Chapter 4  
Dynamic Workflow

Michael Adams

4.1 Introduction

Change is an accepted part of every modern workplace. To remain effective and competitive, organizations must continually adapt their business processes to manage the rapid changes demanded by the dynamic nature of the marketplace or service environment.

However, workflow management systems are generally designed to support the modeling of rigidly structured business processes, which in turn derive well-defined workflow instances. The proprietary process definition frameworks often imposed make it difficult to support (1) dynamic evolution and adaptation (i.e., modifying process definitions during execution) following unexpected or developmental change in the business processes being modeled; and (2) deviations from the prescribed process model at runtime.

The term flexibility is used to denote the degree to which a workflow system is able to support or handle expected or unexpected deviations in the execution of process instances, both from within the context of the instance or from the external environment, without negatively impacting on the essence of the process or its expected completion.

Historically, there is generally little or no flexibility provided by systems to accommodate the natural evolution of the work process or organizational goals. Manual interventions into workflow processes become increasingly frequent as staff attempt to manipulate workflow inputs and outputs to conform with changes in workplace practices. These manual intrusions necessitate reduced productivity and increased processing time. Since it is undertaken in an ad-hoc manner, manual handling incurs an added penalty: the corrective actions undertaken are not added to “organizational memory,” and so natural process evolution is not incorporated into future iterations of the process. In fact, after initial deployment, the inevitable system changes are often handled so haphazardly that they can lead to major work disruptions and increasing dissatisfaction to the point where the entire system implementation is considered a failure.
In other work environments, such as those where activities are more creatively focussed, formal representations may provide merely a contingency around which ad-hoc tasks can be formulated. Barthelmess et al. state that “In real life, both in office and scientific lab environments, the enactment of any workcase may deviate significantly from what was planned-modeled” [41]. Thus, adherence to formal representations of task sequences ignores, and may even damage, the informal work practices that also occur in any set of activities.

In summary, a large group of business processes do not easily map to the rigid modeling structures provided due to the lack of flexibility inherent in a framework that, by definition, imposes rigidity. Rather, process models are “system-centric,” meaning that work processes are forced into the paradigm supplied, rather than the paradigm reflecting the way work is actually performed. As a result, users are forced to work outside the system, and/or constantly revise the process model, in order to successfully complete their activities, thereby negating the perceived efficiency gains sought by implementing a workflow solution in the first place. Therefore, for a workflow system to be most effective, it must support dynamic change (i.e., during execution).

4.2 YAWL and Dynamic Workflow

Human work is complex and is governed by rules often to a much lesser extent than computerized processing. While some workplaces have strict operating procedures because of the work they do (e.g., air traffic control), many workplaces have few prespecified routines, but successfully complete activities by developing a set of informal tasks that can be flexibly and dynamically combined to solve a large range of problems. Generally, approaches to workflow flexibility usually rely on a high-level of runtime user interactivity, which directly impedes the basic aim of workflow systems (to bring greater efficiencies to work practices) and distracts users from their primary work procedures into process support activities. Another common theme is the complex update, modification, and migration issues required to evolve process models.

The YAWL language supports flexibility through a number of constructs at design time. Like many other languages, YAWL supports parallel branching, choice, and iteration natively, which allow for certain paths to be chosen, executed, and repeated based on conditionals and data values of the instance. In addition (and unlike most other languages), YAWL also supports advanced constructs such as multiple-atomic and multiple-composite tasks, where several instances of a task or subnet can be executed concurrently (and dynamically created), and cancelation sets, which allow for arbitrary tasks (or sets of tasks), to canceled or removed from a process instance. Chapter 2 deals with these forms of flexibility in more detail.

