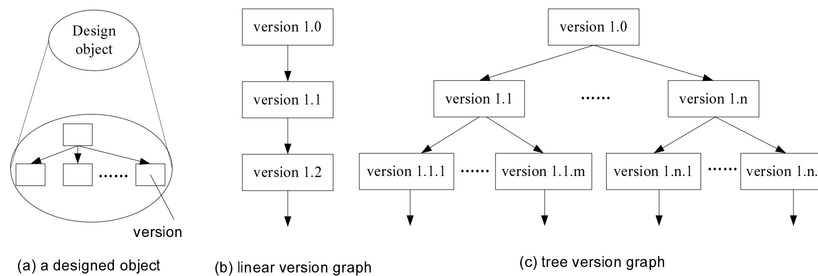


Fig. 1. A design object and two types of version graphs.

There are various operations that can be performed on versions of a design object. In [12] a number of operations on design objects and versions are distinguished, these operations include:

(1) Operations where versions are created, modified or deleted. Versions can be created afresh, through copying an already existing version, or by synthesising data present in a database and not used in any existing design object. Examples of such operations are *check-in*, *check-out*, *release*. A *check-in* operation leads to a new variant number, while a *release* operation leads to a new revision number.
(2) Operations where a version can be frozen (*freeze*) or thawed (*thaw*). A frozen version cannot be modified till it is (explicitly) thawed.

State transition diagrams are a technique used to capture versioning of design objects in engineering-data management [49]. Transitions reflect the possible applications of version operations and may change the state of a design object. When a design object is in a certain state, only those version operations can be applied that are linked to transitions that have this state as the source state. The application of a version operation associated with a certain transition takes the design object from the source state of this transition to its target state.

Through the use of a state transition diagram an ordering on the application of version operations can be imposed. For example, one can enforce that a *check-in* operation can only be applied to a design object that has been checked out or that an object can be released only when it has been checked in beforehand.

The following definition formalises the notion of a state transition diagram in the context of the versioning of design objects. In the remainder of the paper we will often abbreviate 'design object' to 'object'.

Definition 1 (Object versioning lifecycle). *An object versioning lifecycle* $\mathcal{L}$ *for an object is a finite state automaton* (S, V, δ, s_0, G) *where:*

- S *is a finite non-empty set of object states,*
- V *is the set of version operations,*
- $\delta : S \times V \rightarrow S$ *is the state transition function,*
- $s_0 \in S$ *is the initial state, and*
- $G \subseteq S$ *is the set of final (accepting) states.*

As mentioned before, examples of version operations are: *create*, *check-in*, *check-out*, *release*, *scrub*, *retain*, *delete*, *thaw*, *freeze*, etc. Also, we introduce some notations. If $\delta(s, v) = s'$, we write $s \xrightarrow{v} s'$. If $\sigma = v_1 v_2 ... v_n$ is a sequence of version operations that move the object from state s to s', we write $s \xrightarrow{\sigma} s'$.

2.2 Workflow Management

In workflow management one is concerned with providing support for the execution of business processes. The correct application of workflow management may save time and money and may make it easier to demonstrate compliance with e.g. best practices or legislation. A workflow management system routes work, when it becomes available, to authorised resources (could be people but also software applications) for execution and provides these resources with the information required to perform this work (and keeps relevant information for

future tasks that results from the conduct of this work). For a comprehensive overview of workflow management, the reader is referred to [5].

In [2], Wil van der Aalst argued that Petri nets provide a suitable formalism for the modelling of workflows. He introduced a subclass of Petri nets, referred to as workflow nets, as well as the notion of *soundness* in order to determine correctness of workflow nets. In order to be make this paper self-contained we will provide formal definitions for these notions.

Definition 2 (Petri net [27]). *A Petri net N is a tuple (P, T, F) where P is a finite set of places, T is a finite set of transitions such that $P \cap T = \varnothing$, and $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation.*

For each node x, i.e. a place or a transition, its pre-set, denoted as $\bullet x$, is the set of nodes which are input of x, i.e. $\bullet x = \{y | (y, x) \in F\}$, while its post-set, $x \bullet$, is the set of nodes which are output of x, i.e. $x \bullet = \{y | (x, y) \in F\}$.

A marking of a Petri net is an assignment of tokens to its places. A marking represents a state of the net.

Definition 3 (Marking). *Let $N = (P, T, F)$ be a Petri net, a* marking *M of N is a function from its places to the set of natural numbers, i.e. $M : P \to \mathbb{N}$.*

A marking can be written as a linear combination of places, e.g. $M = 6p_1 + 0p_2 + 3p_3$. For convenience we only list places that contain tokens so marking M is written as $6p_1 + 3p_3$. By treating markings as multisets we can apply multiset operations such as union and difference. For a Petri net $N = (P, T, F)$ we can compare markings. A marking M is greater than a marking M', $M > M'$, iff for all $p \in P : M(p) > M'(p)$. In a similar vein one can define $M \geq M'$ and $M < M'$.

Definition 4 (Petri net with an initial marking). *A Petri net system $\mathcal{P}$ is a Petri net (P, T, F) with a designated initial marking M_0, $\mathcal{P} = (P, T, F, M_0)$.*

Transitions can change the marking of a Petri net if they are enabled.

Definition 5 (Enabled transition). *Let M be a marking of Petri net $N = (P, T, F)$, transition t is* enabled *in M, denoted $M[t>$, iff for every place $p \in \bullet t : M(p) > 0$, or more succinctly $\bullet t \geq M$.*

Definition 6 (Firing a transition). *Let M be a marking of Petri net $N = (P, T, F)$, and t a transition that is enabled in M, i.e. $M[t>$. Firing transition t yields a marking M' defined by: $M' = M - \bullet t + t \bullet$. We denote this as $M \xrightarrow{t} M'$.*

Let $\sigma = t_1 \ldots t_n$ be a sequence of transitions (not all transitions need to be different) and M be a marking in which t_1 is enabled. For all $1 \leq i < n$, firing t_i yields a marking M_i in which t_{i+1} is enabled, and firing t_n yields marking M', i.e. $M \xrightarrow{t_1} M_1 \ldots M_{n-1} \xrightarrow{t_n} M'$, then we write $M \xrightarrow{\sigma} M'$. We write $M \xrightarrow{*} M'$ to indicate that a transition sequence σ exists such that $M \xrightarrow{\sigma} M'$. For convenience, we allow σ to be the empty transition sequence, in which case $M = M'$.

Definition 7 (Reachable marking). *Let $N = (P, T, F, M_0)$ be a Petri net system. M is a reachable marking of N iff $M_0 \xrightarrow{*} M$.*

The reader is referred to [27, 1, 53] for an overview of Petri nets.

Definition 8 (Workflow net [1]). *A workflow net (WF-net) is a Petri net (P, T, F) which has a designated and unique source place i, i.e. $i \in P$ and $\bullet i = \varnothing$, and a designated and unique sink place o, i.e. $o \in P$ and $o\bullet = \varnothing$. It is a requirement that every place and transition is on path from source place i to sink place o.*

The notion of soundness formalises the notion of correctness of a workflow net with respect to control-flow. It was originally proposed in [1], and later refined in [44]. The Woflan tool [46] can check whether a workflow net is sound or not based on the refined soundness notion.

Definition 9 (Soundness, adapted from [44]). *A workflow net (P, T, F) is sound iff it satisfies all of the three following conditions:*

1. proper completion. *Every reachable marking, that marks o only marks o, i.e. for all M with $i \xrightarrow{*} M$, if $M \geq o$ then $M = o$.*
2. option to complete. *From every reachable marking a marking can be reached that marks o, i.e. for all M with $i \xrightarrow{*} M$, there exists a marking M', $M \xrightarrow{*} M'$, such that $M' \geq o$.*
3. no dead tasks. *For every transition there exists a reachable marking that enables it, i.e. for every transition $t \in T$ there exists a reachable marking M, i.e. $i \xrightarrow{*} M$, such that $M[t >$.*

2.3 Compliance: An Illustrative Example

In order to illustrate the issue of compliance between a process model and object lifecycles, we provide an example specified in the TiPLM system[8]. The TiPLM system is a kind of PLM system which is used in more than 100 large enterprises in Mainland China.

Figure 2 shows a process model for designing and reviewing engineering drawings specified in terms of the notation used by the TiPLM system. The key steps of this process include tasks *Design*, *Verify*, *Review*, *Approve*, and *Release*. After engineering drawings are designed, they are checked step by step. If a drawing does not pass a check, it needs to be modified by the *Design* task.

During the execution of the various tasks in the process model, certain object version operations may need to be performed. We first provide a brief overview of the lifecycle of an object.

If the object does not exist, the operation *create* is executed and the object is created. The object's state is then *Checked-out*. In the *Checked-out* state, a designer can generate 2D drawings or 3D models. Upon completion of this work, the object can be checked in by performing the *check-in* operation. The object is then checked into the database and it is in the state *Checked-in*. If a

[8] http://www.thit.com.cn/chanpinshijie/TiPLM.htm

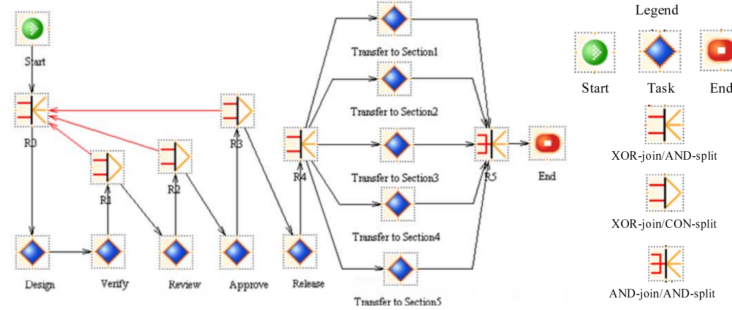


Fig. 2. Design and Review Process of Engineering Drawings.

subsequent task, such as *Verify* or *Approve*, rejects the current version of the design object, the object is checked out and its state changes from *Checked-in* to *Checked-out*. When the object is in this state, it can be modified by the designer. Upon completion of the changes, a new variant number is assigned automatically. When the task *Release* is executed the state of the object becomes *Released* and the revision number of the version is automatically increased. In that case, the object can be made available to other departments (sections 1 to 5 in the process model). The corresponding object lifecycle model is shown in Figure 3.

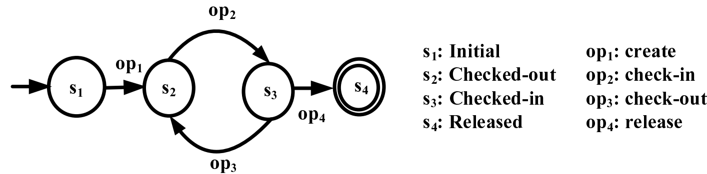


Fig. 3. A Sample Object Lifecycle Model.

The various tasks in the process model may have objectives that can be formulated in terms of the object lifecycles of the objects involved. For example, the task *Design* moves an object from its initial state to the state *Checked-in* (in case it did not yet exist) or from the state *Checked-in* through the state *Checked-out* back to the state *Checked-in* (in case it needed to be modified). Formally speaking, one can thus assign a set of state pairs to a task each reflecting a valid entry and exit state for the object for that task.

In contemporary PLM systems, object version management support through workflows cannot be viewed independently from access control considerations. Through the application of access control it is specified which tasks are allowed to perform which operations on data (and these include version operations). The assignment of version operations for our sample process model is shown in Table 1.

To further illustrate the setting of access control permissions consider two screenshots of the TiPLM system shown in Figure 4. The top screenshot shows property settings for the task *Design*, while the bottom screenshot shows how

Table 1. Access control privileges for tasks in process model of Figure 2

Task	Allowed object version operations
Design	create, check-in, check-out
Release	release
other tasks	–

access permissions can be set for various objects for this task. For an object, the permission setting can be inherited from its parent object (this is represented by the left column with tick boxes). A cross in the corresponding box means that the operation cannot be applied by the parent object, hence it is also forbidden for the child object. A tick in that box means that the operation can be applied by the parent object and hence also by the child object. The right most column of tick boxes can be used to override privileges set for parent objects. For the same object, putting ticks in both boxes or crosses in box boxes is in fact redundant.

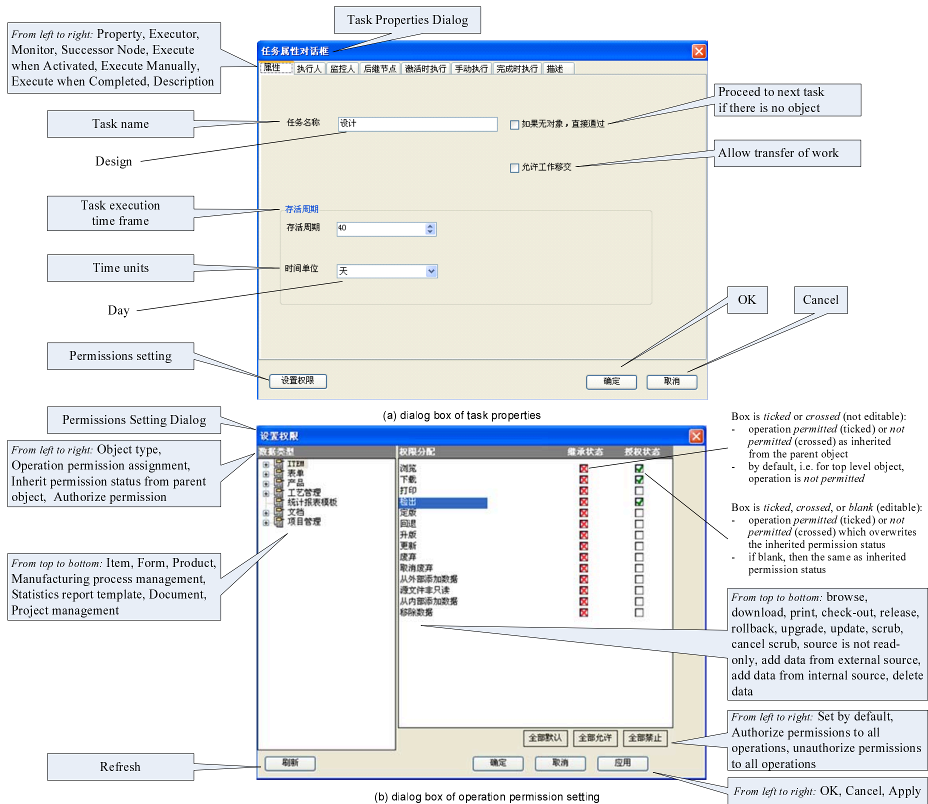


Fig. 4. Screenshots illustrating assignment of version operation privileges to tasks in the TiPLM system.

While the access control mechanism ensures data security by preventing the execution of unauthorised (version) operations during task execution, there is no consideration of the order in which version operations need to be performed. Hence in a PLM system we may have a situation where 1) a process model dictates the order in which tasks need to be executed, 2) the access control mechanism governs the use of version operations during the execution of tasks, and 3) an object versioning lifecycle controls when certain version operations can be applied. The amalgamation of these three models may lead to problems. Some of these problems may be of a syntactical nature, e.g. some version operations used in an object versioning lifecycle are not used by any task, while some problems may be of a semantical nature, e.g. the flow of version operations as prescribed by an object versioning lifecycle may contradict the possible sequences of version operations as can be derived from the ordering of tasks in the process model.

As an example of the latter case, consider the following assignment of version operations to tasks: task *Design* is allowed to perform operations *create* and *check-in*, tasks *Verify* and *Review* are allowed to perform operation *check-out*, and task *Release* is allowed to perform the *release* operation. In this case, the following scenario causes a problem. Imagine the task *Design* executes the version operations *create* and *check-in* taking the object to state *Checked-in*. The subsequent *Verify* task then executes the *check-out* operation and afterwards the object is in state *Checked-out*. If the next task is the task *Review* no version operation can be performed as the *check-out* operation expects the object to be in the state *Checked-in* and if this is followed by the task *Review* we are in fact in a deadlock as this task can only perform the *release* operation which also expects the task to be in the state *Checked-in* and no tasks following from this task can bring the object in this state.

Whether there are semantic problems due to contradictions in the specification of process models, access control privileges, and object lifecycle models may not be trivial to determine as a result of the fact that control-flow dependencies between tasks may be complex in nature. The topic of this paper is to examine what kinds of syntactic and semantic problems may occur in PLM systems when combining product data and process managament, to formally characterise these problems, and to provide a solution for automatically determining whether they are present in a specification.

3 Compliance Checking

In this section we propose a formal approach for conducting compliance checking between process models and object versioning lifecycles. We firstly define a *versioning-annotated process*. It specifies a process model, an object versioning lifecycle, and access control privileges, in a way that certain tasks in the process model are annotated with the object version operations prescribed in the access control privileges. We then formally characterise syntactical and semantical properties of versioning-annotated processes, and accordingly define the notions

of syntactical compatibility and behavioural compliance. These two notions are used to determine if a versioning-annotated process is compliant with an associated object versioning lifecycle. Finally, we present a solution for automatical reasoning about the compliance properties of versioning-annotated processes.

3.1 Versioning-Annotated Process

We define versioning-annotated processes in the form of *annotated* workflow nets. The definition captures *process specifications*, in the form of workflow nets (Definition 8), *object versioning lifecycles*, in the form of state transition diagrams (Definition 1), and *task annotations*, in the form of access privileges and object-state pairs.

Definition 10 (Versioning-annotated workflow net). *A versioning-annotated workflow net (VWF-net) is a tuple* $(\mathcal{P}, \mathcal{L}, R, Q)$ *where:*

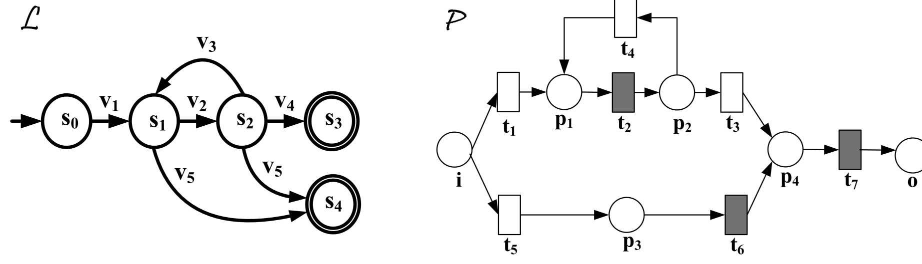
- $\mathcal{P} = (P, T, F)$ *is a process model specified as a workflow net,*
- $\mathcal{L} = (S, V, \delta, s_0, G)$ *is an object versioning lifecycle,*
- $R : T \rightarrow 2^V$ *is a function assigning version operations to process tasks,*
- $Q : T \rightarrow 2^{S \times S}$ *is a function assigning object-state pairs to process tasks, which specifies that for each state pair* $(s, s') \in Q(t)$ *the object can move from* s *to* s'.

For each $(s, s') \in Q(t)$, s is referred to as a *pre-state* of t, i.e. state of the object before task t is carried out, and s' a *post-state* of t, i.e. state of the object after task t is carried out. The set of object-state pairs associated with a task may be derived from its objective(s), which, though not formally stated, may be determined on the basis of its name and its input and output data.

Remark 1. The information about an *object* that a task can work on, and the set of *version operations* that the task can perform on the object, can be obtained from the access control settings specified for that task. Therefore this information is readily available.

Remark 2. In PLM systems (e.g. TiPLM) a task is allowed to work on *multiple* objects. As there is *no interference* between these objects, we can perform the compliance checking of a versioning-annotated process taking into account each object *in isolation*. Therefore, for compliance checking purposes, it is valid to consider just one object in Definition 10.

Example. Figure 5 depicts an illustrative example of a VWF-net which is abstracted from a design and review process of engineering drawings mentioned in the previous subsection. Note that in object versioning lifecycle $\mathcal{L}$ version operation v_5 labels two state transitions which can be uniquely identified as (s_1, v_5, s_4) and (s_2, v_5, s_4). Also, in process model $\mathcal{P}$ transitions t_1, t_3, t_4, t_5 have empty annotations (i.e. they are annotated with an empty set of version operations and an empty set of object-state pairs). These transitions model those tasks that do not perform any version operations on the object involved in the corresponding versioning-annotated process definition.



Task annotations:

task/s	version operations (R)	state pairs (Q)
t_1, t_3, t_4, t_5	$\varnothing$	$\varnothing$
t_2	$\{v_1, v_2\}$	$\{(s_0, s_2)\}$
t_6	$\{v_2, v_3\}$	$\{(s_2, s_2)\}$
t_7	$\{v_4, v_5\}$	$\{(s_2, s_3), (s_2, s_4)\}$

Fig. 5. An illustrative example of a VWF-net.

3.2 Syntactical Compatibility

A versioning-annotated process (VW) is the result of combining a business process ($\mathcal{P}$), an object versioning lifecycle ($\mathcal{L}$), and a version operations and object-state pairs assignment. The amalgamation of these components may lead to problems. We observed that some of these problems are due to the particular assignment of version operations as captured in versioning annotations. For example, assume there exists a version operation (v) in object versioning lifecycle $\mathcal{L}$ such that an object cannot reach any final state in $\mathcal{L}$ without carrying out v. The versioning-annotated process VW will deadlock if v is not assigned to any task in business process $\mathcal{P}$ regardless of the process behaviour. These problems are independent of the dynamic behaviour (or semantics) of a process, and thus are of a syntactical nature. Below we characterise the possible syntactical problems in versioning-annotated processes.

Definition 11 (Compatible versioning annotation). *Let* $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ *be a VWF-net where* $\mathcal{L} = (S, V, \delta, s_0, G)$. VW *is* annotated *in a way* compatible *with object versioning lifecycle* $\mathcal{L}$ *if and only if it satisfies all of the six following conditions:*

- empty annotation consistency. *A task has an empty annotation if and only if it carries out no version operations, i.e. for all* $t \in T$, $R(t)=\varnothing \Leftrightarrow Q(t)=\varnothing$.
- version operation assignment completeness. *The tasks altogether should be able to perform all possible version operations in the object versioning lifecycle, i.e.* $V = \bigcup_{t \in T} R(t)$.

