1. Introduction

Requirements specifications are usually documented in the form of natural language (NL) by requirements analysts (RAs) at the early stage of software development. Commonly, requirements are described by use cases, scenarios, or even narrative sentences. However, such human-centric NL-based requirements are usually error-prone and inaccurate, particularly when the number of requirements is extremely large [1].

Requirements quality is defined in terms of the 3Cs: consistency, correctness and completeness [2], [3]. In other words, the major classes of defects in requirements specifications are inconsistency [4], incorrectness [5], and incompleteness [6]. Inconsistency occurs when a specification contains conflicting or contradictory descriptions of the expected behavior of the system-to-be [7]. Incorrectness occurs when captured requirements do not reflect actual needs of stakeholders [2]. Incompleteness may occur when there are missing functions, constraints, entities in documented requirements [2]. Inconsistency, incorrectness and incompleteness are quality problems that permeate all aspects of software development. If these quality problems are left to the design and implementation phase, it will become time-consuming to solve them [4].

Model checking techniques have emerged with respect to the treatment of requirements quality. Researchers propose to build formalized models for NL-based requirements with temporal logics, such as CTL formula, and then use model checking techniques to automatically verify properties of formalized models [8], [9]. A common problem for existing model checking techniques is the unacceptable checking time when nondeterministic automata are generated from requirements specifications. The checking time will extremely increase when the problem space becomes large [10].

In this paper, we propose a pattern matching method to detect requirements inconsistency, incorrectness and incompleteness, and to consequently improve requirements quality. Our approach first transforms NL-based requirements to business process models and then adopts workflow patterns to check the quality of business process models. Compared to temporal logics used in model checking, patterns are less complicated and much easier to learn for users [11]. One challenge of pattern matching is the selection of appropriate patterns. If the selected patterns are not generic enough or not familiar for users, the usability of this approach will decrease. In this work, we choose workflow patterns due to their popularity in both academia and industry.

This paper is organized as follows: Section 2 introduces the background of workflow patterns and the requirements of GETS; Section 3 presents the whole process of our approach and the framework of a prototype, following by Sect. 4 that uses an example of GETS to illustrate our approach. Section 5 compares our work with related work. Finally Sect. 6 discusses our work and concludes this paper.

2. Background

2.1 Workflow Patterns

A workflow is an enactment of a business process that is described in terms of a flow of work through an organization [12]. Workflow patterns are developed by workflow patterns initiative that aims to provide a collection of generic recurring process constructs. Due to its high reusability, workflow patterns have a wide applicability in both academia and industry [13]. Figure 1 shows the popularity of workflow patterns through comparing with other requirements patterns, such as event patterns [11], essential use cases [2], problem frames [14] and object system models [15] in Google trends (http://www.google.com/trends/). We can see from Fig. 1 that workflow patterns have received...
After the trade is successfully printed, the system will notify regulations and, if legal, publishes the trades to the public. The printing agent verifies the printing data according to the financial printing agent for verification and publishing. The printing agent verifies the printing data according to the financial regulations and, if legal, publishes the trades to the public. After the trade is successfully printed, the system will notify the trader with the new trading information, including the traded price and trading sizes, and so on. During the process of matching orders, the operator first needs to price the new order based on the real-time market price information; next, cross the new order with other opposite-side orders that have the same equity symbol. Once orders are crossed, the system executes matched-orders to trades and updates relevant order information". In most cases, a requirement description may include requirements at different levels. For example, in the above narrative description, Order Trading is the top-level requirement whereas Order Matching is the requirements at a lower level. Thanks to the natural decomposition of business process models, the different levels of details can also be modeled.

3. Applying Workflow Patterns to Requirements Modeling and Analysis

This section introduces the process of the requirements quality improvement approach based on workflow patterns matching.

3.1 Overview

Figure 2 provides a holistic overview of our approach. At the initial stage, RAs collect requirements from stakeholders and express them using natural language. A domain object library needs to be built first for all domain objects, and then manipulations performed on each object and the states transitions caused by the manipulations (Fig. 2). The action language [24] is used to represent the start state, manipulation and end state in the domain object library whereas propositional logic [4] is used to represent the relationships between states.