The YAWL environment also supports flexibility through its service-oriented architecture (cf. Chap. 7). This means that dedicated services can be built that leverage the power of the YAWL enactment engine to provide flexibility for processes in various ways.
An integral service distributed as part of the YAWL environment that provides for dynamic flexibility and exception-handling support for YAWL processes is the Worklet Service. The remainder of this chapter will discuss the unique conceptual design of the Worklet Service and how it supports dynamic workflow. The way the Worklet Service handles runtime exceptions is described in the next chapter. The service implementation (with examples) is detailed in Chap. 11. In addition, Chap. 6 describes Declare, an approach to workflow flexibility using constraints.

4.3 Worklets: Theoretical Basis

Whenever a series of actions is undertaken with a view of achieving a preconceived result, some plan or set of principles is implemented that guide and shape those actions towards that goal. To be effective, a plan must be described using constructs and language that are relevant to both the actions being performed and the desired result, and be comprehensible by its participants and stakeholders. In workflow terms, analysts seek to model some aspect of the real world by using a metaphor that bears some resemblance to the real world, but also represents an understanding of computational processes. Such metaphors are abstract constructions that form a common reference model, which assist us in representing the external world through computers.

The fundamental and widely understood computational metaphor [242] takes a set of inputs, performs a series of functional steps in a strict sequence, and, on completion, produces some output that represents the goal of the process. Thus the computational metaphor describes a single, centralized thread of control, which very much reflects its mathematical ancestry, and reveals the influence of pioneers such as von Neumann and his team, and especially Turing, whose abstract machine proposed “step-at-a-time” processing, and which in turn reflects the influence on thinking of the contemporaneous development of assembly-line manufacturing.

As the prevailing technological advances influenced the structure of early computers, so too has the computational metaphor become a significant model system for the conceptualization and interpretation of complex phenomena, from cognition to economics to ecology. Of particular interest is the way the metaphor has been applied to the definition of organizational behavior issues and the representation of organizational work processes. The computational metaphor remains applicable to well-defined problem domains where goal-directed, sequential, endpoint-driven planning is required. Such domains were the early beneficiaries of workflow management systems. Consequently, workflow systems typically provide support for standardized, repetitive activities that do not vary between execution instances.

Adherence to the metaphor by workflow systems has been an important factor in their acceptance by organizations with structured work practices. Descriptions can be found throughout the workflow literature to the “processing,” “manufacturing,” and “assembly-line” modeling metaphors that are employed by commercial workflow systems. However, while the Workflow Management Coalition claims that “even office procedures can be processed in an assembly line” [271], there are
many aspects where administrative and service processes differ from manufacturing processes.

It may be that while the computational metaphor has been a major enabler of workflow solutions, it may also have played the part of an inhibiting factor in the development of workflow systems able to effectively and dynamically support flexible work practices. Many technologically adept systems fail because they ignore human and social factors. A workflow management system that better supports flexible work environments requires a sound theoretical foundation that describes how work is conceived, carried out, and reflected upon. One such theoretical base can be found in Activity Theory.

### 4.3.1 Activity Theory – An Overview

This section gives a brief summary of Activity Theory, which forms the theoretical framework for the Worklet Service. Activity Theory is a powerful and clarifying descriptive tool rather than a strongly predictive theory that originated in the former Soviet Union in the 1930s and 1940s as part of the cultural-historical school of psychology.

Before an activity is performed in the real world, it is typically planned using a model. Generally, the better the model, the more successful the activity. However, models and plans are not rigid and accurate descriptions of the execution steps but always incomplete and tentative. Plans can be used as historical data by investigating not adherence to the plan, but the deviations from it. It is the deviations that represent a learning situation, and therefore only the deviations need to be recorded. The experience of using a plan to guide an activity is gained during the instantiation of the activity. In order for plans to become resources for future instantiations of an activity, it is important that the planning tool allows for the ongoing creation and dynamic modification of a plan based on experience gained while operating the plan.