- local object path existence. *A task, by performing the assigned version operations, is able to move an object from its pre-state to post-state in all the assigned state pairs, i.e. for all* $t \in T$ *and for all* $(s, s') \in Q(t)$*, there is a sequence of version operations* $\sigma \in R(t)^*$ *such that* $s \xrightarrow{\sigma} s'$.
- no locally assigned dead version operation. *A task is able to perform, at least once, every assigned version operation, i.e. for all* $t \in T$ *and for all* $v \in R(t)$*, there exists a sequence of version operations* $\sigma \in R(t)^*$ *and an object-state pair* $(s, s') \in Q(t)$*, such that* $s \xrightarrow{\sigma} s'$ *and* v *is part of sequence* σ.
- no dead object state transition. *Any transition in the object versioning lifecycle is possible in the context of a certain task, i.e. for all* $s, s' \in S$ *and* $v \in V$ *such that* $s \xrightarrow{v} s'$*, there exists* $t \in T$ *with* $v \in R(t)$*, a sequence of version operations* $\sigma \in R(t)^*$*, and an object-state pair* $(s, s') \in Q(t)$*, such that* $s \xrightarrow{\sigma} s'$ *and* v *is part of sequence* σ.
- global object path existence. *The tasks altogether should be able to move the object from its initial state to one of the final states, i.e. let* $(S, \bigcup_{t \in T} Q(t))$ *be a directed graph, there exists a path from* s_0 *to an* $s \in G$ *in this graph.*

The six conditions are used to check syntactical compatibility of a versioning-annotated process. Note that the first, the third, the fourth and the fifth condition apply to individual tasks (i.e. local check), while the other two conditions apply to all the tasks in the process as a whole (i.e. global check). Again, the checking is based on the annotations of tasks only, that is, the behaviour of the tasks (e.g., the order of task executions) is not taken into account at this stage.

Example. We apply the above syntactical compatibility checking to the VWF-net shown in Figure 5. The result shows that the object state transition leading from s_1 to s_4 and labelled v_5 in object versioning lifecyle $\mathcal{L}$ can never be performed in the context of any of the tasks in process $\mathcal{P}$. The problem can be fixed by either 1) removing the dead object state transition from the object versioning lifecyle (if the problem is related to the lifecycle) or 2) adding a state pair that covers the dead object state transition to an appropriate task in the process model (if the problem is related to the version operations assignment to tasks in the process). In this example, we assume that the object versioning lifecycle $\mathcal{L}$ is wrong, and thus remove the dead object state transition (s_1, v_5, s_4) from $\mathcal{L}$. The revised VWF-net then holds a compatible versioning annotation.

3.3 Behavioural Compliance

We continue to examine semantical problems with a versioning-annotated process. As mentioned before, these problems may be due to the fact that, for example, the possible sequences of version operations that can be derived from the ordering of tasks in business process $\mathcal{P}$ contradicts the flow of version operations as prescribed by object versioning lifecycle $\mathcal{L}$. We first define *execution semantics* of a VWF-net based on the definition of execution semantics of a WF-net as given in Section 2.

Definition 12 (Marking of VWF-net). *Let $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ be a VWF-net, a* marking *of VW is a pair (M, s) where M is a marking of WF-net $\mathcal{P}$ and s is a state of the object in its versioning lifecycle model $\mathcal{L}$.*

Given that WF-net $\mathcal{P}$ has a unique source place i and a unique sink place o and $\mathcal{L} = (S, V, \delta, s_0, G)$, the initial marking of VW can be written as (i, s_0) and a final marking of VW can be written as (o, s_f) where $s_f \in G$.

Definition 13 (Enabled transition in VWF-net). *Let (M, s) be a marking of VWF-net $(\mathcal{P}, \mathcal{L}, R, Q)$, transition t is* enabled *in (M, s), denoted $(M, s)[t>$, if and only if the two following conditions are both fulfilled:*

(1) t is enabled in M, i.e. $M[t>$, and
(2) $Q(t) = \varnothing$ or otherwise there exists $s' \in S$ such that $(s, s') \in Q(t)$.

Definition 14 (Firing a transition in VWF-net). *Let (M, s) be a marking of VWF-net $(\mathcal{P}, \mathcal{L}, R, Q)$ and t an enabled transition in (M, s), i.e. $(M, s)[t>$. If firing t in M leads to M' in $\mathcal{P}$ (i.e. $M \xrightarrow{t} M'$), then firing (t, σ) in (M, s) leads to (M', s'), written $(M, s) \xrightarrow{(t,\sigma)} (M', s')$, where*

- *if $Q(t) = \varnothing$: $s = s'$ and $\sigma = null$,*
- *otherwise: $(s, s') \in Q(t)$, $\sigma \in R(t)^*$ and $s \xrightarrow{\sigma} s'$.*

Let (M, s) be a marking in which t_1 is enabled and firing (t_1, σ_1) leads to (M_1, s_1). For all $1 \leq i < n$, firing (t_i, σ_i) yields a marking (M_i, s_i) in which t_{i+1} is enabled, and firing (t_n, σ_n) yields a marking (M', s'). That is,

$$(M, s) \xrightarrow{(t_1,\sigma_1)} (M_1, s_1) \ldots (M_{n-1}, s_{n-1}) \xrightarrow{(t_n,\sigma_n)} (M', s')$$

where $t_i \in T$ and $\sigma_i \in R(t_i)^* \cup \{null\}$ $(i = 1, ..., n)$. We write $(M, s) \xrightarrow{*} (M', s')$ to indicate that there exists a firing sequence $\alpha = (t_1, \sigma_1) \ldots (t_n, \sigma_n)$ such that $(M, s) \xrightarrow{\alpha} (M', s')$. For convenience, we allow α to be an empty firing sequence, in which case $M = M'$ and $s = s'$.

Definition 15 (Reachable marking of VWF-net). *Let $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ be a VWF-net, i the source place of $\mathcal{P}$, and s_0 the initial state of $\mathcal{L}$ (i.e. (i, s_0) is the initial marking of VW). (M, s) is a reachable marking of VW if and only if $(i, s_0) \xrightarrow{*} (M, s)$. $\mathbb{M}_{vw} = \{(M, s) | (i, s_0) \xrightarrow{*} (M, s)\}$ is the set of reachable markings of VW.*

As inspired by the soundness properties of a WF-net (see Definition 9), we specify the behavioural compliance of a VWF-net $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ as that there are no deadlocks during the execution of VW (i.e. the net has the option to complete), there are no dead tasks and no task has unused versioning annotations in VW, and that once the sink place of the process WF-net $\mathcal{P}$ is marked there are no more tokens left in $\mathcal{P}$ regardless of the state of $\mathcal{L}$ (i.e. VW has proper completion).

Definition 16 (Behavioural compliance). *Let $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ be a VWF-net where $\mathcal{P} = (P, T, F)$ is a WF-net and $\mathcal{L} = (S, V, \delta, s_0, G)$ is an object versioning lifecycle. VW is* compliant with *$\mathcal{L}$ if and only if it satisfies all of the four following conditions:*

- proper completion. *For every reachable marking, if it marks o and a final state of $\mathcal{L}$, then it only marks o and that state of $\mathcal{L}$, that is, for all (M, s_f) with $s_f \in G$ such that $(i, s_0) \xrightarrow{*} (M, s_f)$, if $M \geq o$ then $M = o$.*
- option to complete. *From every reachable marking a marking can be reached that marks o and a final state of $\mathcal{L}$, that is, for all (M, s) with $(i, s_0) \xrightarrow{*} (M, s)$, a marking (M', s_f) with $(M, s) \xrightarrow{*} (M', s_f)$ exists, such that $M' \geq o$ and $s_f \in G$.*
- no dead tasks. *For every task in $\mathcal{P}$ there exists a reachable marking that enables it, that is, for all $t \in T$, there is a reachable marking (M, s), i.e. $(i, s_0) \xrightarrow{*} (M, s)$, such that $(M, s)[t>$.*
- no unused versioning annotations. *Every task t with a non-empty versioning annotation in $\mathcal{P}$ should satisfy:*
  (1) *for every state pair $(s, s') \in Q(t)$, there is a reachable marking (M, s), i.e. $(i, s_0) \xrightarrow{*} (M, s)$; and*
  (2) *for every version operation $v \in R(t)$, there is a sequence of version operations $\sigma \in R(t)^*$ and an object-state pair $(s, s') \in Q(t)$ such that $s \xrightarrow{\sigma} s'$ and v is part of sequence σ (i.e. the "no locally assigned dead version operation" condition in Definition 11).*

From Definition 10, we can see that a VWF-net (VW) extends a WF-net ($\mathcal{P}$) with an object versioning lifecycle model, and thus it follows that the behavioural compliance of VW is influenced to some extent by the soundness properties of $\mathcal{P}$. This relationship is explored in the following proposition.

Proposition 1. *Let $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ be a VWF-net, where $\mathcal{P} = (P, T, F)$ is a WF-net and $\mathcal{L} = (S, V, \delta, s_0, G)$ is an object versioning lifecycle.*

(a) *If $(i, s_0) \xrightarrow{*} (M, s)$ for some $s \in S$ in VW, then $i \xrightarrow{*} M$ in $\mathcal{P}$.*
(b) *If $\mathcal{P}$ does not have the* weak *option to complete, then VW does not have the option to complete. As defined in [52], $\mathcal{P}$ has the weak option to complete iff there is a reachable marking M with $M \geq o$.*
(c) *If $\mathcal{P}$ has a dead task, then VW has a dead task.*
(d) *If $\mathcal{P}$ has proper completion, then VW has proper completion.*

Proof. (a) In VW, since $(i, s_0) \xrightarrow{*} (M, s)$, there is a firing sequence (t_1, σ_1) ... (t_n, σ_n) such that $(i, s_0) \xrightarrow{(t_1, \sigma_1)} (M_1, s_1)$ ... $(M_{n-1}, s_{n-1}) \xrightarrow{(t_n, \sigma_n)} (M, s)$. Then, from Definitions 13 and 14, it follows that $i \xrightarrow{t_1} M_1$ ... $M_{n-1} \xrightarrow{t_n} M$, i.e. $i \xrightarrow{*} M$, in $\mathcal{P}$.

(b) By contradiction. Assume that VW has the option to complete. Let $s_f \in G$, for each $(M, s) \in \mathbb{M}_{VW}$, $(M, s) \xrightarrow{*} (M', s_f)$ and $M' \geq o$. As a result, $(i, s_0) \xrightarrow{*} (M', s_f)$, and from (a), it follows that $i \xrightarrow{*} M'$ in $\mathcal{P}$, i.e. $\mathcal{P}$ has the weak

option to complete. This contradicts the fact that $\mathcal{P}$ does not have the weak option to complete. Hence, the statement holds.

(c) In $\mathcal{P}$, let $t \in T$ be a dead task, then for all M with $i \xrightarrow{*} M$, t is not enabled in M. From (a), it follows that for each $(M, s) \in \mathbb{M}_{VW}$ in VW, we have $i \xrightarrow{*} M$ in $\mathcal{P}$. Then, from Definitions 13, it follows that t is not enabled in any reachable marking in VW, i.e. t is also a dead task in VW.

(d) By contradiction. Assume that VW does not have proper completion. In VW, let $s_f \in G$, there is a $(M, s_f) \in \mathbb{M}_{VW}$ such that $M > o$. From (a), it follows that $i \xrightarrow{*} M$ in $\mathcal{P}$, and given $M > o$, $\mathcal{P}$ does not have proper completion. This contradicts the fact that $\mathcal{P}$ has proper completion. Hence, the statement holds.

3.4 Versioning Compliance Checking

Compliance checking between a versioning-annotated process and its associated object versioning lifecycle comprises both the syntactical compatibility checking and the behavioural compliance checking. Let $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ be a VWF-net capturing a versioning-annotate process. Proposition 1 characterises how the soundness properties of the WF-net $\mathcal{P}$ influence the compliance properties of the corresponding VWF-net VW. To be on the safe side, we assume that $\mathcal{P}$ is a sound WF-net before we conduct the compliance checking of VW. Also, in practice the WF-net $\mathcal{P}$ is always created separately from any object versioning lifecycle and before the VWF-net VW is constructed, and hence it is reasonable to conduct a soundness checking of $\mathcal{P}$ on its own.

Then, for the compliance checking of a versioning-annotated process, the syntactical compatibility checking is conducted first, because most syntactical problems, if undetected, will lead to semantical problems. Once the process passes syntactical compatibility checking, it will then go through behavioural compliance checking. Syntactical checking is straightforward. We focus on behavioural compliance checking and propose an approach involving three steps: firstly, we convert a VWF-net to a (normal) WF-net; secondly, we conduct soundness verification on this WF-net; and finally, based on the soundness verification results of the WF-net, we reason about the behavioural compliance between the original VWF-net and the associated object versioning lifecyle. Below, we define well-formed VWF-nets for behavioural compliance checking.

Definition 17 (Well-formed VWF-net). *A VWF-net $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ is* well-formed *if and only if $\mathcal{P}$ is a sound WF-net and VW has a compatible versioning annotation.*

Model Transformation Let $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ be a well-formed VWF-net, we propose the following steps for transforming VW to a WF-net $\mathcal{W}_{VW}$:

1. *Transform $\mathcal{L}$ to a Petri net $N_{\mathcal{L}}$.* The basic idea is to convert object states in $\mathcal{L}$ into places, object state transitions (labelled with version operations) in $\mathcal{L}$ to transitions. The resulting Petri net $N_{\mathcal{L}}$ has one source place (which models the initial state in $\mathcal{L}$) and one or more sink places (which correspond

to the set of final states in $\mathcal{L}$).

2. *Refine $N_{\mathcal{L}}$ to $N'_{\mathcal{L}}$ based on the version operations annotation in VW.* When a version operation (v) is assigned to *multiple* tasks in process $\mathcal{P}$, each of these tasks should be able to carry out that version operation *independently.* To capture this requirement, we introduce in $N'_{\mathcal{L}}$ multiple copies of transitions that model the state transitions labelled with v, each dedicated to one of the tasks to which v is assigned.

3. *Transform $\mathcal{P}$ to a new WF-net $W_{\mathcal{P}}$ where each transition with a non-empty versioning annotation is split into a sequence of starting transition, executing place, and completing transition.* This is to facilitate the modelling of such a transition carrying out the assigned version operation(s) during its execution.

4. *Refine $W_{\mathcal{P}}$ to $W'_{\mathcal{P}}$ based on the object-state pairs annotation in VW.* When a task (t) is linked to *multiple* object-state pairs, it means that during an execution of t the object can move from a pre-state to a post-state in any, but *only one*, of these state pairs. To capture this requirement, we introduce in $W'_{\mathcal{P}}$ multiple copies of starting and completing transitions of task t to model respectively the individual pre-states and post-states in the object-state pairs assigned to task t.

5. *Combine $N'_{\mathcal{L}}$ in Step 2 and $W'_{\mathcal{P}}$ in Step 4 into one WF-net $\mathcal{W}_{VW}$.* The two nets are connected as follows. Firstly, the places (modelling object states) in $N'_{\mathcal{L}}$ are connected with the starting or completing transitions of tasks in $W'_{\mathcal{P}}$, capturing the fact that such a task can only be started (or completed) when the object is in a valid pre-state (or post-state). Secondly, the executing places of tasks in $W'_{\mathcal{P}}$ are connected with the transitions (modelling object state transitions) in $N'_{\mathcal{L}}$, capturing the fact that version operations are carried out during the execution of those tasks to which the operations are assigned. For both connections we use bi-directional arcs to capture the behaviour of checking if a condition holds (i.e. the object is in a valid state, or a task is being executed). Finally, we introduce a source place to connect $N'_{\mathcal{L}}$ and $W'_{\mathcal{P}}$ in the beginning, and a sink place to connect them at the end.

6. *Add mutex place to control the access to the object for carrying out version operations by only one task at a time.* The *mutex* place gets marked once the process is ready to start, and unmarked once the process is completed. For every task that has a non-empty annotation, the starting (x) transition takes away the token from the *mutex* place and the completing (y) transition releases the token to the *mutex* place.

Now we formalise the above rules for transforming a VWF-net to a WF-net. One thing that should be noted is in an object versioning lifecycle, multiple object state transitions may be labelled with the same version operation. For example, in the object versioning lifecycle in Figure 5 version operation v_5 labels

two object state transitions (s_1, v_5, s_4) and (s_2, v_5, s_4). Hence, when a version operation v appears in the annotation of a task t and v labels more than one object state transition, it is necessary to check if each of these state transitions (s, v, s') is possible in the context of task t. This is captured by predicate $\textsc{ValidStateTrans}(t, (s, v, s'))$ in the following definition.

Definition 18 (Transformation from VWF-net to WF-net). *Let* $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ *be a well-formed VWF-net where* $\mathcal{P} = (P, T, F)$ *is a WF-net and* $\mathcal{L} = (S, V, \delta, s_0, G)$ *is an object versioning lifecycle. Given the following notations:*

- $T_\varnothing = \{t \in T | R(t) = \varnothing\}$ *is the set of tasks with an empty versioning annotation (i.e. any* $t \in T \backslash T_\varnothing$ *has a non-empty versioning annotation),*
- *for any* $t \in T \backslash T_\varnothing$, $\mathsf{PrS}(t)=\{s \in S | \exists_{s' \in S}[(s, s') \in Q(t)]\}$ *is the set of pre-states and* $\mathsf{PoS}(t)=\{s \in S | \exists_{s' \in S}[(s', s) \in Q(t)]\}$ *the set of post-states in the object-state pairs assigned to t, and*
- *predicate* $\textsc{ValidStateTrans}(t, s, v, s')$ *holds if there is an object-state pair* $(s_1, s_2) \in Q(t)$ *and two sequences of version operations* $\sigma_1, \sigma_2 \in R(t)^*$ *such that* $s_1 \overset{\sigma_1}{\to} s \overset{v}{\to} s' \overset{\sigma_2}{\to} s_2$.

VWF-net VW *can be transformed to WF-net* $\mathcal{W}_{VW} = (\mathcal{S}, \mathcal{T}, \mathcal{F})$ *where* $\mathcal{W}$ *is the* transformation constructor *and:*

$\mathcal{S} = S \cup P \cup \{\mathsf{e}_t | t \in T \backslash T_\varnothing\} \cup \{\mathsf{i}_\mathcal{W}, \mathsf{o}_\mathcal{W}\} \cup \{\mu\}$

$\mathcal{T} = T_\varnothing \cup X \cup Y \cup Z \cup H \cup \{\mathsf{b}\}$ where

- $X = \{\mathsf{x}_t^s | t \in T \backslash T_\varnothing \wedge s \in \mathsf{PrS}(t)\}$,
- $Y = \{\mathsf{y}_t^s | t \in T \backslash T_\varnothing \wedge s \in \mathsf{PoS}(t)\}$,
- $Z = \{\mathsf{z}_{(s,v,s')}^t | t \in T \backslash T_\varnothing \wedge v \in R(t) \wedge \textsc{ValidStateTrans}(t, (s, v, s'))\}$,
- $H = \{\mathsf{h}_s | s \in G\}$

$$
\begin{aligned}
\mathcal{F} = \; & (F \cap (P \times T_\varnothing \cup T_\varnothing \times P)) \cup \\
& \{(p, \mathsf{x}_t^s) | t \in T \backslash T_\varnothing \wedge p \in \bullet t \wedge s \in \mathsf{PrS}(t)\} \cup \\
& \{(\mathsf{y}_t^s, p) | t \in T \backslash T_\varnothing \wedge p \in t \bullet \wedge s \in \mathsf{PoS}(t)\} \cup \\
& \{(\mathsf{x}_t^s, \mathsf{e}_t) | t \in T \backslash T_\varnothing \wedge s \in \mathsf{PrS}(t)\} \cup \{(\mathsf{e}_t, \mathsf{y}_t^s) | t \in T \backslash T_\varnothing \wedge s \in \mathsf{PoS}(t)\} \cup \\
& \{(s, \mathsf{x}_t^s) | t \in T \backslash T_\varnothing \wedge s \in \mathsf{PrS}(t)\} \cup \{(\mathsf{x}_t^s, s) | t \in T \backslash T_\varnothing \wedge s \in \mathsf{PrS}(t)\} \cup \\
& \{(s, \mathsf{y}_t^s) | t \in T \backslash T_\varnothing \wedge s \in \mathsf{PoS}(t)\} \cup \{(s, \mathsf{y}_t^s) | t \in T \backslash T_\varnothing \wedge s \in \mathsf{PoS}(t)\} \cup \\
& \{(\mathsf{e}_t, \mathsf{z}_{(s,v,s')}^t) | t \in T \backslash T_\varnothing \wedge v \in R(t) \wedge \textsc{ValidStateTrans}(t, s, v, s')\} \cup \\
& \{(\mathsf{z}_{(s,v,s')}^t, \mathsf{e}_t) | t \in T \backslash T_\varnothing \wedge v \in R(t) \wedge \textsc{ValidStateTrans}(t, s, v, s')\} \cup \\
& \{(s, \mathsf{z}_{(s,v,s')}^t) | t \in T \backslash T_\varnothing \wedge v \in R(t) \wedge \textsc{ValidStateTrans}(t, s, v, s')\} \cup \\
& \{(\mathsf{z}_{(s,v,s')}^t, s') | t \in T \backslash T_\varnothing \wedge v \in R(t) \wedge \textsc{ValidStateTrans}(t, s, v, s')\} \cup \\
& \{(\mu, \mathsf{x}_t^s) | t \in T \backslash T_\varnothing \wedge s \in \mathsf{PrS}(t)\} \cup \{(\mathsf{y}_t^s, \mu) | t \in T \backslash T_\varnothing \wedge s \in \mathsf{PoS}(t)\} \cup \\
& \{(\mathsf{i}_\mathcal{W}, \mathsf{b})\} \cup \{(\mathsf{b}, \mathsf{i})\} \cup \{(\mathsf{b}, s_0)\} \cup \{(\mathsf{b}, \mu)\} \cup \\
& \{(s, \mathsf{h}_s) | s \in G\} \cup \{(o, \mathsf{h}_s) | s \in G\} \cup \{(\mu, \mathsf{h}_s) | s \in G\} \cup \{(\mathsf{h}_s, \mathsf{o}_\mathcal{W}) | s \in G\}
\end{aligned}
$$

Example. We apply the above rules to transforming the VWF-net in Figure 5 to a WF-net. Note that to pass the syntactical compatibility checking, the object state transition (s_1, v_5, s_4) is removed from object versioning lifecycle $\mathcal{L}$ before the transformation. The resulting WF-net is shown in Figure 6.