Action language is a language designed for modeling actions and their effects [24]. The semantics of action language are based on the assumption that “things remain the same until something happens to make them change” [25]. This assumption is the same as our assumption of representing requirements in WPPL. Propositional logic is the system of logic with the simplest semantics for the use of formalizing properties of structures [26]. Propositions can be joined using logical connectives to make new propositions, such as logical and, logical or and so on. The reason why we choose action language and propositional logic is that both of them prove adequate for expressing key elements of high-level requirements (e.g. actions, objects, states and their relationships) and make the conflicts checking simple by removing unnecessary details [4]. Section 3.2 will give some examples of action language and propositional logic.

The NL requirements are then subjected to two-step transformations: 1) transformation into a set of process models; 2) translation into a set of formalized requirements models in WPPL. We noted that the first step is not mandatory when it is easy to extract the process information from the NL requirement. The transformation to WPPL requires both information provided by business process models and

![Fig. 1](Image) The popularity of workflow patterns comparing with other patterns.
These formalized requirements models are then treated as the input of pattern matching for the 3Cs analysis. The 3Cs analysis is performed based on process conflicts checking rules that are defined to detect conflicts between functional requirements in requirements models. Notice that the detection of conflicts between non-functional requirements needs to adopt different mechanisms, such as the NFR framework [27], QARCC [28], which is out of the scope of this paper. If any inconsistency, incorrectness or incompleteness is found, a detailed report is presented to the users (e.g. RAs), providing them with opportunities to examine the situation. Then the specification is revised accordingly.

3.2 The Semantics of WPPL

A WPPL specification describes a requirement in the form of a business process which is composed of workflow patterns. The description of each requirement is structured in two layers. The outer layer describes the key properties of requirements, such as input data, output data, pre-condition, post-condition, whereas the inner layer expresses the dynamic structure of the requirement. The BNF grammar of WPPL is given as follows.

```bash
/*The outer layer*/
<requirement>::= <name> <input> <output>
<precondition> <postcondition> <process>
:name ::= name Action
<input> ::= input Entity+
<output> ::= output Entity+
<precondition> ::= precondition State+
<postcondition> ::= postcondition State+
/*The inner layer*/
<process>::=<pattern> | <pattern><process>
<pattern>::=<pattern_name><action_set>
<action_set>::= Action, Action | Action, <action_set>
<pattern_name>::= seq | par_split | sync | excl_choice |
```

Figure 3 shows an excerpt of the outer layer of the WPPL specification of Order Trading requirement. Each requirement has a unique name for itself. The name is defined with the action form similar to action language [24], such as action(entity1, entity2, ...). For example, the name for Order Trading is trade(order). The outer layer treats a requirement as a black box and includes input, output, pre-condition and post-condition, which are required to analyze the 3Cs of requirements. Precondition and post-condition are states, such as state(entity), which represents an entity’s state. For example, the precondition for trade(order) is that the order is prepared, i.e. prepared(order), whereas the post-condition for trade(order) is either the order is traded, i.e. traded(order), or the order has been added to the order_book, i.e. updated(order_book). Propositional logic is used to represent the relationships between states. In Fig. 3, traded(order) and updated(order_book) are joined with the logical or, denoted as ∨.

The inner layer of a WPPL specification consists of process description that describes the inner properties of the requirement. Figure 4 presents an excerpt of the inner layer of the WPPL specification of Order Trading requirement. Each requirement can be treated as one or more processes consisting of multiple actions through a sequence of control flows, which are represented by workflow patterns. Due to
space limitations, in the main body of this paper, we will demonstrate three basic workflow patterns: \texttt{seq} (sequence), \texttt{par\_split} (parallel split) and \texttt{excl\_choice} (exclusive choice). The reader may refer to the Appendix for other workflow patterns.

Take Order Trading as an example, the process is represented as \texttt{seq(enter(order), match(order))}; \texttt{excl\_choice(match(order), add(order, order\_book), print(trade))}; \texttt{seq(print(trade), notify(trader))}.

In order to provide a holistic view for all elements in WPPL, we develop a meta-model as shown in Fig. 5. In this meta-model, as described in WPPL, a \textit{requirement} consists of \textit{input}, \textit{output}, \textit{precondition}, \textit{postcondition} and \textit{process}. A process is composed of workflow \textit{patterns}. Patterns consist of at least two \textit{actions}. As an action can also be considered as a process, in some cases, we can say that a requirement is an action.