One of the traditional limitations of workflow implementations in less than strictly defined work processes is that it is very difficult if not impossible to incorporate every deviation into the workflow template and therefore future instantiations of the plan. Some deviations may apply to every future instance. Some may only apply once in a while, but these cases should not be left out of the plan. In the normal course of events, these “deviations” are performed externally to the system. But incorporating dynamic change is a fundamental feature of the Worklet Service’s design.

To summarize, Activity Theory states that human activity has four basic characteristics:

1. Every activity is directed towards a material or ideal object satisfying a need, which forms the overall motive of the activity. For example, an order fulfillment process is directed toward completing the order and delivering the goods to the customer.
2. Every activity is mediated by artifacts, either external (order forms, delivery invoices) or internal (cognitive – compiling freight routes, knowledge, and experience of order staff)

3. Each individual activity is almost always part of collective activities, structured according to the work practice in which they take place. For example, a order cannot be fulfilled without reference to a diversity of other information, such as carrier availability, financial arrangements, stockists, and so on. Thus collective activities are organized according to a division of labor

4. Finally, human activity can be described as a hierarchy with three levels: activities realized through chains of actions, which may be performed through operations:

   - An activity consists of one or more actions, and describes the overall objective or goal
   - An action equates to a single task carried out to achieve some preconceived result. For example, an order fulfillment activity consists of a number of actions: (1) creating an order request, (2) filling the order, (3) appointing a carrier, (4) receiving payment, and (5) final delivery. Each action is achieved through operations determined by the actual conditions in the context of the activity
   - Operations describe the actual performance of the action, and are dependent on the context or conditions that exist for each action. Exactly how the carrier appointment is performed, for example, depends clearly on the concrete conditions, for example, the distance, the type of carrier requested, the size and weight of the freight, availability of appropriate carriers, and so on

4.3.2 Principles Derived from Activity Theory

Ten fundamental principles, representing an interpretation of the central themes of Activity Theory applicable to an understanding of organizational work practices, have been derived and are summarized below.

- **Principle 1 – Activities are hierarchical**: An activity consists of one or more actions. Each action consists of one or more operations.
- **Principle 2 – Activities are communal**: An activity almost always involves a community of participants working towards a common objective.
- **Principle 3 – Activities are contextual**: Contextual conditions and circumstances deeply affect the way the objective is achieved in any activity.
- **Principle 4 – Activities are dynamic**: Activities are never static but evolve asynchronously, and historical analysis is often needed to understand the current context of the activity.
- **Principle 5 – Activities are mediated**: An activity is mediated by tools, rules, and divisions of labor.
Principle 6 – Actions are chosen contextually: A repertoire of actions and operations is created, maintained, and made available to any activity, which may be performed by making contextual choices from the repertoire.

Principle 7 – Actions are understood contextually: The immediate goal of an action may not be identical to the objective of the activity of which the action is a component. It is enough to have an understanding of the overall objective of the activity to motivate successful execution of an action.

Principle 8 – Plans guide work: A plan is not a blueprint or prescription of work to be performed, but merely a guide, which is modified depending on context during the execution of the work.

Principle 9 – Exceptions have value: Exceptions are merely deviations from a preconceived plan. Deviations will occur with almost every execution of the plan, and give rise to a learning experience, which can then be incorporated into future executions.

Principle 10 – Granularity based on perspective: A particular piece of work might be an activity or an action depending on the perspective of the viewer.

Activity Theory offers a number of interesting insights into workflow research domains, particularly the related issues of workflow adaptability, flexibility, evolution, and exception handling. The derived principles above have formed the theoretical foundations for the implementation and deployment of the Worklet Service. Activity Theory was chosen as the theoretical framework because it provides, as demonstrated in this section, a tight fit between actual work practices and the requirements of PAIS designed to support them. This section does not claim Activity Theory to be the only applicable theoretical framework, but merely one from which sound principles of work practice for adaptive business processes could be derived.