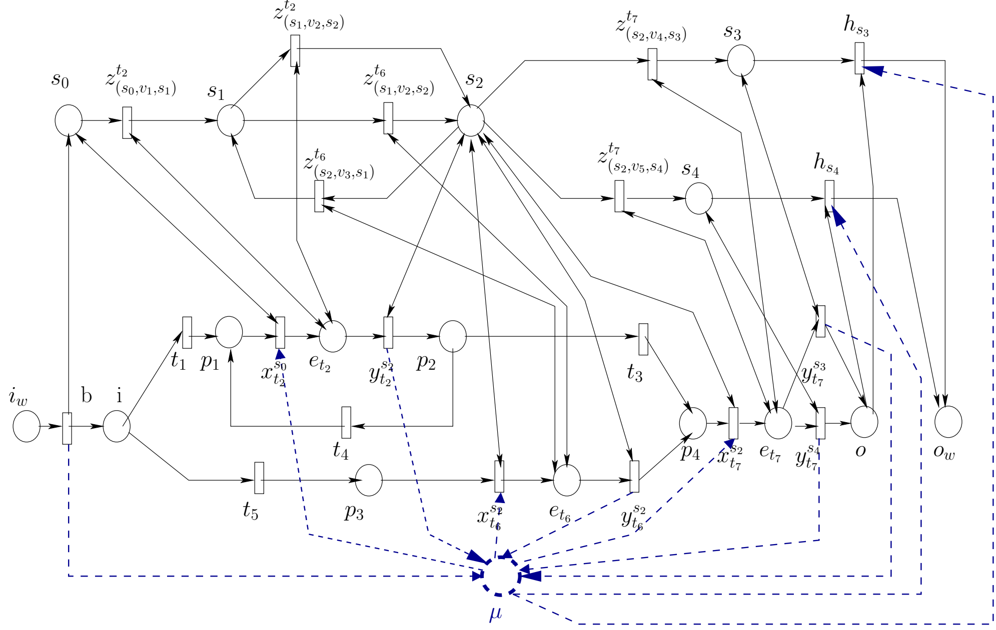


Fig. 6. A WF-net converted from a well-formed VWF-net as revised from the one in Figure 5 by removing the object state transition (s_1, v_5, s_4) from $\mathcal{L}$.

Soundness Verification and Reasoning Since a well-formed VWF-net comprises a sound WF-net and such a WF-net has proper completion, from Proposition 1(d), it follows that a well-formed VWF-net has proper completion.

Lemma 1. *A well-formed VWF-net has proper completion.*

We now focus on checking the other three behavioural compliance properties of a VWF-net: *option to complete*, *no dead tasks*, and *no unused versioning annotations*. Consider a well-formed VWF-net VW. We observe that the state space of VW is similar to that of $\mathcal{W}_{VW}$. Thus, we examine if it is possible to establish a bisimulation equivalence relation between these two state spaces. If we can establish such a relation, we can then derive more properties from that, e.g., the above behavioural compliance properties.

A bisimulation relation is etablished between labelled transition systems (LTSs). An LTS is a structure consisting of states with transitions, labelled with actions, between them. Thus, we will treat the state spaces of VW and $\mathcal{W}_{VW}$ as LTSs. The basic definitions of LTSs can be found in many references (e.g. [43, 42]). We adopt the one from [42] as it defines the set of final states which is necessary in specifying the notion of option to complete for an LTS.

Definition 19 (Labelled Transition System [42]). *A* Labelled Transition System (LTS) *is a 5-tuple* $(\mathcal{Q}, \mathcal{A}, \rightarrow, q_0, \Omega)$ *where*

– $\mathcal{Q}$ *is a set of* states*,*
– $\mathcal{A}$ *is a set of* action labels*,*
– $\rightarrow \subseteq (\mathcal{Q} \times (\mathcal{A} \cup \{\tau\}) \times \mathcal{Q})$ *is a* transition relation*, where* $\tau \notin \mathcal{A}$ *is the* silent action*,*
– $q_0 \in \mathcal{Q}$ *is the* initial state*, and*
– $\Omega \subseteq \mathcal{Q}$ *is the set of* final states*.*

Definition 20 (Semantical notations of an LTS, adapted from [43]). *Let* $L = (\mathcal{Q}, \mathcal{A}, \rightarrow, q_0, \Omega)$ *be an LTS. Let* $q, q' \in \mathcal{Q}$*, we write the following semantical notations for* L *:*

– L: $q \xrightarrow{\alpha} q'$*, if a transition labelled* $\alpha \in \mathcal{A} \cup \{\tau\}$ *changes the state of* L *from* q *to* q'*, i.e.* $(q, \alpha, q') \in\rightarrow$*.*
– L: $q \overset{\epsilon}{\Longrightarrow} q'$*, if there is a path from* q *to* q' *which consists of an arbitrary number* n *of* τ*-steps* $(n \geq 0)$*, i.e.* $q = q'$ *or* $\exists_{q'' \in \mathcal{Q}}[L\text{: } q \Rightarrow q'' \xrightarrow{\tau} q']$*.*
– *let* $\beta = \overline{a}_1 ... \overline{a}_n$ *(where* $\overline{a}_1, ..., \overline{a}_n \in \mathcal{A}$*) be a sequence of non-silent action/s, then* L: $q \overset{\beta}{\Longrightarrow} q'$*, which denotes* $q \Rightarrow q_1 \xrightarrow{\overline{a}_1} q'_1 \Rightarrow ... \Rightarrow q_n \xrightarrow{\overline{a}_n} q'_n \Rightarrow q'$ *(where* $q_1, q'_1, ..., q_n, q'_n \in \mathcal{Q}$*), that is, a path from* q *to* q' *passing through a sequence of actions that reduces to* β *after leaving out the silent ones.*

Definition 21 (Reachability in an LTS). *Let* $L = (\mathcal{Q}, \mathcal{A}, \rightarrow, q_0, \Omega)$ *be an LTS. For any* $q, q' \in \mathcal{Q}$*, if there exists a sequence of actions* $\psi \in \mathcal{A}^*$ *such that* L: $q \overset{\psi}{\Longrightarrow} q'$*, then* q' *is* reachable *from* q*, which can be denoted as* L: $q \xrightarrow{\overline{*}} q'$*. A state* $q \in \mathcal{Q}$ *is a* reachable state *if and only if* L: $q_0 \xrightarrow{\overline{*}} q$*.*

Definition 22 (Option to complete for an LTS). *An LTS* $L = (\mathcal{Q}, \mathcal{A}, \rightarrow, q_0, \Omega)$ *has the option to complete if and only if for every reachable state* $q \in \mathcal{Q}$*, there exists a final state* $q_f \in \Omega$ *such that* L: $q \xrightarrow{\overline{*}} q_f$*.*

Definition 23 (Dead action in an LTS). *Given an LTS* $L = (\mathcal{Q}, \mathcal{A}, \rightarrow, q_0, \Omega)$*,* $\overline{a} \in \mathcal{A}$ *is a dead action if and only if for any reachable state* $q \in \mathcal{Q}$*, there does not exist any state* $q' \in \mathcal{Q}$ *such that* L: $q \xrightarrow{\overline{a}} q'$*.*

Based on the above definitions of an LTS and its semantics, we now define the LTS (L_{VW}) of a well-formed VWF-net (VW). The LTS L_{VW} shows the internal steps between every two subsequent markings of the VWF-net VW and as a result it captures all the intermediate states. Our purpose is to determine the three compliance properties - *option to complete, no dead tasks, and no unused versioning annotations* - of the VWF-net VW through determining the behavioral properties, *option to complete and no dead actions*, of its labelled transition system L_{VW}.

Definition 24 (Predicate *POSSIBLE***).** *Let* $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ *be a well-formed VWF-net, where* $\mathcal{P} = (P, T, F)$ *is a WF-net and* $\mathcal{L} = (S, V, \delta, s_0, G)$ *is an object versioning lifecycle.* M *is a marking of* $\mathcal{P}$ *and* (i, s_0) *is the initial marking of* VW*. Then:*

– $\textsc{Possible}(M, t, s)$ *holds if 1)* $M[t>$ *and 2) an object-state pair* $(s_1, s_2) \in Q(t)$ *and two sequences of version operations* $\sigma_1, \sigma_2 \in R(t)^*$ *exist such that* $(i, s_0) \xrightarrow{*} (M, s_1)$ *and* $s_1 \xrightarrow{\sigma_1} s \xrightarrow{\sigma_2} s_2$.
– $\textsc{Possible}(M, t, s, s', v)$ *holds if 1)* $M[t>$*, 2)* $v \in R(t)$*, and 3) an object-state pair* $(s_1, s_2) \in Q(t)$ *and two sequences of version operations* $\sigma_1, \sigma_2 \in R(t)^*$ *exist such that* $(i, s_0) \xrightarrow{*} (M, s_1)$ *and* $s_1 \xrightarrow{\sigma_1} s \xrightarrow{v} s' \xrightarrow{\sigma_2} s_2$.

Definition 25 (LTS of VWF-net VW). *Let* $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ *be a well-formed VWF-net, where* $\mathcal{P} = (P, T, F)$ *is a WF-net and* $\mathcal{L} = (S, V, \delta, s_0, G)$ *is an object versioning lifecycle. The LTS of VWF-net* VW*, referred to as* L_{VW}*, is* $(\mathcal{Q}_{vw}, \mathcal{A}_{vw}, \rightarrow_{vw}, q_0^{vw}, \Omega_{vw})$ *where*

– $\mathcal{Q}_{vw} = \{(M, s) | (i, s_0) \xrightarrow{*} (M, s)\} \cup \{(\hat{M}\!\uparrow\! t, s) | t \in T \backslash T_\varnothing \wedge \textsc{Possible}(\hat{M} + \bullet t, t, s)\}$
– $\mathcal{A}_{vw} = \{\overline{t} | t \in T_\varnothing\} \cup \{\overline{(t, s)} | t \in T \backslash T_\varnothing \wedge s \in \mathsf{PrS}(t) \cup \mathsf{PoS}(t)\} \cup \{\overline{(t, v)} | t \in T \backslash T_\varnothing \wedge v \in R(t)\}$,
– $\rightarrow_{vw} = \{((M, s), \overline{t}, (M', s)) | t \in T_\varnothing \wedge M \xrightarrow{t} M'\} \cup$
$\{((\hat{M}\!\uparrow\! t, s), \overline{t'}, (\hat{M}'\!\uparrow\! t, s)) | t' \in T_\varnothing \wedge \hat{M} \xrightarrow{t'} \hat{M}'\} \cup$
$\{((M, s), \overline{(t, s)}, ((M - \bullet t)\!\uparrow\! t, s)) | t \in T \backslash T_\varnothing \wedge M[t> \wedge\, s \in \mathsf{PrS}(t)\} \cup$
$\{((\hat{M}\!\uparrow\! t, s), \overline{(t, v)}, (\hat{M}\!\uparrow\! t, s')) | t \in T \backslash T_\varnothing \wedge \textsc{Possible}(\hat{M} + \bullet t, t, s, s', v)\} \cup$
$\{((\hat{M}\!\uparrow\! t, s), \overline{(t, s)}, (\hat{M} + t\bullet, s)) | t \in T \backslash T_\varnothing \wedge (\hat{M} + \bullet t)[t> \wedge\, s \in \mathsf{PoS}(t)\}$
– $q_0^{vw} = (i, s_0)$, and
– $\Omega_{vw} = \{o\} \times G$.

Remark 3. For $t \in T \backslash T_\varnothing$, $\hat{M}\!\uparrow\! t$ represents an intermediate marking in which t is being executed, that is, after the tokens in the input places of t ($\bullet t$) being consumed and before the tokens in the output places of t ($t\bullet$) being produced.

Based on the above definition, the following proposition specifies the correspondence between a firing step in a well-formed VWF-net and a firing sequence in the LTS of the VWF-net. Recall that $\mathbb{M}_{vw}$ is the set of reachable markings of a VWF-net VW (see Definition 15).

Proposition 2. *Let* $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ *be a well-formed VWF-net, where* $\mathcal{P} = (P, T, F)$ *is a WF-net and* $L_{VW} = (\mathcal{Q}_{vw}, \mathcal{A}_{vw}, \rightarrow_{vw}, q_0^{vw}, \Omega_{vw})$ *be the LTS of* VW*. Then:*

(*a*) $\mathbb{M}_{vw} \cup \{(\hat{M}\!\uparrow\! t, s) | t \in T \backslash T_\varnothing \wedge \textsc{Possible}(\hat{M} + \bullet t, t, s)\} = \mathcal{Q}_{vw}$;
(*b*) *for any* $(M, s), (M', s') \in \mathbb{M}_{vw}$*,* $(M, s) \xrightarrow{(t, null)} (M', s')$ *in* VW *if and only if* L_{VW}: $(M, s) \xrightarrow{\overline{t}} (M', s')$ *(where* $s = s'$*), for some* $t \in T_\varnothing$ *where* $M \xrightarrow{t} M'$;
(*c*) *for any* $(M, s), (M', s') \in \mathbb{M}_{vw}$*,* $(M, s) \xrightarrow{(t, v_1 \ldots v_n)} (M', s')$ *in* VW *if and only if* L_{VW}: $(M, s) \xrightarrow{\overline{(t, s)}} ((M - \bullet t)\!\uparrow\! t, s) \xrightarrow{\overline{(t, v_1)}} ((M - \bullet t)\!\uparrow\! t, s_1) \ldots \xrightarrow{\overline{(t, v_n)}} ((M - \bullet t)\!\uparrow\! t, s') \xrightarrow{\overline{(t, s')}} (M', s')$*, for some* $t \in T \backslash T_\varnothing$ *where* $M \xrightarrow{t} M'$*,* $(s, s') \in Q(t)$*,* $v_1, \ldots, v_n \in R(t)$*, and* $s \xrightarrow{v_1} s_1 \ldots \xrightarrow{v_n} s'$*. By using a short-hand notation* $\overline{\omega}^t_{(s, \sigma, s')} =$

$(\overline{t,s})(\overline{t,v_1})...(\overline{t,v_n})(\overline{t,s'})$ *where* $\sigma = v_1...v_n$ *(i.e.* $\sigma \in R(t)^*$ *and* $s \xrightarrow{\sigma} s'$*), the above firing sequence in* L_{VW} *can be written as* L_{VW}: $(M,s) \xrightarrow{\overline{\omega}^t_{(s,\sigma,s')}} (M',s')$.

The following proposition specifies the correspondence between the state reachability of a well-formed VWF-net and that of the LTS of the VWF-net.

Proposition 3. *Let* $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ *be a well-formed VWF-net, where* $\mathcal{P} = (P,T,F)$ *is a WF-net, and* $L_{VW} = (\mathcal{Q}_{vw}, \mathcal{A}_{vw}, \rightarrow_{vw}, q_0^{vw}, \Omega_{vw})$ *be the LTS of* VW*. Then:*

(a) *For any* $(M,s),(M',s') \in \mathbb{M}_{vw}$, $(M,s) \xrightarrow{*} (M',s')$ *in* VW *if and only if* L_{VW}: $(M,s) \xrightarrow{\overline{*}} (M',s')$.

(b) *For any* $(\hat{M}{\uparrow}t, s_i) \in \mathcal{Q}_{vw} \backslash \mathbb{M}_{vw}$*, there exist* $(M,s),(M',s') \in \mathbb{M}_{vw}$ *such that* L_{VW}: $(M,s) \xrightarrow{\overline{*}} (\hat{M}{\uparrow}t, s_i)$ *and* L_{VW}: $(\hat{M}{\uparrow}t, s_i) \xrightarrow{\overline{*}} (M',s')$.

(c) *Every state* $q^{vw} \in \mathcal{Q}_{vw}$ *is a reachable state.*

Proof. (a) ($\Rightarrow$) In VW, given $(M,s),(M',s') \in \mathbb{M}_{vw}$, if $(M,s) \xrightarrow{*} (M',s')$, there is a firing sequence $(M,s) \xrightarrow{(t_1,\sigma_1)} (M_1,s_1) ... \xrightarrow{(t_n,\sigma_n)} (M',s')$. Let $(M_{i-1},s_{i-1}) \xrightarrow{(t_i,\sigma_i)} (M_i,s_i)$ be any firing step within the above sequence.

- If $t_i \in T_\varnothing$, then $\sigma_i = null$. Based on Proposition 2(b), there exists in L_{VW} a firing step L_{VW}: $(M_{i-1},s_{i-1}) \xrightarrow{\overline{t_i}} (M_i,s_i)$ (and $s_{i-1} = s_i$).
- If $t_i \in T \backslash T_\varnothing$, then $\sigma_i \in R(t_i)^*$ and $s_{i-1} \xrightarrow{\sigma_i} s_i$. Based on Proposition 2(c), there exists in L_{VW} a firing sequence L_{VW}: $(M_{i-1},s_{i-1}) \xrightarrow{\overline{\omega}^{t_i}_{(s_{i-1},\sigma_i,s_i)}} (M_i,s_i)$.

This means that every firing step in VW has a corresponding firing step/sequence in L_{VW}. Hence, if $(M,s) \xrightarrow{*} (M',s')$ in VW, then L_{VW}: $(M,s) \xrightarrow{\overline{*}} (M',s')$.

($\Leftarrow$) In L_{VW}, given $(M,s),(M',s') \in \mathbb{M}_{vw}$, if L_{VW}: $(M,s) \xrightarrow{\overline{*}} (M',s')$, there is a firing path L_{VW}: $(M,s) \xrightarrow{\delta_1} (M_1,s_1) ... \xrightarrow{\delta_n} (M',s')$, where 1) $M \xrightarrow{t_1} M_1 ... \xrightarrow{t_n} M'$ for some $t_1,...,t_n \in T$, and 2) let L_{VW}: $(M_{i-1},s_{i-1}) \xrightarrow{\delta_i} (M_i,s_i)$ be any firing step/sequence in the path, and $t_i \in \{t1,...,t_n\}$, then

$$\delta_i = \begin{cases} \overline{t_i} & (t_i \in T_\varnothing) \\ \overline{\omega}^{t_i}_{(s_{i-1},\sigma_i,s_i)} & (t_i \in T \backslash T_\varnothing, (s_{i-1},s_i) \in Q(t_i), \sigma_i \in R(t_i), s_{i-1} \xrightarrow{\sigma_i} s_i) \end{cases}$$

If L_{VW}: $(M_{i-1},s_{i-1}) \xrightarrow{\overline{t_i}} (M_i,s_i)$, then $(M_{i-1},s_{i-1}) \xrightarrow{(t_i,null)} (M_i,s_i)$ in VW (based on Proposition 2(b)). Otherwise, if L_{VW}: $(M_{i-1},s_{i-1}) \xrightarrow{\overline{\omega}^{t_i}_{(s_{i-1},\sigma_i,s_i)}} (M_i,s_i)$, then $(M_{i-1},s_{i-1}) \xrightarrow{(t_i,\sigma_i)} (M_i,s_i)$ in VW (based on Proposition 2(c)). This means that given the above firing path from (M,s) to (M',s') in L_{VW}, there exists a corresponding firing sequence from (M,s) to (M',s') in VW. Hence, if L_{VW}: $(M,s) \xrightarrow{\overline{*}} (M',s')$, then $(M,s) \xrightarrow{*} (M',s')$ in VW.

(b) For any $(\hat{M}{\uparrow}t, s_i) \in \mathcal{Q}_{vw} \backslash \mathbb{M}_{vw}$, based on the fact $\textsc{Possible}(\hat{M}{+}{\bullet}t, t, s_i)$, there is $(\hat{M}{+}{\bullet}t, s) \in \mathbb{M}_{vw}$ where $s \xrightarrow{\sigma} s_i \xrightarrow{\sigma'} s'$ for some $(s,s') \in Q(t)$ and

$\sigma, \sigma' \in R(t)^*$. Assume $\sigma = v_1...v_i$ and $\sigma' = v_{i+1}...v_n$. From Definition 25, it follows that L_{VW}: $(\hat{M}{+}\bullet t, s) \overset{\overline{(t,s)}}{\rightarrow} (\hat{M}{\uparrow}t, s) \overset{\overline{(t,v_1)}}{\rightarrow} ... \overset{\overline{(t,v_i)}}{\rightarrow} (\hat{M}{\uparrow}t, s_i) \overset{\overline{(t,v_{i+1})}}{\rightarrow} ... \overset{\overline{(t,v_n)}}{\rightarrow}$ $(\hat{M}{\uparrow}t, s') \overset{\overline{(t,s')}}{\rightarrow} (\hat{M}{+}t\bullet, s')$. Based on Proposition 2(c), we have $(\hat{M}{+}\bullet t, s) \overset{(t,\sigma\sigma')}{\longrightarrow}$ $(\hat{M}{+}t\bullet, s')$ in VW, and therefore $(\hat{M}{+}t\bullet, s') \in \mathbb{M}_{vw}$.

(c) This follows directly from (a) and (b).

The following lemma specifies how to reason about the property of *option to complete* for a well-formed VWF-net and its LTS.

Lemma 2. *Let $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ be a well-formed VWF-net, where $\mathcal{P} = (P, T, F)$ is a WF-net and $\mathcal{L} = (S, V, \delta, s_0, G)$ is an object versioning lifecycle, and $L_{VW} = (\mathcal{Q}_{vw}, \mathcal{A}_{vw}, \rightarrow_{vw}, q_0^{vw}, \Omega_{vw})$ be the LTS of VW. VW has the option to complete if and only if L_{VW} has the option to complete.*

Proof. ($\Rightarrow$) There are two cases in examining the states of L_{VW}.