3.3 Checking Process Conflicts of Requirements

Process conflicts refer to contradictions between assumptions and principles that underlie a business process design and determine its sequence of activities or events. We define a set of process conflicts checking rules according to control flows of each workflow pattern. Checking process conflicts can be divided into the following steps:

Step 1. Model initial textual requirements (e.g. scenarios, use cases) with business process modeling approach. We recommend UML 2.0 Activity Diagram [29] as it is the most famous modeling language to model processes and workflows.

Step 2. Transform original requirements descriptions and UML activity models to WPPL. First, the transformation starts from the outer layer, i.e. identifying precondition, post-condition, input and output in each requirement, and then deals with the inner layer, i.e. transform requirements to processes using WPPL.

Step 3. Check the precondition and post-condition of each requirement to determine whether there is coexistence of different or conflicting states. For example, the post-condition of \textit{trade(order)} is \textit{traded(order)} \land \neg\textit{traded(order)}, indicating that there are conflicting requirements in \textit{trade(order)}.

Step 4. Match each process against workflow patterns and check process conflicts against a set of checking rules. Any conflict is found, report them to RAs. Considering the reuse of some requirements, we divide the 3Cs into two levels: 1) consistent, correct, and complete (3Cs); and 2) strictly consistent, strictly correct, strictly complete (S-3Cs). We will focus on the 3Cs checking rules in the following paragraphs with the description of each pattern, our interpretation, the UML representation, WPPL representation and the 3Cs checking rules of each pattern. The difference between the 3Cs and S-3Cs checking rules will be introduced in Appendix. Two sets of process conflicts checking rules are defined and users can select one level according to their actual requirements.

When a requirement description comprises multiple requirements at different levels, RAs need to 1) transform each requirement in the description into a business process model, 2) identify each requirement’s outer layer and inner layer, 3) ensure that the input and precondition of the parent requirement is the same as the input and precondition of the first sub-requirement; Besides, the output and post-condition of the parent requirement is the same as the output and post-condition of the last sub-requirements, and 4) go through step 3 to step 4 to check the conflicts among requirements at the same level.

· Sequence Pattern
  
  Description: A task in a process in enabled after the completion of a preceding task in the same process [16].

  Interpretation: An action \texttt{a} must terminate before \texttt{b} can begin.

  UML Activity Diagram: see Fig. 6a.

  WPPL example: \texttt{seq(a, b)}.
The 3Cs Checking Rules:

\textit{SEQR1:} There must exist sharing data between the output of the preceding action and the input of the subsequent action, i.e. \(a.\text{output} \cap b.\text{input} \neq \emptyset\).

\textit{SEQR2:} There must exist at least one state between the post-condition of the preceding action and the pre-condition of the subsequent action, i.e. \(a.\text{postcondition} \cap b.\text{precondition} \neq \emptyset\).

- Parallel Split Pattern
  
  Description: The divergence of a branch into two or more parallel branches each of which execute concurrently [16].
  
  Interpretation: Following the completion of an action \(a\), action \(b\) can begin \((i = 1 \ldots n)\). There are no interdependencies between any two \(b\).
  
  UML Activity Diagram: see Fig. 6.b.
  
  WPPL example: \texttt{par\_split}(a, b_1, \ldots , b_n)
  
  The 3Cs Checking Rules:
  
  \textit{PSR1:} There must exist sharing data between the output of the preceding branch action and the input of any parallel branch action, i.e. \(\forall b_i (i = 1 \ldots n), a.\text{output} \cap b_i.\text{input} \neq \emptyset\).
  
  \textit{PSR2:} There must exist at least one state between the post-condition of the branch action and the pre-condition of any parallel branch action, i.e. \(\forall b_i (i = 1 \ldots n), a.\text{postcondition} \cap b_i.\text{precondition} \neq \emptyset\).

- Exclusive Choice Pattern
  
  Description: The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches [16].
  
  Interpretation: Following the completion of the incoming branch action \(a\), only one of \(b\) is completed.
  
  UML Activity Diagram: see Fig. 6.c.
  
  WPPL example: \texttt{excl\_choice}(a, b_1, \ldots , b_n).
  
  3Cs Checking Rules:
  
  \textit{ECR1:} There must exist sharing data between the input of any outgoing branch action and the output of incoming branch action, i.e. \(\forall b_i (i = 1 \ldots n), a.\text{input} \cap b_i.\text{output} \neq \emptyset\).
  