4.4 Conceptualization of Worklets

The consideration of the derived principles of Activity Theory formed the conceptual foundations of the Worklet Service, a discrete service that transforms otherwise static workflow processes into fully flexible and dynamically extensible process instances that are also supported by dynamic exception handling capabilities (cf. Chap.5). This chapter represents a conceptual view of the Worklet Service; the implementation and use of the service is described in detail in Chap. 11.

Fundamentally, a workflow management system that is based on the principles derived from Activity Theory would satisfy the following criteria:

- A flexible modeling framework: a process model is to be regarded as a guide to an activity’s objective, rather than a prescription for it
- A repertoire of actions: extensible at any time, the repertoire would be made available for each task during each execution instance of a process model
Dynamic, contextual choice: to be made dynamically from the repertoire at runtime by considering the specific context of the executing instance.

Dynamic process evolution: allow the repertoire to be dynamically extended at runtime, thus providing support for unexpected process deviations, not only for the current instance, but also for other current and future instantiations of the process model, leading to natural process evolution.

Thus, to accommodate flexibility, such a system would provide each task of a process instance with the ability to be linked to an extensible repertoire of actions, one of which to be contextually and dynamically chosen at runtime to carry out the task. To accommodate exception handling, such a system would provide an extensible repertoire of exception-handling processes to each process instance, members of which to be contextually and dynamically chosen to handle exceptions as they occur.

To support dynamic workflow, the Worklet Service presents the repertoire-member selection actions as worklets. In effect, a worklet is a small, self-contained, complete workflow process, which handles one specific task (action) in a larger, composite process (activity). A top-level or parent process model is developed that describes the workflow at a macro level. From that manager process, worklets may be contextually selected and invoked from the repertoire of each enabled task, using an associated extensible set of selection rules, when the task instance becomes enabled during execution. New worklets for handling a task may be added to the repertoire at any time (even during process execution) as different approaches to completing a task are developed and derived from the context of each process instance. Importantly, the new worklet becomes an implicit part of the process model for all current and future instantiations, avoiding issues of migration and version control. In this way, the process model undergoes a dynamic natural evolution.

The worklet approach provides support for the modeling, analysis, and enactment of business processes, and directly provides for dynamic exception handling, ad-hoc change, and process evolution, without having to resort to off-system intervention and/or system downtime.

### 4.5 Context, Rules, and Worklet Selection

For any situation, there are multiple situational and personal factors that combine to influence a choice of action. The set of factors that are deemed to be relevant to the current situation we call its context.

The consideration of context plays a crucial role in many diverse domains, including philosophy, pragmatics, semantics, cognitive psychology, and artificial intelligence. Capturing situated context involves quantifying and recording the

---

1 In Activity Theory terms, a worklet may represent one action within an activity, or may represent an entire activity.
relevant influencing factors and relationships between the inner state and the external environment.

A taxonomy of contextual data that may be recorded and applied to a workflow instance may be categorized as follows (examples are drawn from the Order Fulfillment process):

- **Generic (case independent):** Data attributes that can be considered likely to occur within any process (of course, the data values change from case to case). Such data would include descriptors such as when created, created by, times invoked, last invoked, current status; and role or agent descriptors such as experience, skills, rank, history with this process and/or task, and so on. Process execution states and process log data also belong to this category.

- **Case dependent with a priori knowledge:** The set of data that are known to be pertinent to a particular case when it is instantiated. Generally, this data set reflects the data variables of a particular process instance. Examples are customer name, address, and delivery location; freight costs, size, and weight; ordered item names, descriptions, costs, etc.; and deadlines both approaching and expired.

- **Case dependent with no a priori knowledge:** The set of data that only becomes known when the case is active and deviations from the known process occur. Examples in this category may include complications that arise for incorrect payments; unavailable stock, routes, and/or couriers; natural disasters preventing delivery; and so on.