(1) Let $(M, s) \in \mathbb{M}_{vw}$. Since VW has the option to complete, $(M, s) \overset{*}{\rightarrow} (o, s_f)$ holds for some $s_f \in G$ in VW. From Proposition 3(a), it follows that L_{VW}: $(M, s) \overset{\overline{*}}{\rightarrow} (o, s_f)$.
(2) Let $(\hat{M}{\uparrow}t, s_i) \in \mathcal{Q}_{vw} \backslash \mathbb{M}_{vw}$. From Proposition 3(c), L_{VW}: $(i, s_0) \overset{*}{\rightarrow} (\hat{M}{\uparrow}t, s_i)$. From Proposition 3(b), there is a $(M', s') \in \mathbb{M}_{vw}$ such that L_{VW}: $(\hat{M}{\uparrow}t, s_i) \overset{\overline{*}}{\rightarrow}$ (M', s'). Since $(M', s') \overset{*}{\rightarrow} (o, s_f)$ holds for some $s_f \in G$ in VW, from Proposition 3(a), it follows that L_{VW}: $(M', s') \overset{*}{\rightarrow} (o, s_f)$. Hence, L_{VW}: $(\hat{M}{\uparrow}t, s_i) \overset{\overline{*}}{\rightarrow} (o, s_f)$.

($\Leftarrow$) From Proposition 3(a), L_{VW}: $(i, s_0) \overset{\overline{*}}{\rightarrow} (M, s)$ for every $(M, s) \in \mathbb{M}_{vw}$. Since L_{VW} has the option to complete, L_{VW}: $(M, s) \overset{\overline{*}}{\rightarrow} (o, s_f)$ holds for some $s_f \in G$. Again, from Proposition 3(a), it follows that $(M, s) \overset{*}{\rightarrow} (o, s_f)$ in VW.

The following lemma specifies how to reason about the properties of *no dead tasks* and *no unused versioning annotations* for a well-formed VWF-net, and the property of *no dead actions* for the corresponding LTS.

Lemma 3. *Let $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ be a well-formed VWF-net, where $\mathcal{P} = (P, T, F)$ is a WF-net and $\mathcal{L} = (S, V, \delta, s_0, G)$ is an object versioning lifecycle, and $L_{VW} = (\mathcal{Q}_{vw}, \mathcal{A}_{vw}, \rightarrow_{vw}, q_0^{vw}, \Omega_{vw})$ be the LTS of VW. There are no dead tasks and no task has unused versioning annotations in VW if and only if there are no dead actions in L_{VW}.*

Proof. According to Definition 16, there are no dead tasks in VW if and only if for all $t \in T$, there is a $(M, s) \in \mathbb{M}_{vw}$ such that $(M, s)[t{>}$. No task has unused versioning annotations in VW if and only if for every $t \in T \backslash T_{\varnothing}$:

(1) for each $(s, s') \in Q(t)$, there is a $(M, s) \in \mathbb{M}_{vw}$ such that $(M, s)[t{>}$; and
(2) t has no locally assigned dead version operations.

Since VW is a well-formed VWF-net, it has a compatible versioning annotation, in which there are no locally assigned dead version operations to each task.

From Definition 23, there are no dead actions in L_{VW} if and only if for all $\overline{a} \in \mathcal{A}_{vw}$, there exist $(M, s), (M', s') \in \mathcal{Q}_{vw}$ such that L_{VW}: $(i, s_0) \xrightarrow{*} (M, s)$ and L_{VW}: $(M, s) \xrightarrow{\overline{a}} (M', s')$.

To facilitate the proof, we write the set of actions of L_{VW} (given in Definition 25) as follows:

$$\mathcal{A}_{vw} = \bigcup_{\mathbf{t}\in\mathbf{T}_\varnothing} \{\overline{t}\} \;\cup \bigcup_{\mathbf{t}\in\mathbf{T}\setminus\mathbf{T}_\varnothing} \Big(\bigcup_{(s,s')\in Q(t)} \{\overline{(t,s)}, \overline{(t,s')}\} \;\cup\; \{\overline{(t,v)} | v \in R(t)\} \Big)$$

($\Rightarrow$) We consider two cases in examining the actions in L_{VW}.

(1) For each $t \in T_\varnothing$ in VW, there is a $(M, s) \in \mathbb{M}_{vw}$ such that $M[t\rangle$, and t corresponds to action $\overline{t} \in \mathcal{A}_{vw}$. Let $M \xrightarrow{t} M'$, then $(M, s) \xrightarrow{(t,null)} (M', s)$ in VW. From Proposition 2(b), L_{VW}: $(M, s) \xrightarrow{\overline{t}} (M', s)$. Hence, for each $t \in T_\varnothing$, the corresponding action $\overline{t}$ can fire.

(2) For each $t \in T\setminus T_\varnothing$ in VW, there is $(M, s) \in \mathbb{M}_{vw}$ such that $(M, s)[t\rangle$ for each $(s, s') \in Q(t)$, and t corresponds to the set of actions:

$$\mathcal{A}^t_{vw} = \bigcup_{(s,s')\in Q(t)} \{\overline{(t,s)}, \overline{(t,s')}\} \;\cup\; \{\overline{(t,v)} | v \in R(t)\}$$

We use a short-hand notation $R_{(s,s')}(t)$ to represent the set of version operations $\{v_1, ...v_n\} \subseteq R(t)$ such that $s \xrightarrow{v_1} ... \xrightarrow{v_n} s'$ for $(s, s') \in Q(t)$. Let $M \xrightarrow{t} M'$, then $(M, s) \xrightarrow{(t,v_1...v_n)} (M', s')$ in VW. From Proposition 2(c), L_{VW}: $(M, s) \xrightarrow{\overline{(t,s)}} ((M - \bullet t)\uparrow t, s) \xrightarrow{\overline{(t,v_1)}} ... \xrightarrow{\overline{(t,v_n)}} ((M - \bullet t)\uparrow t, s') \xrightarrow{\overline{(t,s')}} (M', s')$, where all the actions $\{\overline{(t,s)}, \overline{(t,s')}\} \cup \{\overline{(t,v)} | v \in R_{(s,s')}(t)\}$ fire (in sequence). Then, for all $(s, s') \in Q(t)$, the following set of the actions can fire:

$$\hat{\mathcal{A}}^t_{vw} = \bigcup_{(s,s')\in Q(t)} \{\overline{(t,s)}, \overline{(t,s')}\} \;\cup \bigcup_{(s,s')\in Q(t)} \{\overline{(t,v)} | v \in R_{(s,s')}(t)\}$$

Given that VW has no locally assigned dead version operations, $R(t) = \bigcup_{(s,s')\in Q(t)} R_{(s,s')}(t)$, and thus $\mathcal{A}^t_{vw} = \hat{\mathcal{A}}^t_{vw}$. Hence, for each $t \in T\setminus T_\varnothing$, all the corresponding actions $\overline{a} \in \mathcal{A}^t_{vw}$ can fire.

($\Leftarrow$) By contradiction. Assume that VW has a dead task or a task in VW has unused versioning annotations. This leads to two cases.

(1) There is a dead task t, i.e. no $(M, s) \in \mathcal{Q}_{vw}$ exists such that $(M, s)[t\rangle$.
  (a) If $t \in T_\varnothing$, it is mapped to an action $\overline{t}$ in L_{VW}. Since L_{VW} does not have dead actions, there are $(M, s), (M', s) \in \mathcal{Q}_{vw}$ such that L_{VW}: $(M, s) \xrightarrow{\overline{t}} (M', s)$. From Proposition 2(b), we have $(M, s) \xrightarrow{(t,null)} (M', s)$ in VW.

(b) If $t \in T \backslash T_{\varnothing}$, it is mapped to the set of actions $\mathcal{A}^t_{vw}$. Since L_{VW} does not have dead actions, for each $(s, s') \in Q(t)$, there are $(M, s), (M', s') \in \mathcal{Q}_{vw}$ with $M \xrightarrow{t} M'$ such that L_{VW}: $(M, s) \overset{\overline{(t,s)}}{\rightarrow} ((M - \bullet t) \uparrow t, s) \overset{\overline{(t,v_1)}}{\rightarrow} ... \overset{\overline{(t,v_n)}}{\rightarrow} ((M - \bullet t) \uparrow t, s') \overset{\overline{(t,s')}}{\rightarrow} (M', s')$ where $v_1, ..., v_n \in R_{(s,s')}(t)$. Then, from Proposition 2(c), we have $(M, s) \overset{(t,v_1...v_n)}{\longrightarrow} (M', s')$ in VW.

From both (a) and (b), we can draw the conclusion that in VW for every $t \in T$, there is a $(M, s) \in \mathbb{M}_{vw}$ such that $(M, s)[t>$. This however contradicts the assumption statement. Hence, there are no dead tasks in VW if there are no dead actions in L_{VW}.

(2) A task $t \in T \backslash T_{\varnothing}$ has unused versioning annotations. Versioning annotations comprise state pairs and version operations. As a well-formed VWF-net, VW does not have locally assigned dead version operations. Thus, if task t has unused versioning annotations, they must include an unused state pair $(s, s') \in Q(t)$ such that (M, s) is not a reachable state, i.e. $(M, s) \notin \mathbb{M}_{vw}$. As stated in the above case (1)(b), since L_{VW} does not have dead actions, given task $t \in T \backslash T_{\varnothing}$ and state pair $(s, s') \in Q(t)$, we can reach the conclusion that there are $(M, s), (M', s') \in \mathcal{Q}_{vw}$ such that $(M, s) \overset{(t,v_1...v_n)}{\longrightarrow} (M', s')$ in VW. From Proposition 2(a), it follows that $(M, s) \in \mathbb{M}_{vw}$, which then contradicts the assumption that $(M, s) \notin \mathbb{M}_{vw}$. Hence, no task has unused versioning annotations in VW if there are no dead actions in L_{VW}.

Similarly, we now define the LTS ($L_{\mathcal{W}}$) of the WF-net ($\mathcal{W}_{VW}$) that is transformed from a well-formed VWF-net (VW). The LTS $L_{\mathcal{W}}$ captures the state space of the WF-net $\mathcal{W}_{VW}$. Our purpose is to reason about two soundness properties - *option to complete and no dead transitions* - of the WF-net $\mathcal{W}_{VW}$ through the two behavioural properties, *option to complete* and *no dead actions* of its labelled transition system $L_{\mathcal{W}}$.

Definition 26 (LTS of WF-net $\mathcal{W}_{VW}$). *Let $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ be a well-formed VWF-net where $\mathcal{P} = (P, T, F)$ is a WF-net. $\mathcal{W}_{VW} = (\mathcal{S}, \mathcal{T}, \mathcal{F})$ is the WF-net which results from the transformation of VW where $\mathcal{T} = T_{\varnothing} \cup X \cup Y \cup Z \cup H \cup \{\mathsf{b}\}$. $\mathcal{W}_{VW}$ has a unique source place $i_{\mathcal{W}}$ and a unique sink place $o_{\mathcal{W}}$. The LTS of WF-net $\mathcal{W}_{VW}$, referred to as $L_{\mathcal{W}}$, is $(\mathcal{Q}_w, \mathcal{A}_w, \rightarrow_w, q^w_0, \Omega_w)$ where*

- $\mathcal{Q}_w = \{M | i_{\mathcal{W}} \xrightarrow{*} M\}$,
- $\mathcal{A}_w = \{\bar{t} | t \in T_{\varnothing}\} \cup$
  $\{\overline{(t, s)} | t \in T \backslash T_{\varnothing} \wedge s \in \mathsf{PrS}(t) \cup \mathsf{PoS}(t)\} \cup$
  $\{\overline{(t, v)} | t \in T \backslash T_{\varnothing} \wedge v \in R(t)\}$,
- $\rightarrow_w = \{(M, \bar{t}, M') | t \in T_{\varnothing} \wedge M \xrightarrow{t} M'\} \cup$
  $\{(M, \overline{(t, s)}, M') | (M \xrightarrow{\mathsf{x}^s_t} M' \wedge \mathsf{x}^s_t \in X) \vee (M \xrightarrow{\mathsf{y}^s_t} M' \wedge \mathsf{y}^s_t \in Y)\} \cup$
  $\{(M, \overline{(t, v)}, M') | M \xrightarrow{\mathsf{z}^t_{(s,v,s')}} M' \wedge \mathsf{z}^t_{(s,v,s')} \in Z\} \cup$
  $\{(M, \tau, M') | M \xrightarrow{\mathsf{b}} M' \vee (M \xrightarrow{\mathsf{h}} M' \wedge \mathsf{h} \in H)\}$,

– $q_0^w = i_{\mathcal{W}}$, and
– $\Omega_w = \{o_{\mathcal{W}}\}$.

Lemma 4. *Let VW be a well-formed VWF-net. $\mathcal{W}_{VW}$ has the option to complete if and only if $L_{\mathcal{W}}$ has the option to complete.*

Proof. From Definition 26, it can be observed that the reachability graph of WF-net $\mathcal{W}_{VW}$ can be represented by the corresponding LTS $L_{\mathcal{W}}$ by ignoring the labels of actions. Hence, the lemma holds.

Lemma 5. *Let $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ be a well-formed VWF-net where $\mathcal{P} = (P, T, F)$ is a WF-net and $\mathcal{L} = (S, V, \delta, s_0, G)$ is an object versioning lifecycle. $\mathcal{W}_{VW} = (\mathcal{S}, \mathcal{T}, \mathcal{F})$, where $\mathcal{T} = T_\varnothing \cup X \cup Y \cup Z \cup H \cup \{\mathsf{b}\}$, which has $L_{\mathcal{W}} = (\mathcal{Q}_w, \mathcal{A}_w, \rightarrow_w, q_0^w, \Omega_w)$ as its associated labelled transition system. $\mathcal{W}_{VW}$ does not have dead transitions if and only if $L_{\mathcal{W}}$ does not have dead actions.*

Proof. From Definition 18 and Definition 26, we observe the correspondence between the transitions in $\mathcal{W}_{VW}$ and the actions in $L_{\mathcal{W}}$, and write the set of actions of $\mathcal{L}_{\mathcal{W}}$ as follows:

$$\mathcal{A}_w = \bigcup_{t \in T_\varnothing} \{\overline{t}\} \cup \bigcup_{\mathsf{x}_t^s \in X} \{\overline{(t,s)}\} \cup \bigcup_{\mathsf{y}_t^s \in Y} \{\overline{(t,s)}\} \cup \bigcup_{\mathsf{z}^t_{(s,v,s')} \in Z} \{\overline{(t,v)}\}.$$

Also, following observation from Definition 26, $L_{\mathcal{W}}$ represents the reachability graph of $\mathcal{W}_{VW}$ and labels each state transition with a corresponding action or treats it as a silent action.

($\Rightarrow$) All the actions $\overline{a} \in \mathcal{A}_w$ in $L_{\mathcal{W}}$ are used to label the corresponding transitions $\gamma \in \mathcal{T}$ in $\mathcal{W}_{VW}$. Hence, if each $\gamma \in \mathcal{T}$ in $\mathcal{W}_{VW}$ is not a dead transition, then each $\overline{a} \in \mathcal{A}_w$ in $L_{\mathcal{W}}$ is not a dead action.

($\Leftarrow$) All the transitions $\gamma \in T \backslash (H \cup \{b\})$ in $\mathcal{W}_{VW}$ are labelled with the corresponding actions $\overline{a} \in \mathcal{A}_w$ in $L_{\mathcal{W}}$. Hence, if each $\overline{a} \in \mathcal{A}_w$ in $L_{\mathcal{W}}$ is not a dead action, then each $\gamma \in T \backslash (H \cup \{b\})$ in $\mathcal{W}_{VW}$ is not a dead transition. Next, in $\mathcal{W}_{VW}$, transition b can always fire in the initial marking $i_{\mathcal{W}}$, i.e. it is not a dead transition. Now we consider H transitions. In $\mathcal{W}_{VW}$, for each $\mathsf{h} \in H$, there is a $s_f \in G$ such that $o + s_f + \mu \xrightarrow{\mathsf{h}} o_{\mathcal{W}}$. As a well-formed VWF-net, VW has compatible versioning annotations, and thus there are no dead object state transitions (see Definition 11). Hence, for all $(s_i, v, s_f) \in S \times V \times G$ with $s_i \xrightarrow{v} s_f$, there exist $t \in T$ with $v \in R(t)$, $s \in S$ with $(s, s_f) \in Q(t)$, and $\sigma \in R(t)^*$, such that $s \xrightarrow{\sigma} s_i \xrightarrow{v} s_f$. For each $(s, s_f) \in Q(t)$, there is a $y_t^{s_f}$ in $\mathcal{W}_{VW}$, and since there are no dead transitions in $\mathcal{W}_{VW}$, $y_t^{s_f}$ can fire resulting in marking $o + s_f + \mu$, in which transition h is then enabled. Hence, each $\mathsf{h} \in H$ is not a dead transition.

As mentioned before, we observe that the state space of a well-formed VWF-net VW is similar to that of the corresponding WF-net $\mathcal{W}_{VW}$. Thus, we hope to check whether there is a bisimulation relation between the LTS of VW (L_{VW}) and the LTS of $\mathcal{W}_{VW}$ ($L_{\mathcal{W}}$). Given the fact that $L_{\mathcal{W}}$ has silent transitions, it is logical to check whether a *weak* bisimulation relation can be established. The

concept of weak bisimulation was first proposed in [26], and later on refined in other references such as [43, 42]. Below, we adopt the definition of weak bisimulation from [43] and add the condition on the final states as given in [42].

Definition 27 (Weak bisimulation, adapted from [43, 42]). *Let $L = (\mathcal{Q}, \mathcal{A}, \rightarrow, q_0, \Omega)$ and $\hat{L} = (\hat{\mathcal{Q}}, \hat{\mathcal{A}}, \rightarrow', \hat{q}_0, \hat{\Omega})$ be two LTSs. The relation $\mathcal{R} \subseteq \mathcal{Q} \times \hat{\mathcal{Q}}$ is a weak simulation if and only if it satisfies the following three conditions:*

(1) $(q_0, \hat{q}_0) \in \mathcal{R}$.

(2) for all $q, q' \in \mathcal{Q}$ *and* $\hat{q} \in \hat{\mathcal{Q}}$:

- *if* $(q, \hat{q}) \in \mathcal{R}$ *and* L: $q \xrightarrow{\tau} q'$, *then* $(q', \hat{q}) \in \mathcal{R}$; *and*
- *if* $(q, \hat{q}) \in \mathcal{R}$ *and* L: $q \xrightarrow{\overline{a}} q'$ *for some* $\overline{a} \in \mathcal{A}$, *then there is a* $\hat{q} \in \hat{\mathcal{Q}}$ *such that* $\hat{L}$: $\hat{q} \stackrel{\overline{a}}{\Longrightarrow} \hat{q}'$ *and* $(q', \hat{q}') \in \mathcal{R}$.

(3) for all $q_f \in \Omega$, $\hat{q} \in \hat{\mathcal{Q}}$: *if* $(q_f, \hat{q}) \in \mathcal{R}$, *then there is a* $\hat{q}_f \in \hat{\Omega}$ *such that* $\hat{L}$: $\hat{q} \stackrel{\epsilon}{\Longrightarrow} \hat{q}_f$ *and* $(q_f, \hat{q}_f) \in \mathcal{R}$.

If both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak simulations, then $\mathcal{R}$ is a weak bisimulation, *and L and $\hat{L}$ are weakly bisimilar over $\mathcal{R}$, denoted by $L \simeq_{\mathcal{R}} \hat{L}$.*

The following proposition is derived from an equivalent definition to the above definition of weak bisimulation as given in [43].

Proposition 4 (derived from [43]). *Let $L = (\mathcal{Q}, \mathcal{A}, \rightarrow, q_0, \Omega)$ and $\hat{L} = (\hat{\mathcal{Q}}, \hat{\mathcal{A}}, \rightarrow', \hat{q}_0, \hat{\Omega})$ be two LTSs that hold a weak bisimulation relation $\mathcal{R} \subseteq \mathcal{Q} \times \hat{\mathcal{Q}}$. For all $q, q' \in \mathcal{Q}$ and $\hat{q} \in \hat{\mathcal{Q}}$, if $(q, \hat{q}) \in \mathcal{R}$ and L: $q \stackrel{\psi}{\Longrightarrow} q'$ for some $\psi \in \mathcal{A}^*$, then there is a $\hat{q}' \in \hat{\mathcal{Q}}$ such that $\hat{L}$: $\hat{q} \stackrel{\psi}{\Longrightarrow} \hat{q}'$ and $(q', \hat{q}') \in \mathcal{R}$. Symmetrically, for all $\hat{q}, \hat{q}' \in \hat{\mathcal{Q}}$ and $q \in \mathcal{Q}$, if $(\hat{q}, q) \in \mathcal{R}^{-1}$ and $\hat{L}$: $\hat{q} \stackrel{\psi}{\Longrightarrow} \hat{q}'$ for some $\psi \in \hat{\mathcal{A}}^*$, then there is a $q' \in \mathcal{Q}$ such that L: $q \stackrel{\psi}{\Longrightarrow} q'$ and $(\hat{q}', q') \in \mathcal{R}^{-1}$.*

Lemma 6. *Let $L = (\mathcal{Q}, \mathcal{A}, \rightarrow, q_0, \Omega)$ and $\hat{L} = (\hat{\mathcal{Q}}, \hat{\mathcal{A}}, \rightarrow', \hat{q}_0, \hat{\Omega})$ be two LTSs and $L \simeq_{\mathcal{R}} \hat{L}$. L has the option to complete if and only if $\hat{L}$ has the option to complete.*

Proof. ($\Rightarrow$) By contradiction. Assume $\hat{L}$ does not have the option to complete, that is, there is a reachable state $\hat{q} \in \hat{\mathcal{Q}}$ such that for all $\hat{q}_f \in \hat{\Omega}$, $\hat{L}$: $\hat{q} \stackrel{\overline{*}}{\nrightarrow} \hat{q}_f$.