  \textit{ECR2:} There must exist at least one state between the post-condition of the incoming branch action and the pre-condition of any outgoing branch action, i.e. \(\forall b_i (i = 1 \ldots n), a.\text{postcondition} \cap b_i.\text{precondition} \neq \emptyset\).
  
  \textit{ECR3:} Any two pre-condition of the outgoing branch actions should not be the same, i.e. \(\forall b_i, b_j (i, j = 1 \ldots n), i \neq j, b_i.\text{precondition} \neq b_j.\text{precondition}\). The pre-condition is the guard condition. It is impossible to appear such circumstance when guard conditions of two alternative actions are identical.
  
  \textit{ECR4:} Any two post-condition of the outgoing branch actions should not be the same, i.e. \(\forall b_i, b_j (i, j = 1 \ldots n), i \neq j, b_i.\text{postcondition} \neq b_j.\text{postcondition}\). The operations performed by \(b_i\) and \(b_j\) must be different, so are the post-conditions of \(b_i\) and \(b_j\).

3.4 Handling Requirements Defects

The reported process conflicts may be caused by inconsistency (conflicting requirements), incorrectness (wrongly documented requirements) or incompleteness (missing requirements).

RAs maintain an issue log to document all of these inconsistent, incorrect or incomplete requirements and the priority of each issue. The handling process starts from issues with higher priority.

If RAs attribute the conflict on one requirement, i.e. contradictory elements situate in one requirement, RAs need to further check whether the two outer layers contain contradictory elements, e.g. the two pre-conditions cannot coexist. If they are contradictory with each other, RAs need to select one element as the only correct element.

If RAs attribute the conflict on different requirements, they need to take the following four steps to resolve these conflicts:

1) Identify the conflicting pattern;

2) Check any element missing in the requirements related to the pattern;

3) Check any error existing in the requirements related to the pattern;

4) Check any inconsistency existing in the requirements related to the pattern.

First, RAs also need to ensure that the detected conflicts are not caused by incompleteness and incorrectness by adding missing requirements and correcting error requirements. Then if conflicts still exist, they need to find the reason for the inconsistency, and determine to take which strategy to handle this inconsistency. According to Nuseibe et al. [7], there are a number of inconsistency handling strategies that can be chosen. RAs can ignore the inconsistency completely, tolerate it for a while, or resolve it immediately, with respect to the type of the inconsistency.
3.5 Prototype

A prototype was designed to support our approach. The architecture of the prototype is shown in Fig. 7, which consists of the following major components: 1) a domain object manager, which is responsible for adding, deleting, and editing the domain objects; 2) a WPPL editor, which edits WPPL-based requirements specifications; 3) a WPPL analyzer, which interprets the WPPL; 4) a pattern matcher, which is responsible for matching the appropriate pattern for the WPPL-based requirement; 5) a rules checker, which checks the 3Cs against the corresponding rules of the matched pattern; 6) a reporter which reports defects to RAs; 7) a requirements manager which manages the WPPL-based requirements; and 8) a rules manager, which is responsible for adding, deleting, and editing the 3Cs checking rules.

4. Illustration

The correctness of the 3Cs checking rules has been validated by RAs participating in GETS. We have applied our approach and the prototype to the requirements analysis of GETS’s use case specification, including 70 use cases. A real world Order Trading requirement specification of GETS [22] is used as an example to aid our description and illustration of our approach in this section. The correct description of Order Trading is shown in Sect. 2, and the relevant process model is shown in Fig. 8. In order to illustrate our approach, we present another requirement specification of Order Trading and its two sub-requirements — Order Entry and Order Crossing which were documented by another RA with defects by means of use cases (Fig. 9 a).

According to the steps presented in Sect. 3.3, we first need to construct the new process model according to the revised requirements description, as shown in Fig. 9 b. We can see from Fig. 9 b that Order Trading (i.e. trade(order)) is composed of four actions — enter(order), cross(order), notify(trader) and add(order, order_book), which are further described in WPPL by RAs as shown in Fig. 9 c.