Methods for capturing contextual data typically focus on collecting a complete set of knowledge from an “expert” and representing it in a computationally suitable way. Such approaches depend heavily on the expert’s ability to interpret their own expertise and express it in nonabstract forms. However, experts often have difficulty providing information on how they reach a specific judgment, and will offer a justification instead of an explanation. Furthermore, the justification given varies with the context in which the judgement was made.

Theories of context generally fall into two distinct groups: divide-and-conquer, a top-down approach that views context as a way of partitioning a global model into simpler pieces (e.g., the “expert” approach described above), and compose-and-conquer, a bottom-up approach that holds that there is no tangible global model to begin with, but only local perspectives, and so views context in terms of locality in a (possible or potential) network of relations with other local perspectives.

A top-down, complete representation of knowledge within a given domain is considered by many researchers to be impossible to achieve in practice, and is perhaps not even desirable. In terms of using context as a factor in computational decision making, it is considered more judicious to capture only that subset of the complete contextual state of a particular domain relevant to making a correct and informed decision.

One bottom-up approach to the capture of contextual data that offers an alternative method to global knowledge construction is Ripple Down Rules (RDR), which comprise a hierarchical set of rules with associated exceptions.
The fundamental feature of RDR is that it avoids the difficulties inherent in attempting to precompile a systematic understanding, organization, and assembly of all knowledge in a particular domain. The RDR method is well established and fully formalized and has been implemented as the basis for a variety of commercial applications, including systems for reporting DNA test results, environmental testing, intelligent document retrieval, fraud detection based on patterns of behavior, personal information management, and data mining of large and complex data sets. The Worklet Service uses RDR to define rules that allow the correct worklet to be chosen from a repertoire of available worklets for a given task in a process instance, using the particular context of the instance.

An RDR Knowledge Base is a collection of simple rules of the form “if condition then conclusion” (together with other associated descriptors), conceptually arranged in a binary tree structure (e.g., Fig. 4.1). Each rule node may have a false (“or”) branch and/or a true (“exception”) branch to another rule node, except for the root node, which contains a default rule and can have a true branch only. If a rule is satisfied, the true branch is taken and the associated rule is evaluated; if it is not satisfied, the condition of the root node is evaluated.

![Fig. 4.1 Conceptual structure of a Ripple-Down rule (assess claim example)](image-url)
satisfied, the false branch is taken and its rule evaluated. When a terminal node is reached, if its rule is satisfied, then its conclusion is taken; if its rule is not satisfied, then the conclusion of the last rule satisfied on the path to that node is taken. For terminal nodes on a true branch, if its rule is not satisfied then the last rule satisfied will always be that of its parent (since it must have evaluated to true for the terminal node to be evaluated).

This tree traversal gives RDR implied locality – a rule on an exception branch is tested for applicability only if its parent (next-general) rule is also applicable. This feature provides the fundamental benefit of RDR: general rules are defined first, and refinements to those rules added later as the need arises or as knowledge about the domain grows. Thus, there is always a working rule set that extends over time.

For example, the rule tree in Fig. 4.1 represents the following illustrative set of (somewhat artificial) business rules:

1. **Node 0**: By default, assess a claim
2. **Node 1**: An exception to the node 0 rule is that for those cases where the claim amount is greater than $10,000, the claim must be referred to a manager (since the condition of node 0 is always satisfied, the condition of node 1 will always be evaluated)
3. **Node 3**: An exception to the node 1 rule is that cases where the claim amount is greater than $10,000 may be assessed by an assessor of at least Senior role (satisfied node 1)
4. **Node 6**: An exception to the node 3 rule is that those cases where the claim amount is greater than $50,000 must always be referred to a manager (regardless of the rank of available assessors (satisfied node 3)
5. **Node 2**: If the claim amount is less than $10,000 (and so the condition of node 1 was not satisfied), then if the claimant has already made a claim in the current year it must be investigated
6. **Node 4**: An exception to the node 2 rule is that if the claim amount is also less than $3,000, simply assess the claim (i.e., satisfied node 2, but the amount is too trivial to warrant an investigation)
7. **Node 5**: An exception to the node 4 rule is that for those cases where the claimant status is set to “suspect,” the claim should always be investigated (satisfied node 4)
8. **Node 7**: If the claim amount is less than or equal to $10,000 (unsatisfied node 1) and there has been a claim from this claimant in the current year (satisfied node 2) and the claim amount is also greater than or equal to $3,000 (unsatisfied node 4) and the claim is for storm damage, then the claim should be escalated for express payment