For reachable state $\hat{q}$, there is a $\psi \in \hat{\mathcal{A}}^*$ such that $\hat{L}$: $\hat{q}_0 \stackrel{\psi}{\Longrightarrow} \hat{q}$. Given the weak bisimulation relation $\mathcal{R}$, we have $(\hat{q}_0, q_0) \in \mathcal{R}^{-1}$. From Proposition 4, there is a $q \in \mathcal{Q}$ such that L: $q_0 \stackrel{\psi}{\Longrightarrow} q$ and $(\hat{q}, q) \in \mathcal{R}^{-1}$. Since L has the option to complete, there is a $q_f \in \Omega$ such that L: $q \xrightarrow{\overline{*}} q_f$, i.e. L: $q \stackrel{\psi'}{\Longrightarrow} q_f$ for some $\psi' \in \mathcal{A}^*$. Given that $(q, \hat{q}) \in \mathcal{R}$, from Proposition 4, there is a $\hat{q}' \in \hat{\mathcal{Q}}$ such that $\hat{L}$: $\hat{q} \stackrel{\psi'}{\Longrightarrow} \hat{q}'$ and $(q_f, \hat{q}') \in \mathcal{R}$. According to Condition (3) in Definition 27, there is a $\hat{q}_f \in \Omega$ such that $\hat{L}$: $\hat{q}' \stackrel{\epsilon}{\Longrightarrow} \hat{q}_f$ and $(q_f, \hat{q}_f) \in \mathcal{R}$. Then we have $\hat{L}$: $\hat{q} \stackrel{\psi'}{\Longrightarrow} \hat{q}_f$, i.e. $\hat{L}$: $\hat{q} \xrightarrow{\overline{*}} \hat{q}_f$. This however contradicts the assumption that "$\hat{L}$ does not have the option to complete". Hence, ($\Rightarrow$) part of the lemma holds.

($\Leftarrow$) Given the weak bisimulation relation $\mathcal{R}$, the proof for ($\Leftarrow$) part of the lemma is symmetrical to that for the ($\Rightarrow$) part.

Lemma 7. *Let* $L = (\mathcal{Q}, \mathcal{A}, \rightarrow, q_0, \Omega)$ *and* $\hat{L} = (\hat{\mathcal{Q}}, \hat{\mathcal{A}}, \rightarrow', \hat{q}_0, \hat{\Omega})$ *be two LTSs.* $L \simeq_{\mathcal{R}} \hat{L}$ *and* $\mathcal{A} = \hat{\mathcal{A}}$*.* L *does not have dead actions if and only if* $\hat{L}$ *does not have dead actions.*

Proof. ($\Rightarrow$) By contradiction. Assume $\hat{L}$ has a dead action $\overline{a} \in \hat{\mathcal{A}}$, that is, for all reachable state $\hat{q} \in \hat{\mathcal{Q}}$, there does *not* exist a $\hat{q}' \in \hat{\mathcal{Q}}$ such that $\hat{L}$: $\hat{q} \xrightarrow{\overline{a}} \hat{q}'$.

Since L does not have dead actions and $\mathcal{A} = \hat{\mathcal{A}}$, $\overline{a} \in \mathcal{A}$ is not a dead action in L, and thus there are $q, q' \in \mathcal{Q}$ and $\psi \in \mathcal{A}^*$ such that L: $q_0 \stackrel{\psi}{\Longrightarrow} q \xrightarrow{\overline{a}} q'$. Given the weak bisimulation relation $\mathcal{R}$, we have $(q_0, \hat{q}_0) \in \mathcal{R}$. From Proposition 4, there is a $\hat{q} \in \hat{\mathcal{Q}}$ such that $\hat{L}$: $\hat{q}_0 \stackrel{\psi}{\Longrightarrow} \hat{q}$ and $(q, \hat{q}) \in \mathcal{R}$, and subsequently, there is a $\hat{q}' \in \hat{\mathcal{Q}}$ such that $\hat{L}$: $\hat{q} \xrightarrow{\overline{a}} \hat{q}'$ and $(q', \hat{q}') \in \mathcal{R}$. This however contradicts the assumption that $\overline{a}$ is a dead action in $\hat{L}$. Hence, ($\Rightarrow$) part of the lemma holds.

($\Leftarrow$) Given the weak bisimulation relation $\mathcal{R}$, the proof for ($\Leftarrow$) part of the lemma is symmetrical to that for the ($\Rightarrow$) part.

Given the definitions of weak bisimulation, we now examine if there is a weak bisimulation relation between the LTS of a well-formed VWF-net (L_{VW}) and the LTS of the corresponding WF-net ($L_{\mathcal{W}}$).

Definition 28 (State space relation $\mathcal{H}$). *Let* $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ *be a well-formed VWF-net, where* $\mathcal{P} = (P, T, F)$ *is a WF-net and* $\mathcal{L} = (S, V, \delta, s_0, G)$ *is an object versioning lifecycle.* $\mathcal{W}_{VW}$ *is the WF-net transformed from* VW*. Let* $L_{VW} = (\mathcal{Q}_{vw}, \mathcal{A}_{vw}, \rightarrow_{vw}, q_0^{vw}, \Omega_{vw})$ *and* $L_{\mathcal{W}} = (\mathcal{Q}_w, \mathcal{A}_w, \rightarrow_w, q_0^w, \Omega_w)$ *be the associated labelled transition systems of* VW *and* $\mathcal{W}_{VW}$ *respectively. The state space relation* $\mathcal{H} \subseteq \mathcal{Q}_{vw} \times \mathcal{Q}_w$ *can be defined as:*

$$\begin{aligned}\mathcal{H} = &\{((i, s_0), i_{\mathcal{W}})\} \cup \{((o, s_f), o_{\mathcal{W}}) | s_f \in G\} \cup \\ &\{((M, s), M + s + \mu) | (i, s_0) \xrightarrow{*} (M, s)\} \cup \\ &\{((\hat{M}{\uparrow}t, s), \hat{M} + e_t + s) | t \in T \backslash T_{\varnothing} \wedge \textsc{Possible}(\hat{M} + \bullet t, t, s)\}\end{aligned}$$

Theorem 1. *Let* $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ *be a well-formed VWF-net, where* $\mathcal{P} = (P, T, F)$ *is a WF-net and* $\mathcal{L} = (S, V, \delta, s_0, G)$ *is an object versioning lifecycle.* $\mathcal{W}_{VW} = (\mathcal{S}, \mathcal{T}, \mathcal{F})$ *is the WF-net transformed from* VW*, where* $\mathcal{T} = T_{\varnothing} \cup X \cup Y \cup Z \cup H \cup \{b\}$*. Let* $L_{VW} = (\mathcal{Q}_{vw}, \mathcal{A}_{vw}, \rightarrow_{vw}, q_0^{vw}, \Omega_{vw})$ *and* $L_{\mathcal{W}} = (\mathcal{Q}_w, \mathcal{A}_w, \rightarrow_w, q_0^w, \Omega_w)$ *be the associated labelled transition systems of* VW *and* $\mathcal{W}_{VW}$ *respectively. The state space relation* $\mathcal{H} \subseteq \mathcal{Q}_{vw} \times \mathcal{Q}_w$ *is a weak bisimulation, i.e.* $L_{VW} \simeq_{\mathcal{H}} L_{\mathcal{W}}$*.*

Proof. First, we prove $\mathcal{H} \subseteq \mathcal{Q}_{vw} \times \mathcal{Q}_w$ is a weak simulation.

(1) $((i, s_0), i_{\mathcal{W}}) \in \mathcal{H}$.

(2) For all $q_{vw} \in \mathcal{Q}_{vw}$ and $q_w \in \mathcal{Q}_w$ such that $(q_{vw}, q_w) \in \mathcal{H}$, and L_{vw}: $q_{vw} \xrightarrow{\alpha} q'_{vw}$ for some $\alpha \in \mathcal{A}_{vw} \cup \{\tau\}$ and some $q'_{vw} \in \mathcal{Q}_{vw}$. If $\alpha = \tau$, then $(q'_{vw}, q_w) \in \mathcal{H}$; otherwise, if $\alpha \in \mathcal{A}$, then there is a $q'_w \in \mathcal{Q}_w$ such that L_w: $q_w \stackrel{\alpha}{\Longrightarrow} q'_w$ and $(q'_{vw}, q'_w) \in \mathcal{H}$.

(2.1) $q_{vw} = (i, s_0)$ and thus $q_w = i_{\mathcal{W}}$.

(a) $\alpha = \overline{t}$, with $t \in T_\varnothing$ and $i[t\rangle$.

Let $M'{=}i{-}\bullet t{+}t\bullet$, then L_{VW}: $(i, s_0) \xrightarrow{\overline{t}} (M', s_0)$ and $(M', s_0) \in \mathcal{Q}_{vw}$. In $\mathcal{W}_{VW}$, $i_\mathcal{W} \xrightarrow{\mathsf{b}} i{+}s_0{+}\mu$, and since transition b is treated as a silent action, we have $L_\mathcal{W}$: $i_\mathcal{W} \xrightarrow{\tau} i + s_0 + \mu$. Given $t \in T_\varnothing$ and $i \xrightarrow{t} M'$, we have $i{+}s_0{+}\mu \xrightarrow{t} M' + s_0 + \mu$ in $\mathcal{W}_{VW}$, which holds as a specific case of (2.2)(a) with $(M, s) = (i, s_0)$. Since transition $t \in T_\varnothing$ is labelled with action $\overline{t}$ in $L_\mathcal{W}$, we have $L_\mathcal{W}$: $i + s_0 + \mu \xrightarrow{\overline{t}} M' + s_0 + \mu$. Hence, it is true that $L_\mathcal{W}$: $i_\mathcal{W} \xRightarrow{\overline{t}} M' + s_0 + \mu$. Also, from Definition 28, $((M', s_0), M' + s_0 + \mu) \in \mathcal{H}$.

(b) $\alpha = (\overline{t, s_0})$, with $t \in T\backslash T_\varnothing$ and $(i, s_0)[t\rangle$.

Let $\hat{M}{=}i{-}\bullet t$, then L_{VW}: $(i, s_0) \xrightarrow{(\overline{t,s_0})} (\hat{M}{\uparrow}t, s_0)$ and $(\hat{M}{\uparrow}t, s_0) \in \mathcal{Q}_{vw}$. Again, $L_\mathcal{W}$: $i_\mathcal{W} \xrightarrow{\tau} i{+}s_0{+}\mu$. Given $t \in T\backslash T_\varnothing$ and $(i, s_0)[t\rangle$, we have $i + s_0 + \mu \xrightarrow{\mathsf{x}_t^{s_0}} \hat{M} + e_t + s_0$ in $\mathcal{W}_{VW}$, which holds as a specific case of (2.2)(b) with $(M, s) = (i, s_0)$. Since transition $\mathsf{x}_t^{s_0}$ is labelled with action $(\overline{t, s_0})$ in $L_\mathcal{W}$, we have $L_\mathcal{W}$: $i{+}s_0{+}\mu \xrightarrow{(\overline{t,s_0})} \hat{M}{+}e_t{+}s_0$. Hence, it is true that $L_\mathcal{W}$: $i_\mathcal{W} \xRightarrow{(\overline{t,s_0})} \hat{M} + e_t + s_0$. Also, from Definition 28, $((\hat{M}{\uparrow}t, s_0), \hat{M} + e_t + s_0) \in \mathcal{H}$.

(2.2) $q_{vw} = (M, s)$ and thus $q_w = M + s + \mu$.

(a) $\alpha = \overline{t}$, with $t \in T_\varnothing$ and $M[t\rangle$.

Let $M \xrightarrow{t} M'$, then L_{VW}: $(M, s) \xrightarrow{\overline{t}} (M', s)$ and $(M', s) \in \mathcal{Q}_{vw}$. Given $t \in T_\varnothing$, firing t only changes marking M in VW, and based on the transformation rules, task t and its surrounding arcs and places in VW remain the same in $\mathcal{W}_{VW}$. Thus, we have $L_\mathcal{W}$: $M + s + \mu \xrightarrow{\overline{t}} M' + s + \mu$. Also, $((M', s), M' + s + \mu) \in \mathcal{H}$.

(b) $\alpha = (\overline{t, s})$, with $t \in T\backslash T_\varnothing$ and $(M, s)[t\rangle$.

Let $\hat{M}{=}M{-}\bullet t$, then L_{VW}: $(M, s) \xrightarrow{(\overline{t,s})} (\hat{M}{\uparrow}t, s)$ and $(\hat{M}{\uparrow}t, s) \in \mathcal{Q}_{vw}$. Given $t \in T\backslash T_\varnothing$, based on the tranformation rules, t is split into a sequence of x transition, e_t place, and y transition in $\mathcal{W}_{VW}$. The mapping is shown in Figure 7. Since $(M, s)[t\rangle$, when $\mathcal{W}_{VW}$ is in marking $M + s + \mu$, transition x_t^s can fire. Upon firing, x_t^s consumes the tokens in the input places of task t (i.e. $\bullet t$) and the token in the mutex place μ, and marks place e_t (indicating task t is being executed). The object state s remains unchanged. Thus, we have $L_\mathcal{W}$: $M + s + \mu \xrightarrow{(\overline{t,s})} \hat{M} + e_t + s$. Also, $((\hat{M}{\uparrow}t, s), \hat{M} + e_t + s) \in \mathcal{H}$.

(2.3) $q_{vw} = (\hat{M}{\uparrow}t, s)$ and $q_w = \hat{M} + e_t + s$.

In this case, $t \in T\backslash T_\varnothing$, let $M = \hat{M}{+}\bullet t$ then $M[t\rangle$, and there are $s_1, s_n \in S$ where $(s_1, s_n) \in Q(t)$ and $\sigma, \sigma' \in R(t)^*$ such that $s_1 \xrightarrow{\sigma} s \xrightarrow{\sigma'} s_n$.

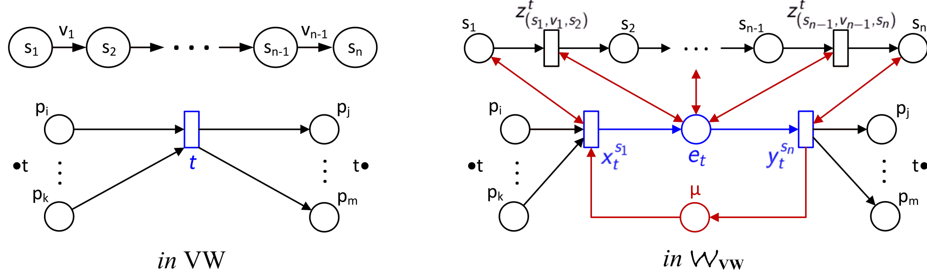


Fig. 7. From VW to $\mathcal{W}_{VW}$: Mapping of task t with state pair $(s_1, s_n) \in Q(t)$ where $s_1 \xrightarrow{v_1 \ldots v_{n-1}} s_n$ and $v_1, ..., v_{n-1} \in R(t)$.

(a) $\alpha = (\overline{t, v})$, with $s \neq s_n$ and $v = \mathsf{head}(\sigma')$.

There is a $s' \in S$ such that $s \xrightarrow{v} s'$, and thus L_{VW}: $(\hat{M}{\uparrow}t, s) \xrightarrow{(\overline{t,v})} (\hat{M}, s')$ and $(\hat{M}, s') \in \mathcal{Q}_{vw}$. In $\mathcal{W}_{VW}$, transition $\mathsf{z}^t_{(s,v,s')}$ is enabled in marking $\hat{M} + e_t + s$ (see Figure 7). Since the firing of transition $\mathsf{x}^{s_1}_t$ marks place e_t while it unmarks the mutex place μ, no e places except e_t are marked. Thus, when $\mathcal{W}_{VW}$ is in marking $\hat{M} + e_t + s$, $\mathsf{z}^t_{(s,v,s')}$ is the only transition that can fire and consume the token in s. Firing $\mathsf{z}^t_{(s,v,s')}$ only changes object states, leading to marking $\hat{M} + e_t + s'$. Since transition $\mathsf{z}^t_{(s,v,s')}$ in $\mathcal{W}_{VW}$ is labelled with action $(\overline{t, v})$ in $L_{\mathcal{W}}$, we have $L_{\mathcal{W}}$: $\hat{M} + e_t + s \xrightarrow{(\overline{t,v})} \hat{M} + e_t + s'$. Also, $(\hat{M}{\uparrow}t, s'), \hat{M} + e_t + s') \in \mathcal{H}$.

(b) $\alpha = (\overline{t, s})$, and $s = s_n$.

Let $M' = \hat{M} + t\bullet$, then L_{VW}: $(\hat{M}{\uparrow}t, s) \xrightarrow{(\overline{t,s})} (M', s)$ and $(M', s) \in \mathcal{Q}_{vw}$. In $\mathcal{W}_{VW}$, transition y^s_t is enabled as long as the e_t and s places are marked, and thus can fire in marking $\hat{M} + e_t + s$ (see modelling of $\mathsf{y}^{s_n}_t$ in Figure 7). Firing y^s_t unmarks place e_t and marks the output places of task t (i.e. $t\bullet$) (changing marking $\hat{M} + e_t$ to M'), and also releases the token to the mutex place μ. The object state s remains marked. Since transition y^s_t in $\mathcal{W}_{VW}$ is labelled with action $(\overline{t, s})$ in $L_{\mathcal{W}}$, we have $L_{\mathcal{W}}$: $\hat{M} + e_t + s \xrightarrow{(\overline{t,s})} M' + s + \mu$. Also, $((M', s), M' + s + \mu) \in \mathcal{H}$.

(c) $\alpha = \overline{t'}$, with $t' \in T_{\varnothing}$ and $(M - \bullet t)[t'>$.

Let $\hat{M} \xrightarrow{t'} \hat{M}'$, then L_{VW}: $(\hat{M}{\uparrow}t, s) \xrightarrow{\overline{t'}} (\hat{M}'{\uparrow}t, s)$ and $(\hat{M}'{\uparrow}t, s) \in \mathcal{Q}_{vw}$. Since firing task $t' \in T_{\varnothing}$ only changes marking $\hat{M}$, we have $L_{\mathcal{W}}$: $\hat{M} + e_t + s \xrightarrow{\overline{t'}} \hat{M}' + e_t + s$. Also, $(\hat{M}'{\uparrow}t, s), \hat{M}' + e_t + s) \in \mathcal{H}$.

(3) $q_{vw} = (o, s_f)$ and thus $q_w = o + s_f + \mu$, where $s_f \in G$.

This implies that $\alpha = \tau$. In $\mathcal{W}_{VW}$, $o + s_f + \mu \xrightarrow{\mathsf{h}_{s_f}} o_w$, and since transition h_{s_f} is treated as a silent action, we have $L_{\mathcal{W}}$: $o + s_f + \mu \xrightarrow{\tau} o_w$. Also, from Definition 28, $((o, s_f), o_{\mathcal{W}}) \in \mathcal{H}$.

Since $\mathcal{H}$ satisfies all three conditions in Definition 27, it is a weak simulation.

Next, we prove $\mathcal{H}^{-1} \subseteq \mathcal{Q}_w \times \mathcal{Q}_{vw}$ is a weak simulation. From Definition 28,

$$\begin{aligned}\mathcal{H}^{-1} = &\{(i_{\mathcal{W}}, (i, s_0))\} \cup \{(o_{\mathcal{W}}, (o, s_f)) | s_f \in G\} \cup \\ &\{(M + s + \mu, (M, s)) | (i, s_0) \xrightarrow{*} (M, s)\} \cup \\ &\{(\hat{M} + e_t + s, (\hat{M}{\uparrow}t, s)) | t \in T \backslash T_{\varnothing} \wedge M[t> \wedge \textsc{Possible}(\hat{M} + \bullet t, t, s)\}\end{aligned}$$

(1) $(i_{\mathcal{W}}, (i, s_0)) \in \mathcal{H}^{-1}$.

(2) For all $q_w \in \mathcal{Q}_w$ and $q_{vw} \in \mathcal{Q}_{vw}$ such that $(q_w, q_{vw}) \in \mathcal{H}^{-1}$, and L_w: $q_w \xrightarrow{\alpha} q'_w$ for some $\alpha \in \mathcal{A}_w \cup \{\tau\}$ and some $q'_w \in \mathcal{Q}_w$. If $\alpha = \tau$, then $(q'_w, q_{vw}) \in \mathcal{H}^{-1}$; otherwise, if $\alpha \in \mathcal{A}_w$, then there is a $q'_{vw} \in \mathcal{Q}_{vw}$ such that L_{vw}: $q_{vw} \xRightarrow{\alpha} q'_{vw}$ and $(q'_w, q'_{vw}) \in \mathcal{H}^{-1}$.

(2.1) $q_w = i_{\mathcal{W}}$ and thus $q_{vw} = (i, s_0)$.
This implies that $\alpha = \tau$. In $\mathcal{W}_{VW}$, $i_{\mathcal{W}} \xrightarrow{\mathsf{b}} i + s_0 + \mu$, and thus $L_{\mathcal{W}}$: $i_{\mathcal{W}} \xrightarrow{\tau} i + s_0 + \mu$. Also, $(i + s_0 + \mu, (i, s_0)) \in \mathcal{H}^{-1}$.

(2.2) $q_w = M + s + \mu$ and thus $q_{vw} = (M, s)$.

(a) $\alpha = \bar{t}$, with $t \in T_{\varnothing}$ and $M[t>$.
In $\mathcal{W}_{VW}$, transition $t \in T_{\varnothing}$ is not connnected with any object state s or the mutex place μ, and thus firing t only changes marking M. Also, t and its surrounding arcs and places are the same as in VW. Let $M \xrightarrow{t} M'$, we have $L_{\mathcal{W}}$: $M + s + \mu \xrightarrow{\bar{t}} M' + s + \mu$. Next, from Definition 25, if $t \in T_{\varnothing}$ and $M \xrightarrow{t} M'$, then L_{VW}: $(M, s) \xrightarrow{\bar{t}} (M', s)$. Also, $(M' + s + \mu, (M', s)) \in \mathcal{H}^{-1}$.

(b) $\alpha = (\overline{t, s})$, with $t \in T \backslash T_{\varnothing}$ and $(M, s)[t >$.
In $\mathcal{W}_{VW}$, transition $t \in T \backslash T_{\varnothing}$ is split into a sequence of x transition, e_t place, and y transition (see Figure 7). Since $(M, s)[t>$ in VW, transition x_t^s is enabled in marking $M + s + \mu$ in $\mathcal{W}_{VW}$, and upon its firing, the marking changes to $(M - \bullet t) + e_t + s$. Let $\hat{M} = M - \bullet t$, we have $L_{\mathcal{W}}$: $M + s + \mu \xrightarrow{(\overline{t,s})} \hat{M} + e_t + s$. Next, given $t \in T \backslash T_{\varnothing}$ and $(M, s)[t>$, from Definition 25, we have L_{VW}: $(M, s) \xrightarrow{(\overline{t,s})} (\hat{M}{\uparrow}t, s)$. Also, $(\hat{M} + e_t + s, (\hat{M}{\uparrow}t, s)) \in \mathcal{H}^{-1}$.