As there are no contradictory elements in all requirements, we forward to match the process of Order Trading with workflow patterns and identify two workflow patterns are used in this process, i.e. sequence pattern and exclusive choice pattern. Process conflicts are checked against relevant the 3Cs Checking rules of these two patterns and find there are two conflicts in this requirement specification: 1) cross(order) should not follow enter(order), as there are no sharing state between the precondition of cross(order) and the post-condition of enter(order); 2) notify(trader) is not one of the outgoing branch of cross(order), as there are neither sharing state between the post-condition of cross(order) and the precondition of notify(trader), nor

---

### Table 1: Use case specifications

<table>
<thead>
<tr>
<th>Name</th>
<th>Order Trading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Trade order is to trade orders with the same symbol into a trade.</td>
</tr>
<tr>
<td>Actor</td>
<td>Trader, Operator, Printing Agent</td>
</tr>
<tr>
<td>Priority</td>
<td>High</td>
</tr>
<tr>
<td>Precondition</td>
<td>The initial order is prepared.</td>
</tr>
<tr>
<td>Post-condition</td>
<td>The new order is traded with the opposite-side order or the new order is added to the order book.</td>
</tr>
<tr>
<td>Process-flow</td>
<td>P1: The trader enters the initial order to the system (see Order Entry use-case). P2: The operator crosses the priced order with other opposite-side orders that have the same equity symbol. If the new order is successfully matched, the system generates the trade for the matched order (see Order Crossing use-case). P3: The system notifies the trader with the new trading information including the traded price, trading sizes and trading time.</td>
</tr>
<tr>
<td>Alternative-flow</td>
<td>A1. Same as P1. A2. The operator crosses the priced order with other opposite-side orders that have the same equity symbol. If the new order cannot be matched, the system saves the new order in the order book and waits to match it with other subsequent orders (see Order Crossing use-case).</td>
</tr>
<tr>
<td>Exceptions</td>
<td>None</td>
</tr>
</tbody>
</table>

### Table 2: Order Entry

<table>
<thead>
<tr>
<th>Name</th>
<th>Order Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Order Entry is to submit the initial order to the system.</td>
</tr>
<tr>
<td>Actor</td>
<td>Trader</td>
</tr>
<tr>
<td>Priority</td>
<td>High</td>
</tr>
<tr>
<td>Precondition</td>
<td>The initial order is prepared.</td>
</tr>
<tr>
<td>Post-condition</td>
<td>The new order is added to the order book.</td>
</tr>
<tr>
<td>Process-flow</td>
<td>P1: The trader submits the initial order to the system. P2: The system checks the validity of the initial order. P3: If the order is valid, the system saves the order in the order book.</td>
</tr>
<tr>
<td>Alternative-flow</td>
<td>A1. Same as P1. P2. If the order is invalid, the system saves the order as an invalid order.</td>
</tr>
<tr>
<td>Exceptions</td>
<td>None</td>
</tr>
</tbody>
</table>

---

### Diagram 1: UML Activity Diagrams for Order Trading and Order Matching

- Fig. 8: UML Activity Diagrams for Order Trading and Order Matching.
- Fig. 9 a: Use case specifications; (b) UML Activity Diagram; (c) WPPL specification for new Order Trading requirement.
sharing data between the output of \texttt{cross(order)} and the input of \texttt{notify(trader)}. Then those conflicts are reported to RAs, and the reasons for the two conflicts have been identified. The first conflict is caused by the incorrect requirement whereas the second conflict is caused by missing requirements. RAs then add \texttt{print(trade)} before \texttt{notify(trader)} and substitute \texttt{match(order)} with \texttt{cross(order)} and repeat the checking process of Order Trading until no defect can be found.

According to the application of our approach, we found that our approach especially fits for the analysis of requirements documented in the form of use cases. As requirement analysts specify use cases in terms of precondition, post-condition, input, output and the process flows, use cases are comfortably matched with the components of WPPL.

5. Related Works

Patterns have been attracting more and more attention in the Requirement Engineering (RE) community by analyzing and validating requirements [30],[31]. In this section, we compare our approach with some of the closely related pattern-based requirements analysis approaches for improving requirements quality.

Problem frames are a systematic approach to the decomposition of problems that allows requirement analysts to relate requirements, domain properties, and machine specifications [14], which are regarded as a type of requirement patterns. Laney et al. [19] developed a systematic approach to composing inconsistent requirements using problem frames. They addressed the composition problem for inconsistent requirements and proposed to resolve inconsistencies by introducing Composition Frames including four options. Both Laney et al.’s approach and our approach build on the premise that requirements are composed by sub-requirements. However, Laney et al. focused themselves on the resolution of inconsistency whereas we put emphasis on both the detection and the resolution of inconsistency, incorrectness and incompleteness.