If the conclusion returned is found to be unsuitable for a particular case instance – that is, while the conclusion was correct based on the current rule set, the circumstances of the case instance make the conclusion an inappropriate choice – a new rule is formulated that defines the contextual circumstances of the instance and is added as a new leaf node using the following algorithm:
• If the conclusion returned was that of a satisfied terminal rule, then the new rule is added as a local exception to the exception “chain” via a new true branch from the terminal node.

• If the conclusion returned was that of a nonterminal, ancestor node (i.e., the condition of the terminal rule was not satisfied), then the new rule is added via a new false branch from the unsatisfied terminal node.

In essence, each added exception rule is a refinement of its parent rule. This method of defining new rules allows the construction and maintenance of the rule set by “subdomain” experts (i.e., those who understand and carry out the work they are responsible for) without regard to any engineering or programming assistance or skill.

Importantly, each rule node also incorporates a set of case descriptors, called the “cornerstone case,” which describe the actual case context that was the catalyst for the creation of the rule. When a new rule is added to the rule set, its conditional predicate is determined by comparing the descriptors of the current case to those of the cornerstone case and identifying a subset of differences. Not all differences will be relevant – *it is only necessary to determine the factor or factors that make it necessary to handle the current case in a different fashion to the cornerstone case to define a new rule*. The identified differences are expressed as attribute-value pairs, using the usual conditional operators. The current case descriptors become the cornerstone case for the newly formulated rule; its condition is formed by the identified attribute-value pairs and represents the context of the case instance that caused the addition of the rule.

Rather than impose the need for a closed knowledge base that must be completely constructed a priori, this method allows for the identification of that part of the universe of discourse that differentiates a particular case *as the need arises*. Indeed, the only context of interest is that needed for differentiation, so that rule sets evolve dynamically, from general to specific, through experience gained as they are applied.

Ripple-Down Rules are well suited to the worklet selection processes, since they:

• Provide a method for capturing relevant, localized contextual data
• Provide a hierarchical structuring of contextual rules
• Do not require the top-down construction of a global knowledge base of the particular domain prior to implementation
• Explicitly provide for the definition of exceptions at a local level
• Do not require expert knowledge engineers for its maintenance
• Allow a rule set to evolve and grow, thus providing support for a dynamic learning system

Each worklet is a representation of a particular situated action that relies on the relevant context of each case instance, derived from case data and other (archival) sources, to determine whether it is invoked to fulfill a task in preference to another worklet within the repertoire. When a new rule is added, a worker describes the contextual conditions as a natural part of the work they perform\(^2\). This level

\(^2\) In practice, the worker’s contextual description would be passed to an administrator, who would add the new rule.
of human involvement – at the “coal-face,” as it occurs – greatly simplifies the capturing of contextual data. Thus RDR allows the construction of an evolving, highly tailored local knowledge base about a business process.

4.6 The Selection Process

The worklet approach allows for two related but discrete areas of dynamic and flexible workflow to be addressed: dynamic selection of tasks and exception handling with corrective and compensatory action. A conceptual synopsis of the selection process is dealt with in this section; exception handled in Chap. 5.