(2.3) $q_w = \hat{M} + e_t + s$ and thus $q_{vw} = (\hat{M}{\uparrow}t, s)$.

In this case, $t \in T \backslash T_{\varnothing}$, let $M = \hat{M} + \bullet t$ then $M[t>$, and there are $s_1, s_n \in S$ where $(s_1, s_n) \in Q(t)$ and $\sigma, \sigma' \in R(t)^*$ such that $s_1 \xrightarrow{\sigma} s \xrightarrow{\sigma'} s_n$.

(a) $\alpha = (\overline{t, v})$, with $s \neq s_n$ and $v = \mathsf{head}(\sigma')$.
There is a $s' \in S$ such that $s \xrightarrow{v} s'$. In $\mathcal{W}_{VW}$, given $s \neq s_n$, when both e_t and s are marked, $\mathsf{z}^t_{(s,v,s')}$ is the *only* transition that can fire and change the object state from s to s' (see Figure 7). Also, firing a z transition *only* changes object states. Thus, we have

$L_{\mathcal{W}}$: $\hat{M}+e_t+s \xrightarrow{\overline{(t,v)}} \hat{M}+e_t+s'$. Next, based on Definition 25, we have L_{VW}: $(\hat{M}{\uparrow}t, s) \xrightarrow{\overline{(t,v)}} (\hat{M}{\uparrow}t, s')$. Also, $(\hat{M}+e_t+s', (\hat{M}{\uparrow}t, s')) \in \mathcal{H}^{-1}$.

(b) $\alpha = \bar{t}$, with $s = s_n$.
In $\mathcal{W}_{VW}$, when $s = s_n$, transition y_t^s is enabled in marking $\hat{M}+e_t+s$ (see $\mathsf{y}_t^{s_n}$ in Figure 7). Let $M' = \hat{M} + t\bullet$, then $M \xrightarrow{t} M'$, and firing y_t^s unmarks place e_t and marks the output places of task t, and also releases the token to the mutex place μ. The object state s remains marked. The marking changes to $M' + s + \mu$. Thus, we have $L_{\mathcal{W}}$: $\hat{M} + e_t + s \xrightarrow{\overline{(t,s)}} M' + s + \mu$. Next, based on Definition 25, we have $(L_{VW}$: $\hat{M}{\uparrow}t, s) \xrightarrow{\overline{(t,s)}} (M', s))$. Also, $(M'+s+\mu, (M', s)) \in \mathcal{H}^{-1}$.

(c) $\alpha = \overline{t'}$, with $t' \in T_\varnothing$ and $\hat{M}[t'>$.
Let $\hat{M} \xrightarrow{t'} \hat{M}'$, given $t' \in T_\varnothing$, we have $L_{\mathcal{W}}$: $\hat{M}+e_t+s \xrightarrow{\overline{t'}} \hat{M}'+e_t+s$. Based on Definition 25, we have L_{VW}: $(\hat{M}{\uparrow}t, s) \xrightarrow{\overline{t'}} (\hat{M}'{\uparrow}t, s)$. Also, $(\hat{M}' + e_t + s, \hat{M}'{\uparrow}t, s)) \in \mathcal{H}^{-1}$.

(3) $q_w = o + s_f + \mu$ and thus $q_{vw} = (o, s_f)$, where $s_f \in G$.
This implies that $\alpha = \tau$. In $\mathcal{W}_{VW}$, $o+s_f+\mu \xrightarrow{\mathsf{h}_{s_f}} o_{\mathcal{W}}$, thus $L_{\mathcal{W}}$: $o+s_f+\mu \xrightarrow{\tau} o_{\mathcal{W}}$ where transition h_{s_f} is treated as a silent action. Also, $(o_{\mathcal{W}}, (o, s_f)) \in \mathcal{H}^{-1}$.

$\mathcal{H}^{-1}$ satifies all three conditions in Definition 27 and is a weak simulation.

Since both $\mathcal{H}$ and $\mathcal{H}^{-1}$ are weak simulations, $\mathcal{H}$ is a weak bisimulation, i.e. the theorem holds.

Based on the above, we can now reason about the behavioural compliance properties of a well-formed VWF-net VW using the soundness properties of the corresponding WF-net $\mathcal{W}_{VW}$.

Lemma 8. *A well-formed VWF-net VW has the option to complete if and only if $\mathcal{W}_{VW}$ has the option to complete.*

Proof. From Lemma 2, it follows that VW has the option to complete if and only if L_{VW} has the option to complete. From Theorem 1, $L_{VW} \simeq_{\mathcal{H}} L_{\mathcal{W}}$, and thus from Lemma 6, it follows that L_{VW} has the option to complete if and only if $L_{\mathcal{W}}$ has the option to complete. Next, from Lemma 4, it follows that $L_{\mathcal{W}}$ has the option to complete if and only if $\mathcal{W}_{VW}$ has the option to complete. Hence, VW has the option to complete if and only if $\mathcal{W}_{VW}$ has the option to complete, i.e. the lemma holds.

Lemma 9. *A well-formed VWF-net VW does not have dead tasks and no task has unused versioning annotations if and only if $\mathcal{W}_{VW}$ does not have dead transitions.*

Proof. From Lemma 3, it follows that VW does not have dead tasks and no task has unused version operations if and only if L_{VW} does not have dead actions.

Next, from Theorem 1, $L_{VW} \simeq_{\mathcal{H}} L_{\mathcal{W}}$. As we have $\mathcal{A}_{vw} = \mathcal{A}_w$ (see Definitions 25 and 26), from Lemma 7, it follows that L_{VW} does not have dead actions if and only if $L_{\mathcal{W}}$ does not have dead actions. From Lemma 5, it then follows that $L_{\mathcal{W}}$ does not have dead actions if and only if $\mathcal{W}_{VW}$ does not have dead transitions. Hence, VW does not have dead tasks and no task has unused versioning annotations if and only if $\mathcal{W}_{VW}$ does not have dead transitions, i.e. the lemma holds.

Theorem 2. *A well-formed VWF-net* $VW = (\mathcal{P}, \mathcal{L}, R, Q)$ *is compliant with the object versioning lifecycle* $\mathcal{L}$ *if and only if* $\mathcal{W}_{VW} = (\mathcal{S}, \mathcal{T}, \mathcal{F})$ *is a sound WF-net.*

Proof. From Definition 16, VWF-net VW is compliant with the object versioning lifecycle $\mathcal{L}$ if and only it satisfies four conditions: (1) proper completion, (2) option to complete, (3) no dead tasks, and (4) no unused versioning annotations.

From Definition 9, WF-net $\mathcal{W}_{VW}$ is sound if and only it satisfies three conditions: (1) proper completion, (2) option to complete, and (3) no dead transitions.

From Lemma 8 and Lemma 9, it follows that VW satisfies conditions (2) to (4) if and only if $\mathcal{W}_{VW}$ satisfies conditions (2) and (3).

Below, we prove that VW has proper completion if and only if $\mathcal{W}_{VW}$ has proper completion (i.e. condition (1) for both nets). From Definition 16, when VWF-net VW has proper completion, for all $(M_{\mathcal{P}}, s_f) \in \mathbb{M}_{VW}$ with $s_f \in G$, if $M_{\mathcal{P}} \geq o$ then $M_{\mathcal{P}} = o$. From Defintion 9, when WF-net $\mathcal{W}_{VW}$ has proper completion, for all M_w with $i_{\mathcal{W}} \xrightarrow{*} M_w$, if $M_w \geq o_{\mathcal{W}}$ then $M_w = o_{\mathcal{W}}$.

($\Rightarrow$) By contradiction. Assume that $\mathcal{W}_{VW}$ does not have proper completion, i.e. let $s_f \in G$, there is a marking $(o+M')+s_f+\mu$ with $i_{\mathcal{W}} \xrightarrow{*} (o+M')+s_f+\mu$, such that $(o+M')+s_f+\mu \xrightarrow{\mathsf{h}} o_{\mathcal{W}}+M'$ for some $\mathsf{h} \in \mathcal{T}$. From Definition 26, it follows that $(o+M')+s_f+\mu \in \mathcal{Q}_w$ in $L_{\mathcal{W}}$. From Definition 28, there is a state $(o+M', s_f) \in \mathcal{Q}_{vw}$ in L_{VW} such that $((o+M')+s_f+\mu, (o+M', s_f)) \in \mathcal{H}^{-1}$. From Proposition 2(a), it follows that $(o+M', s_f) \in \mathbb{M}_{VW}$ in VW. This means that there is a marking $(M_{\mathcal{P}}, s_f) \in \mathbb{M}_{VW}$ with $s_f \in G$ such that $M_{\mathcal{P}} > o$ in VW, which contradicts the fact that VW has proper completion. Hence, VW has proper completion *only if* $\mathcal{W}_{VW}$ has proper completion.

($\Leftarrow$) By contradiction. Assume that VW does not have proper completion, i.e. let $s_f \in G$, there is a marking $(o+M, s_f) \in \mathbb{M}_{VW}$. From Proposition 2(a), it follows that $(o+M, s_f) \in \mathcal{Q}_{vw}$ in L_{VW}. From Definition 28, there is a state $(o+M)+s_f+\mu \in \mathcal{Q}_w$ in $L_{\mathcal{W}}$ such that $((o+M, s_f), (o+M)+s_f+\mu) \in \mathcal{H}$. From Definition 26, it follows that $i_{\mathcal{W}} \xrightarrow{*} (o+M)+s_f+\mu$ in $\mathcal{W}_{VW}$. In $\mathcal{W}_{VW}$, when both the sink place o of $\mathcal{P}$ and a final state s_f of $\mathcal{L}$ are marked, the transition h_{s_f} can fire, and upon its firing, it unmarks the places o, s_f and μ, and marks the sink place $o_{\mathcal{W}}$ of $\mathcal{W}_{VW}$, resulting in marking $o_{\mathcal{W}}+M$. This means that there is a marking M_w with $i_{\mathcal{W}} \xrightarrow{*} M_w$ in $\mathcal{W}_{VW}$ such that $M_w > o_{\mathcal{W}}$, which contradicts the fact that $\mathcal{W}_{VW}$ has proper completion. Hence, VW has proper completion *if* $\mathcal{W}_{VW}$ has proper completion.

Finally, in summary, Figure 8 provides a sketch of the structure of the proof that we have conducted in this section to reach the conclusion in Theorem 2.

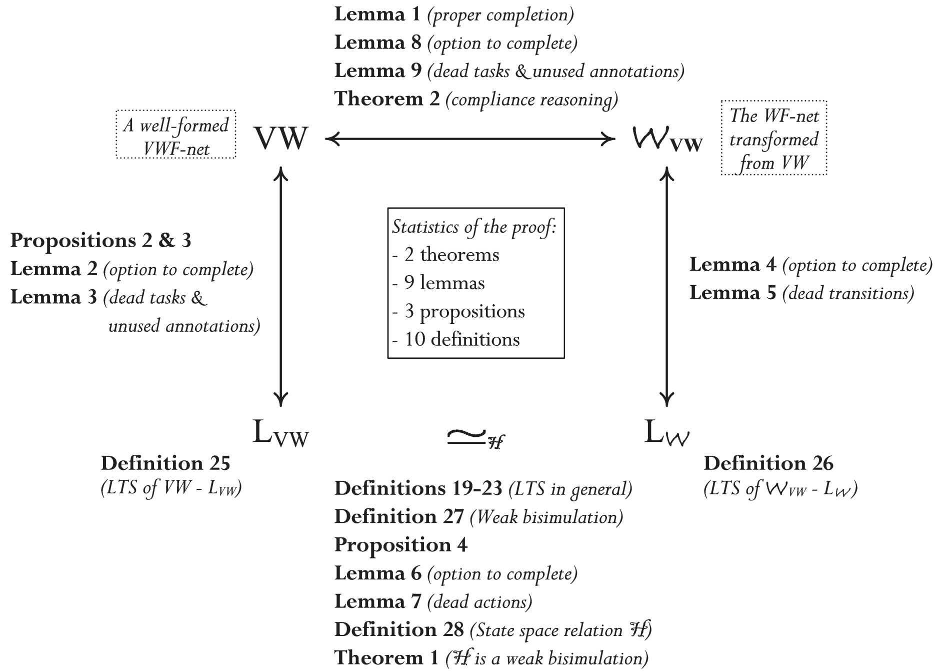


Fig. 8. Structure of the proof for Theorem 2.

4 Tool Support

In this section, we present our tool development efforts to enable version compliance checking of process models. Figure 9 provides an overview of the tool architecture. Our main objective is to provide a VWF-net viewer that possesses both syntactical checking and behaviour compliance checking capabilities. We made use of a well-known, open-source process mining framework (ProM) [4] and developed our tool as a ProM 6 analysis plug-in[9]. There are five components to this tool: namely, versioning-annotated WF-net viewer, a syntactical compatibility checker, a WF-net transformer, a soundness checker and a behavioural compliance interpreter. Among these components, the WF-net transformer, the soundness checker and the behavioural compliance interpreter together constitute the behavioural compliance checker. The soundness checker is provided by an existing ProM plug-in called Woflan that can be used to verify the soundness property of a WF-net [45]. This plug-in is also used to ensure that the WF-net that is used as the input for a VWF-net is sound.

[9] The ProM framework can be downloaded from `http://prom.win.tue.nl/research/wiki/prom/start`

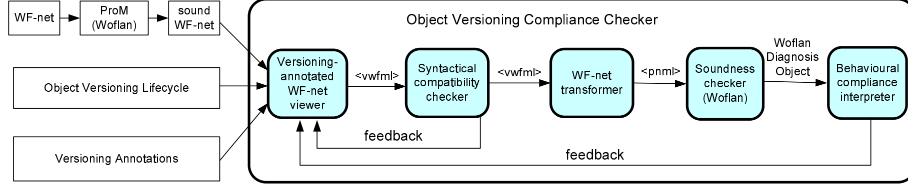


Fig. 9. An overview of the tool structure.

4.1 The XML format for VWF-nets

The concept of versioning-annotated workflow nets (VWF-nets) forms the basis for our compliance checking framework. A VWF-net is generated using the information from PLM systems regarding a business process, the object versioning lifecycles of the objects used in the process together with task versioning annotations. Since each PLM system uses its own formats for describing the process, object versioning lifecycles and task annotations, we developed a generic XML format for VWF-nets (see Figure 10) that is platform-independent. As a result, relevant information from different PLM systems can be translated into this XML format first for compliance checking purposes.

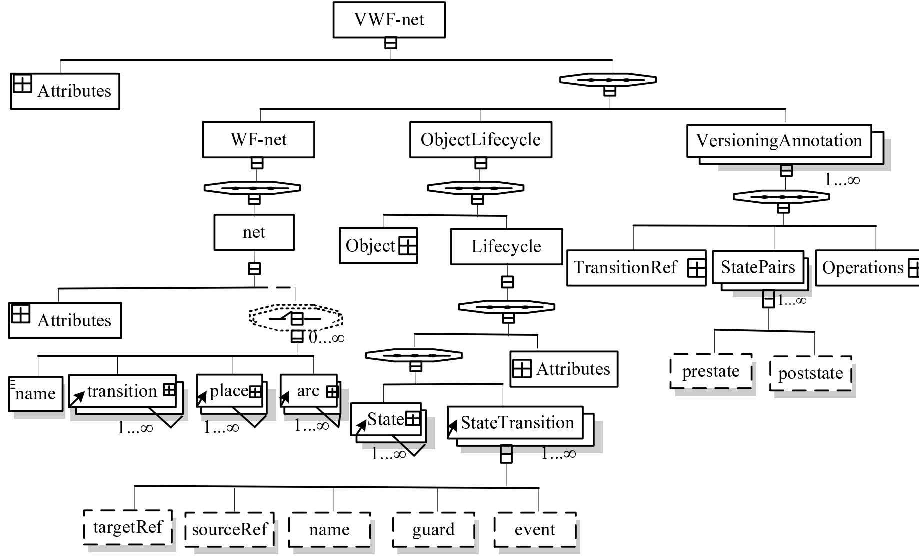


Fig. 10. An XML schema of a versioning-annotated workflow net (VWF-net).

The root node of a VWF-net is labelled as a *VWF-net*, representing a workflow net with versioning annotations. Each VWF-net contains a *WF-net*, an *ObjectLifecycle* and an arbitrary number of *VersionAnnotation* elements as child elements. The *WF-net* element represents a workflow process modelled as a Petri net with the notions of transitions, places and arcs that connect the two. The format of *WF-net* and its sub-elements are in accordance with the Workflow

Net standard provided in [1]. The *ObjectLifecycle* element contains an *Object* element and a *Lifecycle* element, which describe an object handled in a workflow and the object's versioning lifecycle respectively. For the purpose of extensibility, we adopted the XML format of the Java Finite State Machine framework[10] for the *Lifecycle* element. We made use of *State* and *StateTransition* elements (with attributes *name*, *sourceRef* and *targetRef*) to store the information about an object versioning lifecycle. A *VersioningAnnotation* element describes the object's versioning assignment for a task. The *StatePairs* element captures the pre-state and post-state of objects when task is executed. Figure 11 describes an excerpt of the XML document for the VWF-net shown in Figure 5.

```xml
<VWF-net ProcessID="String" ProcessName="String">
    <WF-net>
        <net type="http://www.informatik.hu-berlin.de/top/pnml/basicPNML.rng" id="String">
    </WF-net>
    <ObjectLifecycle>
        <Object id="o1"/>
        <Lifecycle name="o1-LC">
            <State name="s0" type="INITIAL"/>
            <State name="s1" type="NORMAL"/>
            <State name="s2" type="NORMAL"/>
            <State name="s3" type="FINAL"/>
            <State name="s4" type="FINAL"/>
            <StateTransition event="*" guard="false" name="v1" sourceRef="s0" targetRef="s1"/>
            <StateTransition event="*" guard="false" name="v2" sourceRef="s1" targetRef="s2"/>
            <StateTransition event="*" guard="false" name="v3" sourceRef="s2" targetRef="s1"/>
            <StateTransition event="*" guard="false" name="v4" sourceRef="s2" targetRef="s3"/>
            <StateTransition event="*" guard="false" name="v5" sourceRef="s2" targetRef="s4"/>
            <StateTransition event="*" guard="false" name="v5" sourceRef="s1" targetRef="s4"/>
        </Lifecycle>
    </ObjectLifecycle>
    <VersioningAnnotation>
        <Operations>
            <OperationRef name="v1"/>
            <OperationRef name="v2"/>
        </Operations>
        <StatePairs>
            <StatePair prestate="s0" poststate="s2"/>
        </StatePairs>
        <TransitionRef id="T2"/>
    </VersioningAnnotation>
    <VersioningAnnotation>
    <VersioningAnnotation>
</VWF-net>
```

Fig. 11. An excerpt of the XML document for the VWF-net shown in Figure 5.

4.2 Constructing a VWF-net

A VWF-net includes three parts: WF-net, object versioning lifecycle and task versioning annotation information (including task versioning assignment information and task object-state pairs). Among these, the WF-net is derived automatically using the TiWorkflow model convertor. Task versioning assignment information is also encoded automatically by an extractor that was developed to extract the versioning assignment information from the TiPLM database. The object versioning lifecycle information and task object-state pairs are then manually added.

Below, we illustrate how the information from TiPLM systems is encoded in a VWF-net. This implementation is intended to serve as a reference for mapping data from other PLM systems.

1. *Translating a TiWorkflow to a WF-net*

[10] http://unimod.sourceforge.net/fsm-framework.html

We developed a TiWorkflow convertor to translate a TiWorkflow model into a WF-net based on the translation algorithm provided by Zha et.al [54]. This convertor is deployed into TiPLM system as a plug-in. The translation algorithm only focuses on the control flow aspects of a workflow model (i.e., it does not take the business objects and related data into account).
We now briefly present how this is achieved. The elements of a TiWorkflow model are divided into three types: task nodes, links and routing nodes. Example task nodes are shown in Figure 12(a). In a TiWorkflow definition, nodes can be connected directly by a link. In a Petri net, two transitions cannot be connected directly, so a place is inserted between such transitions. A sequence of transitions with links models the execution of a task node, as shown in Figure 12(b). In addition, task routing is implemented by routing nodes in a TiWorkflow definition. There are six types of routing nodes, as shown in Figure 12(c). If we abstract from a condition expression as a Boolean function, a CON-join becomes an XOR-join, and a CON-split becomes an XOR-split. The Petri net semantics for the four kinds of routing nodes are shown in Figure 12(d). The detailed mapping semantics from a TiWorkflow to a WF-net can be found in [54][11]

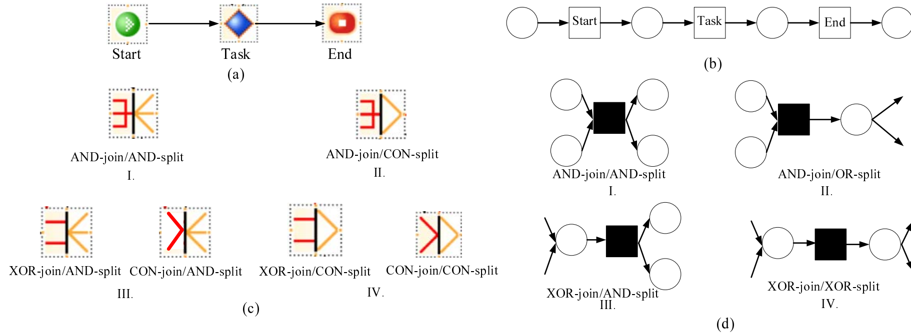


Fig. 12. Semantics of TiWorkflow Elements.