Kamalrudin et al. [2] reported a technique to improve requirements quality using Essential Use Case (EUC) Interaction Patterns, and developed a CASE tool to support their work. This technique first transformed natural language requirements to semi-formal requirements, i.e. EUC, and extracted abstract interactions from Essential Use Case to derive EUC models. Then EUC models were checked against its matched EUC interaction patterns. The process of EUC technique is similar to our approach. Our approach leveraged workflow patterns that have been widely used in the world, whereas EUC technique offers a visual and friendly user interface to RAs.

6. Discussions and Conclusions

This paper proposes a systematic approach that improves the quality of natural language requirements specifications using workflow patterns. Our approach, which starts from the natural language requirements, has the ability to identify possible inconsistency, incorrectness and incompleteness in requirements. We use a real world financial application example to illustrate our approach.

Our approach is novel in introducing one of the most widely used patterns — workflow patterns — to requirements quality improvement. We codify a set of the 3Cs checking rules to each workflow pattern. Due to the popularity of workflow patterns in both researchers and vendors as well as the behavioral property of requirements, it is easy to match natural language requirements with workflow patterns. The second contribution of our approach is the definition of a novel requirements specification language — WPPL, which provides basis for the formalization of natural language requirements. WPPL has strengths especially when the requirement is documented in the form of use cases. In addition, WPPL has good extensibility: it can be used not only in our approach; rather, users can adopt WPPL to specify requirements if they want to define other checking rules for workflow patterns. Yet, WPPL still has weaknesses. For example, in the case of use cases, the transformation to WPPL is easy due to the fact that the structure of use cases conforms to the structure of WPPL. Nevertheless, in case of narrative descriptions, the transformation may require a lot of manual interventions. Our future work will investigate some text mining techniques and identify their synergies with our approach.

In conclusion, through presentation, illustration and discussion, this paper has exposed both strengths and weakness of the pattern matching approach and has shown that this approach is a useful for improving requirements quality. In the future, we will complete the checking rules for other workflow patterns.

Acknowledgments

This work was jointly supported by the National 973 Fundamental Research Development Program of China under the Grant 2009CB320701, the Fundamental Research Funds for the Central Universities, the National Natural Science Foundation of China under Grant No. 61103032, and the State Street Cooperation in USA.

References


Appendix

Both the 3Cs and S-3Cs checking rules for workflow patterns are defined in Table A-1. As workflow patterns involving multiple instances are runtime patterns, it is infeasible to define checking rules for these patterns at the requirements analysis phase. Cancel Task and Cancel Case pattern only involves one major action or task in the process, it is also unnecessary to define checking rules for these two patterns. Therefore, we only present checking rules for other 13 workflow patterns below.

The 3Cs rules and S-3Cs rules differ in the situation under which they are applied. Take sequence pattern for example, when both the preceding action or the subsequent action can be reused in the composition of other requirements, i.e. b is not the only subsequent action of a, or a is not the only preceding action of b, the 3Cs rules should be applied. For example, if there is another requirement which can be represented as seq(a, c), then we should select the 3Cs rules to check process conflicts; otherwise, the S-3Cs rules should be selected. Determining the level of checking rules of other workflow patterns is similar to that of sequence pattern.
Table A-1 The 3Cs checking rules for workflow patterns.