2. *Annotating a WF-net with task versioning assignment*
For task versioning assignment, we developed a versioning assignment extractor. The function of the extractor is twofold: (1) to extract the versioning assignment information from all assigned data operations (e.g. *download*, *browse*, *read*, *write*, *check-in*) in the TiPLM database, and (2) to transform the extracted versioning assignment information to the XML format defined in Section 4.1.
3. *Deriving object versioning lifecycles and object-state pairs*
Product data management in a TiPLM system provides the overall object versioning lifecycle of a business object. Many standardised business processes (e.g., accounting, engineering design process, change process) in a PLM system are required to transform these business objects from one valid

[11] In [54] the terms OR-split and OR-join are used rather than XOR-split and XOR-join. As the semantics of these constructs are that of exclusive splits and joins respectively, we choose the latter terms.

state to another based on the corresponding object versioning lifecycle. The object versioning lifecycle for a certain process can often be derived from the overall object versioning lifecycle. In addition, *task object-state pairs* refine the above interrelationship further by specifying the relationship at the task level instead of the process level. *Task object-state pairs* represent the expected business object states before a task is carried out and afterwards. For instance, in the engineering design processes, the expected state of the object before the *Design* task carried out is *Initial* or *Checked-in* while the expected state of the object after the *Design* task is completed is *Checked-out*.

All this information can be derived in consultation with domain experts in the TiPLM systems. Currently, the object versioning lifecycle information and task object-state pairs are encoded manually in a VWF-net. In the future, this could be improved by extending the versioning-annotated WF-net viewer with an intuitive user interface that allows domain experts to supply the object versioning lifecycle and annotated task object-state pairs information directly.

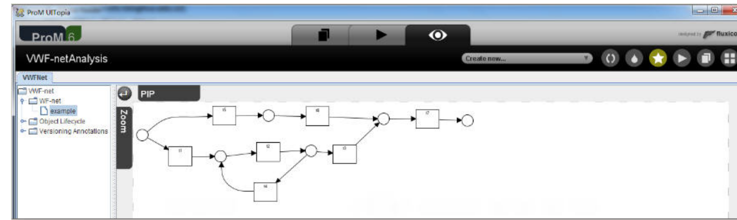
4. *Displaying the resulting VWF-net*
   In order for users to view a VWF-net, we implemented a plug-in in ProM that allows a VWF-net in an XML format to be imported. The main functions of this component are to create a VWF-net object in ProM based on the imported XML file and to display the three parts of a VWF-net. Figure 13 shows the screenshot of the VWF-net from Figure 5.

4.3 Syntactical Compatibility Checker

This component supports the syntactical compatibility checking for VWF-nets. It implements the six syntactical conditions checks defined in Definition 11 (e.g., *empty annotation consistency*, *version operation assignment completeness*, *local object path existence*, *no locally assigned dead version operation*, *no dead object state transition* and *global object path existence*). Figure 14 provides a screenshot of the syntactical compatibility checking results for the erroneous VWF-net in Figure 5. When there are no more syntactical errors in a VWF-net, the VWF-net is then transformed into the corresponding WF-net for behavioural compliance checking.

4.4 WF-net Transformer

This component supports the transformation of a VWF-net into a WF-net so that we can carry out further behavioural compliance checking using the Woflan plug-in. The implemented tranformations are in accordance with the transformation rules defined in Section 3. The upper part of Figure 15(a) shows the WF-net resulting from the transformation of the syntactical correct version of the VWF-net in Figure 5.



(a) business process model

(b) object versioning lifecycle

Task / s	Version Operations	State Pairs
T2	{v1,v2}	{(s0,s2)}
T6	{v2,v3}	{(s2,s2)}
T7	{v4,v5}	{(s2,s3),(s2,s4)}

(c) task versioning annotation

Fig. 13. A screenshot of the VWF-net in Figure 5.

4.5 Soundness Checker

For the purpose of behavioural compliance checking, we made use of an existing ProM plug-in called Woflan [45]. Woflan can be used to verify the soundness property of a WF-net and provides detailed diagnostic information for unsound workflow nets. In Section 3, we proved the behavioural relationship between the versioning-annotated workflow net and the transformed WF-net. Thus, we can use the soundness verification results from the transformed WF-net to reason about the behavioural compliance properties of the original VWF-net.

The interactions between our compliance checker plug-in and the Woflan plug-in are as follows. The compliance checker uses the interface provided by Woflan to carry out the soundness analysis of the transformed WF-net. Woflan returns a WoflanDiagnosis object for the WF-net that contains the analysis results (i.e., the results of the *proper completion*, *option to complete* and *no dead tasks* properties). It should be noted that Woflan checks soundness of WF-net in the following order: firstly the *proper completion* property is checked, secondly the *no dead tasks* property is checked and finally the *option to complete* property is checked. If all steps succeed, then the net is sound. If one fails, the net is not

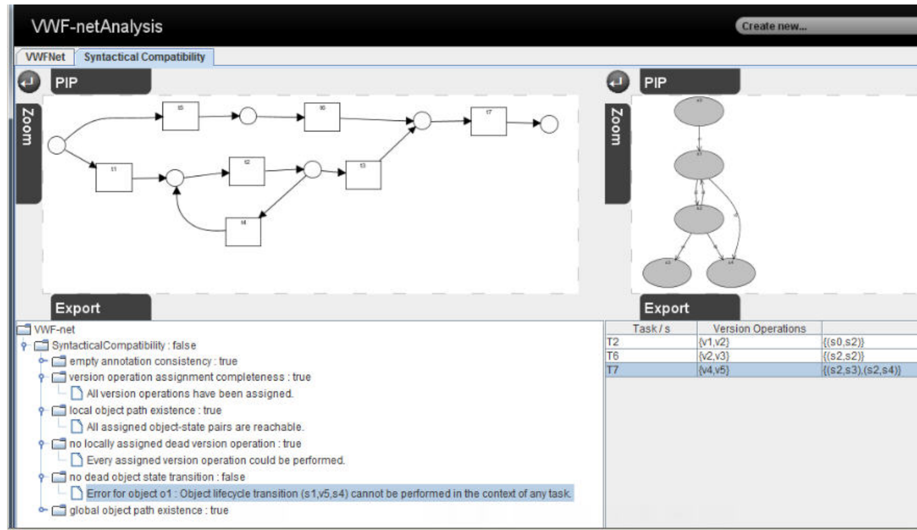


Fig. 14. Results of the syntax check for the VWF-net shown in Figure 5.

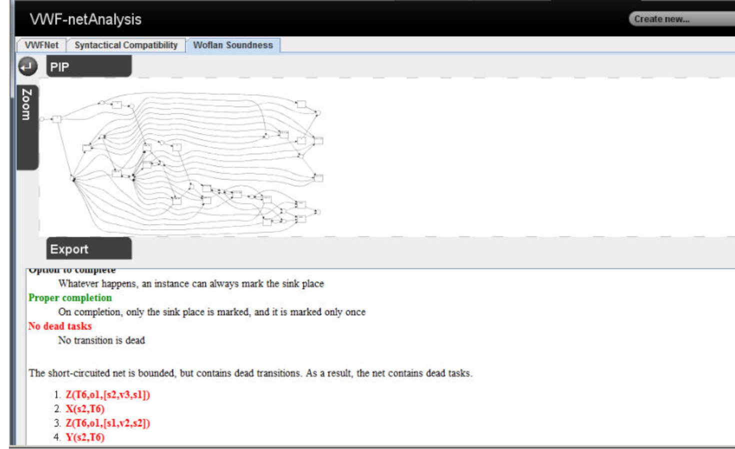
sound and the verification process ends. In such cases, the results of the later properties will be unknown. Thus, if the net is not sound and the property *option to complete* is not satisfied, we can not determine whether the *no dead tasks* property is satisfied. However, if the net is not sound and the *no dead tasks* property is not satisfied, the *option to complete* property is not checked and the result is unknown. Hence, this property should be checked again after the problem of *no dead tasks* is fixed.

The lower part of Figure 15(a) shows the analysis result of the transformed WF-net for the revised VWF-net shown in Figure 5. The soundness result of the transformed WF-net states that the net is not sound as the *no dead tasks* property is not satisfied. The result for the *option to complete* property is unknown in this case.

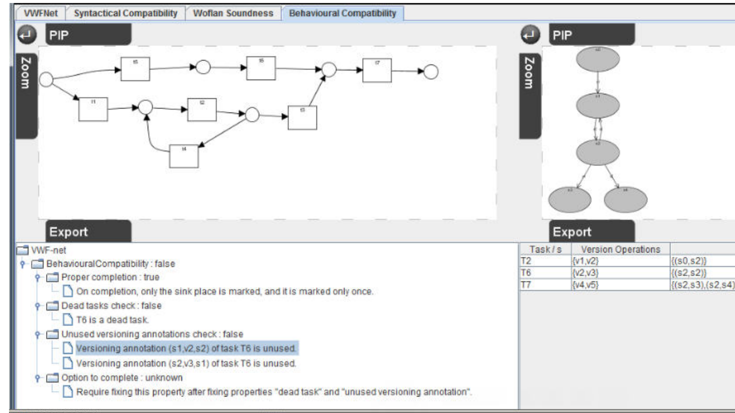
4.6 Behavioural compliance interpreter

When checking the behavioural compliance of a well-formed VWF-net (VW), error messages reported by Woflan refer to the WF-net ($\mathcal{W}_{VW}$). To use such messages as feedback for the Versioning-annotated WF-net viewer (see Figure 9), they need to be interpreted in terms of the original VWF-net. This functionality is provided by the behavioural compliance interpreter component.

Firstly, if $\mathcal{W}_{VW}$ does not have the option to complete, then VW does not have the option to complete. Woflan reports all possible transitions and markings in which firing a transition in a corresponding marking prevents the process from completion in $\mathcal{W}_{VW}$. In this case, the behavioural compliance interpreter maps the reported markings and transitions in $\mathcal{W}_{VW}$ back to the corresponding markings and tasks in VW.



(a) The results of soundness checking of the transformed WF-net



(b) The results of behavioural compliance checking of the VWF-net

Fig. 15. Screenshots for behavioural compliance verification for the revised VWF-net of Figure 5 (without state transition (s_1, v_5, s_4) in the lifecycle of o_1).

Secondly, when there are dead transitions in $\mathcal{W}_{VW}$, there are dead tasks and/or tasks that have unused versioning annotations in VW. If a dead transition in $\mathcal{W}_{VW}$ corresponds to a task in VW, such a task has an empty versioning annotation, in which case, the behavioural compliance interpreter can report the task directly. Otherwise, if a dead transition in $\mathcal{W}_{VW}$ is an X, Y, or Z transition, such a transition is part of the mapping of a task that has a non-empty versioning annotation in VW. In this case, the behavioural compliance interpreter checks if the corresponding task in VW is a dead task or if it has unused versioning annotations.

Figure 15(b) shows the interpreted results of the behavioural compliance interpreter for the revised VWF-net shown in Figure 5: there is a dead task, T_6,

and there are unused versioning annotations, (s_1, v_2, s_2) and (s_2, v_3, s_1), assigned to task T_6[12].

5 Experiments

In this section, we illustrate the applicability of our approach to a real PLM system by carrying out experiments that are based on process models and object versioning information found in the TiPLM system.

5.1 Data Collection

We collected 48 design processes, the object versioning lifecycle information as well as the corresponding versioning assignment information from the THsoft InfoTech company[13]. This company develops the TiPLM system which is widely used in Mainland China (well over 100 companies in the manufacturing industry in China adopted the TiPLM system). In order to sustain development and research, this company maintains domain-specific process model templates and also possesses a collection of deployed business process models.

As a first step, these 48 TiPLM process models were mapped to WF-nets using the TiWorkflow converter contained in the TiPLM system. We then anonymised the models and selected those processes, that were determined to be sound[14]. There were 42 such models and these were used as input models for our experiments.

Next we used a versioning assignment extractor that was specially written for this purpose to extract the object versioning assignment information from the TiPLM system. It should be mentioned that in the TiPLM system, a task can work on more than one data object (i.e., a task can be assigned operations on more than one object). However, there are no interdependencies between these objects. Thus, we perform compliance checking with respect to each object in isolation. For each business object handled in a process, a VWF-net is constructed which annotates the derived WF-net with versioning lifecycle and versioning assignment information. There are altogether 142 business objects handled in the selected process models.

Finally, we interviewed domain experts from THsoft InfoTech company (including the technical director, a senior consultant and an implementation manager) to determine appropriate assignment of object-state pairs to tasks. These assignments were then incorporated into the corresponding VWF-nets. Thus, the total number of VWF-nets used in the experiments is 142. These anonymised VWF-nets can be downloaded from `http://www.yawlfoundation.org/research/compliance`.

---

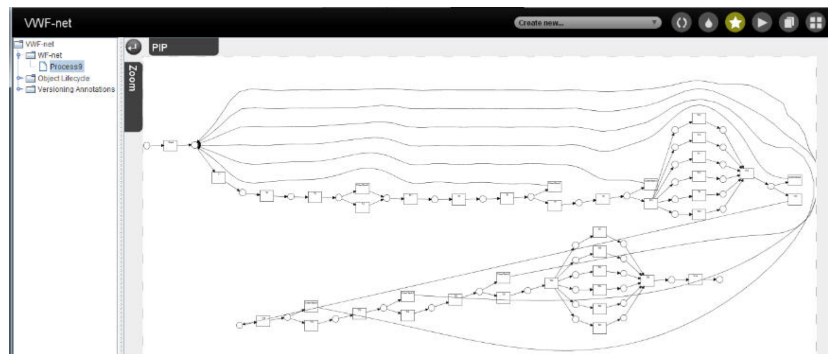
[12] One way to ensure compliance is to change the alternative relationship between T_2 and T_6 into a sequential one.

[13] http://www.thit.com.cn

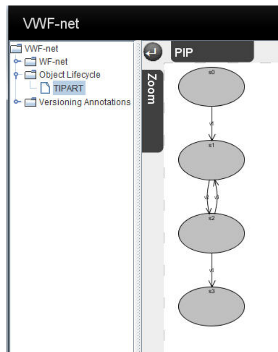
[14] We used the Woflan [45] plug-in in ProM [4] to verify the soundness property of these models.

5.2 Tool Application

Using the collection of 142 VWF-nets, we used the versioning compliance checker that is implemented and integrated in the ProM framework to check versioning compatibility. We first performed syntactical checking on all 142 VWF-nets. Figure 16 shows one such VWF-net which is an engineering part-list designing and auditing process with the anonymised versioning lifecycle of the business object *TIPART*. Figure 17 shows the results from the syntactical checks carried out for the net in Figure 16. When syntactical errors were detected, we discussed these errors with the domain experts to find a way to remedy them. After all syntactical errors were found and fixed, we carried out behavioural compatibility checks. Figure 18(a) shows the analysis result of the transformed WF-net from Figure 16. Figure 18(b) describes the interpreted compliance results of the VWF-net from the Woflan soundness results in Figure 18(a).



(a) business process model



VWF-net — Create new...

VWF-net / WF-net / Process9 / Object Lifecycle / TIPART / Versioning Annotations / Task Annotations

Task / s	Version Operations	State Pairs
T28	{v2}	{}
T24	{v2}	{}
T23	{v2}	{}
T27	{v2}	{}
T25	{v2}	{}
T26	{v2}	{}
T2	{v1,v2,v3}	{(s0,s2),(s2,s2)}
T5	{v2}	{}
T7	{v2}	{}
T15	{v2}	{}
T17	{v2}	{}
T4	{v2}	{}
T9	{v2}	{}
T10	{v2}	{}
T12	{v2}	{}
T11	{v2}	{}
T13	{v2}	{}
T14	{v2}	{}
T19	{}	{(s2,s3)}

(b) object versioning lifecycle (c) task versioning annotation

Fig. 16. Display of an anonymised VWF-net from the TiPLM system.

5.3 Validation

In this subsection, we first present the results of the syntactical compatibility checks followed by the results of the behavioural compliance checks.

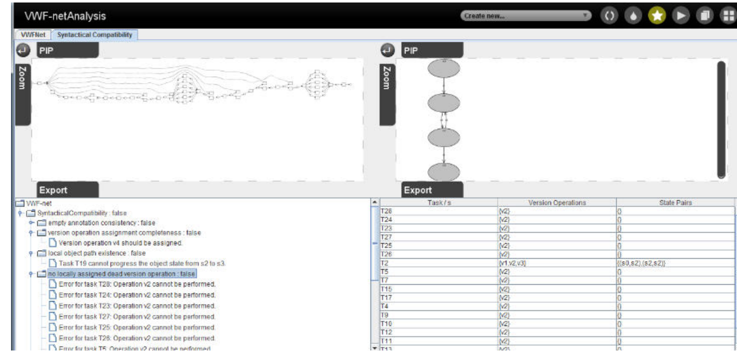


Fig. 17. The screenshot of the syntactical checking phase.

Syntactical compatibility checking results: There are six rules to check syntactical compatibility of VWF-nets: namely, the *empty annotation consistency*, the *version operation assignment completeness*, the *local object path existence*, the *no locally assigned dead version operation*, the *no dead object state transition*, and the *global object path existence*.

The first rule, the *empty annotation consistency*, represents the necessary preconditions for a valid VWF-net and we only continued to check the next five syntactical rules when a VWF-net satisfied this rule. Table 2 shows the results of syntactical checking for the first rule. We noted that 57 out of 142 VWF-nets (40%) fail this rule due to the fact that certain tasks have certain version operations assigned to them but not object-state pairs (condition 1). In addition to that, 122 VWF-nets out of 142 (86%) fail the first rule due to the fact that tasks contained object-state pairs even though no corresponding version operations were assigned (condition 2). For all these VWF-nets where the first rule was violated, the following actions were taken. If the rule was violated because a task had version operations assigned to it but no corresponding object-state pairs (condition 1), we removed its version operations assignment. If on the other hand the rule was violated because the task had object-state pair information but did not have any version operations assigned to it, we added required version operations to the task after discussions with the domain experts. Table 3 shows the results from the remaining five rules. The table illustrates these results in two groups: the group of results on the left is derived from the original set of VWF-nets before they were modified based on the *empty annotation consistency* check shown in Table 2. The group of results shown on the right in Table 3 is derived from VWF-nets that were modified in order to pass this check (in the way discussed before). Before VWF-nets were modified to satisfy the conditions of the first rule, there were 113 (80%) VWF-nets that did not pass the second rule, 127 (89%) for the third rule, 59 (42%) for the fourth rule, and 15 (11%) for the fifth rule respectively. With the modified set of VWF-nets that satisfy the first rule, the error ratios decreased significantly. We found that only 19 VWF-nets (13%) failed the second rule, 4 (3%) the third

Woflan Diagnosis on Net "Process9_TIPART"

The net is not a sound workflow net.

Soundness requirements

Option to complete

Whatever happens, an instance can always mark the sink place

Proper completion

On completion, only the sink place is marked, and it is marked only once

No dead tasks

No transition is dead

The short-circuited net is bounded, contains no dead transitions, but is not live. As a result, completion is not always possible.

The following transitions are not live in the short-circuited net

1. **T4**
2. **SilentTask6**
3. **R1**

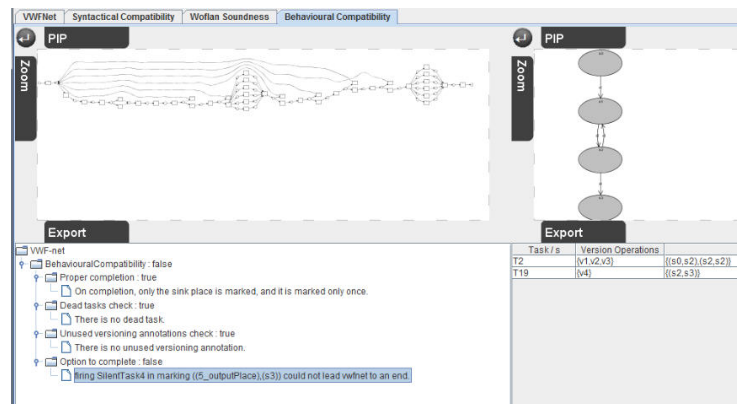
......

The following diagnostic information assumes that there exists a part of the state space from which completion is still possible. Clearly, to avoid losing the option to complete, behavior should be restricted to this part. Thus, any transition leaving the part should be disabled.

Disabling the following transitions at the following (reachable) markings effectively would restrict the behavior to the part from which completion is possible:

1. Transition **SilentTask4** at marking **[5_outputPlace,s3]**.

(a) An excerpt of the Woflan soundness result for the transformed WF-net



(b) Translated behavioural compliance results of the VWF-net

Fig. 18. The result of behavioural compliance checking.

rule, 19 (13%) the fourth rule and none of the VWF-nets failed the fifth and sixth rules.

We now describe the actions taken to remedy violations of rules 2, 3 and 4. As shown in Table 3, there are 19 (13%) VWF-nets that did not pass the *version operation assignment completeness* check (Rule 2). This means that these models cannot guarantee the evolution of a business object from its initial state to one of

	#Passed	#Failed	Error ratio
Rule1-1	85	57	40%
Rule1-2	20	122	86%

Table 2. Results from the syntactical checking (Rule 1).

	Before remedied based on rule1			After remedied based on rule1		
	#Passed	#Failed	Error ratio	#Passed	#Failed	Error ratio
Rule2	29	113	80%	123	19	13%
Rule3	15	127	89%	138	4	3%
Rule4	83	59	42%	123	19	13%
Rule5	127	15	11%	142	0	0%
Rule6	142	0	0%	142	0	0%

Table 3. Results from the syntactical checking (Rules 2-6).

	#Passed	#Failed	Error ratio
Soundness	121	21	15%
Option to complete	128	14	10%
No dead tasks	135	7	5%

Table 4. Soundness result for transformed WF-nets.

	#Passed	#Failed	Error ratio
Behavioural compliance	121	21	15%
Option to complete	128	14	10%
No dead tasks	135	7	5%
No unused versioning annotations	135	7	5%

Table 5. Compliance checking result for VWF-nets.

its final states. Thus, we added the missing versioning operations to appropriate tasks (in some cases the domain experts were consulted for this purpose).

There are 4 (3%) VWF-nets that did not pass the *local object path existence* (Rule 3). This means that additional version operations are needed to guarantee the evolution of a business object from one of its pre-state to a corresponding post-state when a task is executed. Thus, we added the missing operations to corresponding tasks in the VWF-nets.