<table>
<thead>
<tr>
<th>Workflow Pattern</th>
<th>Interpretation</th>
<th>Diagram</th>
<th>WPPL Example</th>
<th>3Cs Checking Rules and Strictly-3Cs Checking Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>An action a must terminate before b can begin</td>
<td><img src="image" alt="Diagram" /></td>
<td>seq(a,b)</td>
<td>SEQR1: a.output b.input \emptyset. SEQR2: a.postcondition b.precondition \emptyset. S-SEQR3: a.precondition b.precondition.</td>
</tr>
<tr>
<td>Parallel Split</td>
<td>Following the completion of a, action b can begin (i=1...n). There are no interdependencies between any two b_i.</td>
<td><img src="image" alt="Diagram" /></td>
<td>par_split(a,b_i,...,b_n)</td>
<td>PSR1: ( \forall b_i (i=1...n), a.output b_i.input \emptyset. ) PSR2: ( \forall b_i (i=1...n), a.postcondition b_i.preconditions \emptyset. ) S-PSR1: a.output b_i.input (i=1...n). S-PSR2: a.precondition b_i.precondition (i=1...n).</td>
</tr>
<tr>
<td>Synchronization</td>
<td>The subsequent branch action b can begin once all input branch actions a_i (i=1...n) complete. There are no interdependencies between every two a_i.</td>
<td><img src="image" alt="Diagram" /></td>
<td>sync(a_i,...,a_n,b)</td>
<td>SYN1: ( \forall a_i (i=1...n), a_i.output b.input \emptyset. ) SYN2: ( \forall a_i (i=1...n), a_i.postcondition b.preconditions \emptyset. ) S-SYN1: a_i.output b.input (i=1...n). S-SYN2: a_i.precondition b.precondition (i=1...n).</td>
</tr>
<tr>
<td>Exclusive Choice</td>
<td>Following the completion of the incoming branch action a, only one of b_i is completed</td>
<td><img src="image" alt="Diagram" /></td>
<td>excl_choice(a,b_i,...,b_n)</td>
<td>ECR1: ( \forall b_i (i=1...n), a.output b_i.input \emptyset. ) ECR2: ( \forall b_i (i=1...n), a.postcondition b_i.preconditions \emptyset. ) ECR3: ( \forall b_i b_j (i,j=1...n), \neg \exists b_k (i=k=j). ) ECR4: ( \forall b_i b_j (i,j=1...n), b_i.precondition b_j.precondition. ) S-ECR1: b_i.output b_j.input (i=j). S-ECR2: b_i.precondition b_j.precondition. S-ECR3: ( \forall b_i b_j (i,j=1...n), \neg \exists b_k (i=k=j). ) S-ECR4: ( \forall b_i b_j (i,j=1...n), b_i.precondition b_j.precondition. )</td>
</tr>
<tr>
<td>Simple Merge</td>
<td>Once any incoming branch action a_i (i=1...n) terminates, the subsequent branch action b_i start. It will never be the case that more than one a_i executes.</td>
<td><img src="image" alt="Diagram" /></td>
<td>merge(b_i,...,b_n,a)</td>
<td>S-MR1: ( \forall a_i (i=1...n), a_i.output b_i.input \emptyset. ) S-MR2: ( \forall a_i (i=1...n), a_i.postcondition b_i.preconditions \emptyset. ) S-MR3: ( \forall a_i a_j (i,j=1...n), \neg \exists a_k (i=k=j). ) S-MR4: a_i.postcondition a_j.postcondition. S-SMR1: a_i.output b_i.input (i=j). S-SMR2: a_i.postcondition a_j.postcondition. S-SMR3: ( \forall a_i a_j (i,j=1...n), \neg \exists a_k (i=k=j). ) S-SMR4: a_i.postcondition a_j.postcondition.</td>
</tr>
<tr>
<td>Multiple Choice</td>
<td>Upon the completion of the incoming branch action a, either one of b_i (i=1...n) or more of them start</td>
<td><img src="image" alt="Diagram" /></td>
<td>mult_choice(a,b_i,...,b_n)</td>
<td>MCR1: ( \forall b_i (i=1...n), a.output b_i.input \emptyset. ) MCR2: ( \forall b_i (i=1...n), a.postcondition b_i.preconditions \emptyset. ) S-MCR1: b_i.output b_i.input (i=1...n). S-MCR2: b_i.postcondition b_i.precondition (i=1...n).</td>
</tr>
<tr>
<td>Synchronizing Merge</td>
<td>When there is only one active incoming branch b_i, the subsequent branch action c will be executed at once, regardless of other branches; when there are multiple active actions b_i, c needs to wait until all active b_i complete.</td>
<td><img src="image" alt="Diagram" /></td>
<td>sync_merge(b_i,...,b_n,c)</td>
<td>SYMR1: ( \forall b_i (i=1...n), a.output c.input \emptyset. ) SYMR2: ( \forall b_i (i=1...n), a.postcondition c.preconditions \emptyset. ) S-SYMR1: b_i.output c.input (i=1...n). S-SYMR2: b_i.postcondition c.precondition (i=1...n).</td>
</tr>
<tr>
<td>Workflow Pattern</td>
<td>Interpretation</td>
<td>Diagram</td>
<td>WPPL Example</td>
<td>3Cs Checking Rules and Strictly-3Cs Checking Rules</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>--------------------------</td>
<td>--------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
</tbody>
</table>