There are 19 (13%) VWF-nets that did not pass the *no locally assigned dead version operation* (Rule 4). This means that there are some assigned version operations which can not be performed when the corresponding task is executed. Thus, we removed the redundant dead operations from the tasks.

Behavioural compliance checking results: The results from the behavioural compliance checks are aimed at providing a better insight into the compatibility

between process models and object versioning lifecycles. Behavioural compliance of a VWF-net is determined through soundness checking of the corresponding WF-net. A VWF-net is considered to be *compliant with a given object versioning lifecycle* if and only if the corresponding WF-net is sound. We used the behavioural compliance interpreter component to report any errors found as a result of the application of the soundness check to the WF-net back to the users in terms of errors of the VWF-net.

The first behavioural compliance checking results were as follows: there were 21 (15%) unsound WF-nets among the 142 translated WF-nets (see Table 4). Specifically, 14 (10%) WF-nets did not satisfy the *option to complete* rule and 7 (5%) WF-nets did not satisfy the *no dead tasks* rule. Accordingly, there were 21 (15%) non-compliant VWF-nets among the 142 VWF-nets (see Table 5). Specifically, 14 (10%) VWF-nets did not satisfy the *option to complete* rule, 7 (5%) VWF-nets did not satisfy the *no dead tasks* rule, and 7 (5%) VWF-nets did not satisfy the *no unused versioning annotations* rule (the last two collections consisted of the same 7 VWF-nets).

For VWF-nets that violate the *option to complete* rule, it is possible to identify non-live tasks. For example, Figure 18 provides dead markings of a VWF-net. Based on this information, we can identify tasks which can cause problems at certain markings. After careful examination, we found a common cause for the violation of the *option to complete* rule. That is, object-state pairs associated with the final state of an object were assigned to a task that is in a loop structure. Thus, the final state of the object remains unsynchronised with the final/end marking of the process. This is a very valuable finding that could not be detected by analysing a process specification alone. One possible remedy for such errors is to ensure that object-state pairs that contain assignments of final states of objects are only associated with tasks that are not part of a loop structure. To ensure that our suggested remedy complies with the business semantics of a process, we communicated our findings to the domain experts. Similarly, for VWF-nets that violated the *no dead tasks* rule and the *no unused versioning annotation* rule and provided this information to the domain experts. Based on subsequent discussions with these domain experts, we corrected these errors and performed syntactical and behavioural checks in an iterative manner until all VWF-nets were found to be compliant.

During our interactions, the domain experts expressed interest in learning more about our experiments. It is our expectation that when the tool can provide automated support for object-state pair annotations and task assignments, the tool can be used by domain experts to validate their process models before the TiPLM system is deployed in an organisation.

6 Related Work

There is a substantial amount of literature in the field of access control and also considerable attention has been paid to access control in a workflow context (for example modifications of RBAC [34], Role-based Access Control, such as

W-RBAC [47]and T-RBAC [11]). In a workflow system, tasks should only be performed by authorised users and this is enforced through the application of rules (e.g. "..Task A can only be performed by resources that play the role R"). Workflow management is a domain independent technology and not generally concerned with controlling the application of version operations on design objects. Therefore, the problem addressed in this paper cannot be solved through the application of classic workflow access control mechanisms.

There has been a great deal of interest in compliance checking of business process models. There are two driving factors behind this interest. One of these concerns the organisation and its environment and the need to be able to demonstrate to this environment that certain best practices are followed or that certain legislation is adhered to (e.g. the well-known Sarbanes-Oxley act). The other factor concerns a need by the organisation itself, to obtain better insight into, and control over, its own processes. The work in this paper falls in the latter category. As stated in [33], compliance can be checked *before-the-fact* or *after-the-fact*. Manual auditing of event logs constitutes an example of after-the-fact compliance checking. The field of process mining provides tools for automated support for checking of such logs. For example, in [3] LTL checkers are used to automatically detect whether workflow logs violated certain rules. In [33] a further distinction of before-the-fact compliance checking is made, namely whether a process model is compliant by design or verified for compliance after design. The work reported in [33] falls into the former category, as does work reported in e.g. [16, 17, 22]. In these papers, regulations and policies drive business process design or processes are even derived from them. The approach reported in [28] proposes the addition of controls in business processes by making use of control patterns in order to detect violations at runtime. Our approach falls in the category of before-the-fact compliance checking.

In order to keep ever evolving requirements and processes aligned, automated support is required. In this regard two challenges are identified by [20]: 1) how to capture compliance requirements, and 2) how to check compliance of such requirements with respect to a process model.

As regards point 1, capturing compliance requirements, there is a substantial body of work on normative specifications and a comprehensive treatment of this area is outside the scope of this paper. It is perhaps worth pointing out that Sergot et al. [36] already used logic programs to represent and reason over regulations. More recently, Farrell et al. [7] investigated the use of Event Calculus for monitoring contract compliance. In [18] FCL (Formal Contract Language) is adopted to express regulation. As in our case there is no need to represent complex compliance rules, we do not investigate the various formalisms and approaches to representing compliance requirements any further.

As regards point 2, compliance checking in relation to a process model, a number of approaches exist. Ghose and Koliadis [15] present an approach where tasks in BPMN process models are annotated with effects and the propagation of these effects can be studied, in order to determine whether there are contradictions in a model. Their approach, however, is not intended for design time

compliance checking, cannot handle loops in process models, and may suffer from state space explosion. Chopra and Sing [10] consider compliance in the context of multi-agent systems and they investigate whether an agent's behaviour conforms to a protocol. Workflow models, however, tend to be more expressive in terms of control-flow dependencies that can be expressed among tasks, than languages for the specification of protocols. The work described in [23, 24] introduces annotations of tasks that can express whether certain tasks should be considered exclusive (i.e. their instances do not occur together in a trace) or co-dependent (i.e. their instances always occur together in a trace). While this involves compliance checking between a process model and annotations, these annotations cannot directly be used to solve the versioning compliance problem as the latter problem involves the investigation of more complex relations between tasks.

As pointed out in [20], approaches to compliance do not usually offer support for the distinction between an obligation, a permission, and a prohibition. The work in [19] incorporates support for these notions in terms of the language FCL (Formal Contract Language). Governatori and Milosevic [19] point out that compliance concerns the relationship between possible process states and normative statements. While these approaches to compliance checking are very powerful and could be used to establish versioning compliance, to use a compliance based approach, one has to translate the object versioning lifecycle into a set of rules, then compute the reachability graph for the process model, and finally examine the traces of process model for compliance.

In some approaches, compliance checking does not involve annotations of business process models. For example, BPMN-Q [8] can be used to express queries over a repository of BPMN process models. Certain compliance requirements can thus be expressed as queries. In [21] BPEL specifications can be checked with respect to certain compliance requirements through the application of LTL and model checking. Concurrent Transaction Logic is used by Roman and Kifer [32] to determine compliance between a workflow and a contract. None of these approaches is directly suitable for checking versioning compliance.

There is a substantial amount of work that is concerned with the integration of control-flow and data aspects for workflow specification. More recently, work has started to emerge that is concerned with correctness of workflows that involve data [40, 41]. This work treats data in a general sense and considers operations such as create, delete, write, and read. It addresses questions as to whether any task can attempt to read data before another task produced such data or whether data is overwritten by a task without the data has been read first. It is not concerned with compliance between tasks ordering relations in a workflow and version operations in an object lifecycle captured as a state transition diagram. In fact, engineering data is not treated as workflow data in our approach.

Artifact-centric modelling (see e.g. [9]) takes the construction of a business artifact as a starting point rather than a process model. As such it is also concerned with compliance between the lifecycle of an artifact and process models that manipulate this artifact. Artifact-centric modelling is concerned with general artifacts and includes considerations of their structure (something we ab-

stract from in our paper). Due to the general nature of artifact-centric modelling, compliance checking can be computationally expensive and it is not tailored to object versioning.

Data management, access control management and process management are all important for Product Lifecycle Management (PLM) systems. The main PLM systems include Teamcenter [37], Winchill [29] and TiPLM [38] and all these systems provide workflow support (each using their own notation). Data management of these PLM systems follows that of EDM (engineering data management) systems which adopt versioning mechanisms (see [50] for more information about EDM and versioning mechanisms). The versioning relations between versions of business objects are captured in *version graphs* and the progression of particular versions is based on the states that the object may pass through (this progression being state-based is pointed out in [49]). For example, the Windchill and TiPLM systems adopt state transition diagrams to describe the states of business objects and potential state changes between these states, whereas states of business objects and possible state changes between these states are captured through rules in Teamcenter. However, in these PLM systems access control in data management and in process management are disconnected and compatability between access control rules in these two areas is not guaranteed.

The question of how to manage access control mechanism within Product Lifecycle Management (PLM) systems has also attracted some attention. According to Rangan et al. [31], current approaches to managing product data access require the use of complex rules to provide data access rights by considering various factors including the type and the state of the business object, the nature of the PLM process and the organisational context. In [35], Schuh et al. state that many current PLM initiatives focus primarily on isolated aspects, such as document management or parts classification, without the necessary holistic approach to the whole product lifecycle and its underlying processes.

7 Conclusion

In this paper the issue of compliance between workflow management systems and product lifecycle management systems was studied in detail. Syntactical requirements for such compliance were formalised as well as semantical requirements, i.e. requirements that ensure desirable runtime behaviour. It was formally demonstrated that the problem of determining behavioural compliance can be solved through a transformation to workflow nets and using an established and already implemented approach to determining soundness of these nets. Both the syntactical and semantical requirements served as the basis for an analysis plug-in that was developed for the ProM 6.0 framework. This plug-in was used to test properties of a number of real-life models. The application of the plug-in to these models showed the presence of a number of syntactical and semantical errors. One common root cause for these errors that was identified was the inclusion in loops of tasks that were intended to leave an object in a final state.

Naturally there is scope for further work. Ideally, domain experts have access to a single integrated tool that extracts all information from the PLM system and then conducts the analysis. Corrections should then be automatically propagated back to the PLM system. Analysis results can be improved if better support is provided for associating object-state pairs with tasks in workflows.

Acknowledgements.

We are grateful to THsoft InfoTech for providing process models, object versioning lifecycles and task versioning assignment information, all contained in their TiPLM system. Furthermore, we would like to thank Lijie Wen and Guido Governatori for their constructive suggestions, and Eric Verbeek for his help in interfacing our compliance checker with Woflan. This paper is supported by the National Basic Research Program (973 Plan) of China (No. 2009CB320700, 2007CB310802), the 863 High-Tech Development Program of China (No. 2008AA042301, 2007AA040607, 2007AA040602), the National Science Foundation of (No. 90718010, 61003099) and National ICT Australia (NICTA) in the context of the Business Process Compliance project. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. 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.
2. W.M.P. van der Aalst. *Information and Process Integration in Enterprises: Rethinking documents*, chapter 10: Three Good Reasons for Using a Petri-net-based Workflow Management System, pages 161–182. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Norwell, 1998.
3. W.M.P. van der Aalst, H.T. de Beer, and B.F. van Dongen. Process mining and verification of properties: An approach based on temporal logic. In R. Meersman et al., editor, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences CoopIS, DOA, and ODBASE 2005, Agia Napa, Cyprus, October 31 - November 4, 2005, Proceedings, Part I*, volume 3760 of *Lecture Notes in Computer Science*, pages 130–147. Springer-Verlag, 2005.
4. W.M.P. van der Aalst, B.F. van Dongen, C.W. Günther, A. Rozinat, H.M.W. Verbeek, and A.J.M.M Weijters. ProM: The process mining toolkit. In A.K.A. de Medeiros and B. Weber, editors, *Proceedings of the Business Process Management Demonstration Track (BPMDemos 2009), Ulm, Germany, September 8, 2009*, volume 489 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.
5. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.
6. A. Al-Khudair, W.A. Gray, and J.C. Miles. Object-oriented versioning in a concurrent engineering design environment. In B.J. Read, editor, *Advances in Databases,*

18th British National Conference on Databases, BNCOD 18, Chilton, UK, July 9-11, 2001, Proceedings, volume 2097 of *Lecture Notes in Computer Science*, pages 105–125. Springer-Verlag, 2001.

7. D.H.F. Andrew, J.S. Marek, S. Mathias, and B. Claudio. Using the event calculus for tracking the normative state of contracts. *International Journal of Cooperative Information Systems*, 14(2-3):99–129, 2005.
8. A. Awad, G. Decker, and M. Weske. Efficient compliance checking using BPMN-Q and temporal logic. In M. Dumas, M. Reichert, and M.-C. Shan, editors, *Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 2-4, 2008. Proceedings*, volume 5240 of *Lecture Notes in Computer Science*, pages 326–341. Springer-Verlag, 2008.
9. K. Bhattacharya, G. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In G. Alonso, P. Dadam, and M. Rosemann, editors, *Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007, Proceedings*, volume 4714 of *Lecture Notes in Computer Science*, pages 288–304. Springer-Verlag, 2007.
10. A. Chopra and M. Singh. Producing compliant interactions: Conformance, coverage, and interoperability. In M. Baldoni and U. Endriss, editors, *Declarative Agent Languages and Technologies IV, 4th International Workshop, DALT 2006, Hakodate, Japan, May 8, 2006, Selected, Revised and Invited Papers*, volume 4327 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2006.
11. G. Coulouris, J. Dollimore, and M. Roberts. Role and task-based access control in the PerDiS groupware platform. In *RBAC '98: Proceedings of the third ACM workshop on Role-based access control*, pages 115–121. ACM, 1998.
12. K.R. Dittrich and A.L. Raymond. Version support for engineering database systems. *IEEE Transactions on Software Engineering*, 14(4):429–437, 1988.
13. P.H. Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, USA, March 1991.
14. G. Feng, D. Cui, C. Wang, and J. Yu. Integrated data management in complex product collaborative design. *Computers in Industry*, 60(1):48–63, 2009.
15. A. Ghose and G. Koliadis. Auditing business process compliance. In B.J. Krämer, K.J. Lin, and P. Narasimhan, editors, *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings*, volume 4749 of *Lecture Notes in Computer Science*, pages 169–180. Springer, 2007.
16. S. Goedertier and J. Vanthienen. Designing compliant business processes with obligations and permissions. In J. Eder and S. Dustdar, editors, *Business Process Management Workshops, BPM 2006 International Workshops, BPD, BPI, ENEI, GPWW, DPM, semantics4ws*, volume 4103 of *Lecture Notes in Computer Science*, pages 5–14. Springer, 2006.
17. S. Goedertier and J. Vanthienen. Compliant and flexible business processes with business rules. In G. Regev, P. Soffer, and R. Schmidt, editors, *Proceedings of the CAiSE'06 Workshop on Business Process Modelling, Development, and Support BPMDS'06, Luxemburg, June 5-9, 2006*, volume 236 of *CEUR Workshop Proceedings*, pages 94–104. CEUR-WS.org, 2007.
18. G. Governatori, J. Hoffmann, S. Sadiq, and I. Weber. Detecting regulatory compliance for business process models through semantic annotations. In W.M.P. van der Aalst, J. Mylopoulos, N.M. Sadeh, M.J. Shaw, C. Szyperski, D. Ardagna, M. Mecella, and J. Yang, editors, *Business Process Management Workshops, BPM 2008*

International Workshops, Milano, Italy, September 1-4, 2008. Revised Papers, volume 17 of *Lecture Notes in Business Information Processing*, pages 5–17. Springer, 2009.

19. G. Governatori and Z. Milosevic. A formal analysis of a business contract language. *International Journal of Cooperative Information Systems*, 15(4):659–685, 2006.
20. J. Hoffmann, I. Weber, and G. Governatori. On compliance checking for clausal constraints in annotated process models. *Information Systems Frontiers*, 11(5):1–23, 2009.
21. Y. Liu, S. Müller, and K. Xu. A static compliance-checking framework for business process models. *IBM Systems Journal*, 46(2):335–361, 2007.
22. R. Lu, S.W. Sadiq, and G. Governatori. Compliance aware business process design. In A.H.M. ter Hofstede, B. Benatallah, and H.Y. Paik, editors, *Business Process Management Workshops*, volume 4928 of *Lecture Notes in Computer Science*, pages 120–131, Berlin, Heidelberg, 2007. Springer-Verlag.
23. L.T. Ly, S. Rinderle, and P. Dadam. Semantic correctness in adaptive process management systems. In S. Dustdar, J.L. Fiadeiro, and A.P. Sheth, editors, *BPM'06: Proceedings of the 4th international conference on Business process management*, volume 4102 of *Lecture Notes in Computer Science*, pages 193–208, Berlin, Heidelberg, 2006. Springer-Verlag.
24. L.T. Ly, S. Rinderle, and P. Dadam. Integration and verification of semantic constraints in adaptive process management systems. *Data & Knowledge Engineering*, 64(1):3–23, 2008.
25. J.C. Miles, W.A. Gray, T.W. Carnduff, I. Santoyridis, and A. Faulconbridge. Versioning and configuration management in design using CAD and complex wrapped objects. *Artificial Intelligence in Engineering*, 14(3):249–260, 2000.
26. R. Milner. *Communication and Concurrency.* Prentice Hall, London, England, 1989.
27. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
28. K. Namiri and N. Stojanovic. Pattern-based design and validation of business process compliance. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I*, volume 4803 of *Lecture Notes in Computer Science*, pages 59–76. Springer-Verlag, 2007.
29. PTC. Configuring a Windchill ProjectLink 7.0 Workflow and Lifecycle. `http://www.ptc.com/carezone/tutorials/files/configuring_windchill_projectlink_workflow_lifecycle.pdf`. Accessed Sep 2010.
30. Z.M. Qiu and Y.S. Wong. Dynamic workflow change in PDM systems. *Computers in Industry*, 58(5):453–463, 2007.
31. R.M. Rangan, S.M. Rohde, R. Peak, B. Chadha, and P. Bliznakov. Streamlining product lifecycle processes: A survey of product lifecycle management implementations, directions, and challenges. *Journal of Computing and Information Science in Engineering*, 5(3):227–237, 2005.
32. D. Roman and M. Kifer. Reasoning about the behavior of semantic web services with concurrent transaction logic. In C. Koch, J. Gehrke, M.N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C.Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E.J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 627–638. ACM, 2007.

33. S. Sadiq, G. Governatori, and K. Namiri. Modeling control objectives for business process compliance. In G. Alonso, P. Dadam, and M. Rosemann, editors, *Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007, Proceedings*, volume 4714 of *Lecture Notes in Computer Science*, pages 149–164. Springer-Verlag, 2007.
34. R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
35. G. Schuh, H. Rozenfeld, D. Assmus, and E. Zancul. Process oriented framework to support PLM implementation. *Computers in Industry*, 59(2-3):210–218, 2008.
36. M.J. Sergot, F. Sadri, R.A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory. The British Nationality Act as a logic program. *Communications of the ACM*, 29(5):370–386, 1986.
37. SIEMENS. Teamcenter manufacturing user's and administrator's manual. `http://www.cadfamily.com/download/CAD/Teamcenter8/TeamcenterManufacturingUserAndAdministratorManual.pdf`. Accessed Sep 2010.
38. THsoft InfoTech. TiPLM-product lifecycle management system of THsoft InfoTech (Chinese version). `http://www.thit.com.cn/chanpinshijie/TiPLM.htm`. Accessed Sep 2010.
39. W.F. Tichy. RCS - a system for version control. *Software: practice & experience*, 15(7):637–654, 1985.
40. N. Trčka. Workflow soundness and data abstraction: Some negative results and some open issues. In *APNOC'09: International Workshop on Abstractions for Petri Nets and Other Models of Concurrency, Paris, France, June 22, 2009*, pages 19–25, 2009.
41. N. Trčka, W.M.P. van der Aalst, and N. Sidorova. Data-flow anti-patterns: Discovering data-flow errors in workflows. In P. van Eck, J. Gordijn, and R. Wieringa, editors, *Advanced Information Systems Engineering, 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June 8-12, 2009. Proceedings*, volume 5565 of *Lecture Notes in Computer Science*, pages 425–439. Springer-Verlag, 2009.
42. J.M.E.M. van der Werf. *Compositional design and verification of component-based information systems.* PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2011.
43. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:555–600, 1996.
44. H.M.W. Verbeek. *Verification of WF-nets.* PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2004.
45. H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing workflow processes using Woflan. *Computer Journal*, 44(4):246–279, 2001.
46. H.M.W. Verbeek and W.M.P. van der Aalst. Woflan 2.0: A Petri-net-based workflow diagnosis tool. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 26-30, 2000, Proceeding*, volume 1825 of *Lecture Notes in Computer Science*, pages 475–484. Springer Berlin / Heidelberg.
47. J. Wainer, P. Barthelmess, and A. Kumar. W-RBAC-a workflow security model incorporating controlled overriding of constraints. *International Journal of Cooperative Information Systems*, 12(4):455–485, 2003.
48. H. Wang and Y. Zhang. CAD/CAM integrated system in collaborative development environment. *Robotics and Computer-Integrated Manufacturing*, 18(2):135–145, 2002.

49. B. Westfechtel. Integrated product and process management for engineering design applications. *Integrated Computer-Aided Engineering*, 3(1):20–35, 1996.
50. B. Westfechtel and R. Conradi. Software configuration management and engineering data management: Differences and similarities. In B. Magnusson, editor, *System Configuration Management, ECOOP'98 SCM-8 Symposium, Brussels, Belgium, July 20-21, 1998, Proceedings*, volume 1439 of *Lecture Notes in Computer Science*, pages 95–106. Springer, 1998.
51. B. Westfechtel, B.P. Munch, and R. Conradi. A layered architecture for uniform version management. *IEEE Transactions on Software Engineering*, 27(12):1111–1133, 2001.
52. M.T.K. Wynn. *Semantics, Verification, and Implementation of Workflows with Cancellation Regions and OR-joins.* PhD thesis, Queensland University of Technology, Brisbane, Australia, 2006.
53. C. Yuan. *Principles of Petri Nets (Chinese version).* Publishing House of Electronics Industry, Beijing, China, 1998.
54. H. Zha, W.M.P van der Aalst, J. Wang, L. Wen, and J. Sun. Verifying workflow processes: a transformation-based approach. *Software and Systems Modeling*, 10(2):253–264, 2010.