| **Multiple Merge**       | Every completion of the incoming branch action $b_i$ ($i=1...n$) will trigger the subsequent branch action $c$ to execute, maybe more than once. Only the first completed incoming branch action $b_i$ will trigger the subsequent branch action $c$ to execute. Following the completion of $a$, either one of $b_i$ ($i=1...n$) is completed. The choice of $b_i$ is based on an event that occurs during the process. Following the completion of $a$, a series of branches $b_i$ ($i=1...n$) is executed in an arbitrary order. No two activities are executed at the same time. The completion of $a$ in one process is required before another action $b$ in another process can start. | ![Diagram](attachment:Workflow_Diagram.png) | $\text{mult_merge}(b_1,...,b_n,c)$ | $\text{MMR1: } \forall b_i (i=1...n), b, output(c, input) \oplus 0.$  
$\text{MMR2: } \forall b_i (i=1...n), b, postcondition(c, precondition) \oplus 0.$  
$\text{S-MMR1: } \cup b_i, output(c, input) (i=1...n).$  
$\text{S-MMR2: } \cup b_i, postcondition(c, precondition) (i=1...n).$  |
| **Discriminator**        | Cannot represented with Activity Diagram                                       | ![Diagram](attachment:Workflow_Diagram.png) | $\text{disc}(b_1,...,b_n,c)$ | $\text{DSR1: } \forall b_i (i=1...n), b, output(c, input) \oplus 0.$  
$\text{DSR2: } \forall b_i (i=1...n), b, postcondition(c, precondition) \oplus 0.$  
$\text{S-DSR1: } \cup b_i, output(c, input) (i=1...n).$  
$\text{S-DSR2: } \cup b_i, postcondition(c, precondition) (i=1...n).$  |
| **Deferred Choice**      |                                                                                   | ![Diagram](attachment:Workflow_Diagram.png) | $\text{def_choice}(a,b_1,...,b_n)$ | $\text{DCR1: } \forall b_i (i=1...n), a, output(b_i, input) \oplus 0.$  
$\text{DCR2: } \forall b_i (i=1...n), a, postcondition(b_i, precondition) \oplus 0.$  
$\text{S-DCR1: } \forall b_i (i=1...n), a, output \oplus b_i, input.$  
$\text{S-DCR2: } \forall b_i (i=1...n), a, postcondition = b_i, precondition.$  |
| **Interleaved Parallel Routing** |                                                                                   | ![Diagram](attachment:Workflow_Diagram.png) | $\text{int_par}(a,b_1,...,b_n)$ | $\text{IPR1: } \forall b_i (i=1...n), a, output(b_i, input) \oplus 0.$  
$\text{IPR2: } \forall b_i (i=1...n), a, postcondition(b_i, precondition) \oplus 0.$  
$\text{S-IPR1: } a, output \oplus b_i, input (i=1...n).$  
$\text{S-IPR2: } a, postcondition \oplus b_i, precondition (i=1...n).$  |
| **Milestone**            |                                                                                   | ![Diagram](attachment:Workflow_Diagram.png) | $\text{mil}(a,b)$ | $\text{MLR1: } a, postcondition(b, precondition) \oplus 0.$  
$\text{S-MLR1: } a, postcondition(b, precondition) \oplus 0.$  |

**Ye Wang** received the B.S. degree in Software Engineering from Zhejiang University in 2007. She is a PhD student in the College of Computer Science, Zhejiang University. Her research interests include requirements modeling and analysis.

**Cheng Chang** received the B.S. degree in Software Engineering from Zhejiang University in 2007. He is a research associate in the College of Computer Science, Zhejiang University. His research interests include requirements analysis and software development methodologies.

**Xiaohu Yang** received a PhD degree in computer science from Zhejiang University in 1993. Since 1994, he has been a faculty member and professor in the College of Computer Science, Zhejiang University. He is a member of the IEEE and the IEEE Computer Society. He has a broad research interests in software engineering, requirements engineering, and software technology financial services.

**Alexander J. Kavs** received his BSc degree in occupational safety from the University of Nis, Yugoslavia in 1971. After doing operational work in the field and teaching at the School of Safety Engineering at the University of Ljubljana, Yugoslavia, he moved to the US in 1981 and since then he works in the software development. Now he is senior vice president of State Street Corporation. His research interests include workflow management systems to financial applications and document imaging, designing and programming database management systems, and computer languages.