When a YAWL specification is created in the Editor (cf. Chap. 8), one or more of its tasks may each be associated with a corresponding repertoire of worklets from which one will be selected as a substitute for the task at runtime. Each task associated with the Worklet Service has its own particular repertoire, and its members may be found in a number of other repertoires. Along with the specification, a corresponding RDR rule set is created, which defines the conditions to be evaluated against the contextual data of the case instance. That is, each task may correspond to a particular “tree” of RDR nodes within which are referenced a repertoire of worklets, one of which may be selected and assigned as a substitute for the task dynamically. Not all tasks need be linked to a repertoire – only those for which worklet substitution at runtime is desired.

Each task that is associated with a worklet repertoire is said to be “worklet-enabled.” This means that a process may contain both worklet-enabled tasks and non-worklet-enabled (or ordinary) tasks. Any process instance that contains a worklet-enabled task will become the parent process instance for any worklets invoked from it.

 Importantly, a worklet-enabled task remains a valid (ordinary) task definition, rather than being considered as merely a vacant “placeholder” for some other activity (i.e., a worklet). The distinction is crucial because, if an appropriate worklet for a worklet-enabled task cannot be found at runtime (based on the context of the case and the rule set associated with the task), the task is allowed to run as an “ordinary” task, as it normally would in a process instance. So, instead of the parent process being conceived as a template schema or as a container for a set of placeholders, it is to be considered as a complete process containing one or more worklet-enabled tasks, each of which may be contextually and dynamically substituted at runtime.

It is possible to build for a task an initial RDR rule tree containing many nodes, each containing a reference to a worklet that will be used if the conditional expression for that node and its parent nodes are satisfied; alternately, an initial rule tree can be created that contains a root node only, so that the worklet-enabled task runs as an ordinary task until such time that the rule tree is extended to capture new contextual scenarios (which may not have been known when the process was first defined). Thus, the worklet approach supports the full spectrum of business processes, from the highly structured to the highly unstructured.
Consider the simple insurance claim example in Fig. 4.2. Suppose that, after a while, a new business rule is formulated, which states that when a claim comes to be assessed, if the claim amount is more than $10,000 then it must be referred to a manager. In conventional workflow systems, this would require a redefinition of the model. Using the worklet approach, it simply requires a new worklet to be added to the repertoire for the Assess Claim task and a new rule added as a refinement to the appropriate RDR by the administrator. That is, the new business rule is added as a localized refinement of a more general rule (see Fig. 4.1).

The modified RDR tree can be used to extrapolate a view or schematic representation of the model, with the modified rule for the Assess Claim represented as XOR choice (Fig. 4.3). That is, a tool can be used to translate the RDR set back into a view of a set of tasks and conditional branches within a standard monolithic workflow schema; of course, a translated rule set of a more than trivial size would demonstrate the complexities of describing the entire set of possible branches monolithically. This approach enables the model to be displayed as the derived view in Fig. 4.3, or as the original representation with separate associated worklets, thereby offering layers of granularity depending on factors such as the perspective of the particular stakeholder and the frequency of the occurrence of a condition-set being satisfied. From this it can be seen that an RDR tree may be represented in the modeling notation as a composite set of XOR splits and joins. The advantage of using RDRs is that the correct choice is made dynamically and the available choices grow and refine over time, negating the need to explicitly model the choices and repeatedly update the model (with each iteration increasingly camouflaging the original business logic).

It may also be the case that changes in the way activities are performed are identified, not by an administrator or manager via new business rules, but by a worker who has been allocated a task. Following the example above, after Log Claim completes,
Assess Claim is selected and assigned to a worker’s worklist for action. The worker may decide that the generic Assess Claim task is not appropriate for this particular case, because this claimant resides in an identified storm-damaged location. Thus, the worker rejects the Assess Claim worklet via a button on their inbox. On doing so, the system refers the rejection to an administrator who is presented with the set of case data for Assess Claim (i.e., its cornerstone case), and the set of current case data.

The administrator then compares the two sets of case data to establish which relevant aspects of the current case differ from Assess Claim’s cornerstone. Note that while many of the values of the two cases may differ, only those that relate directly to the need to handle this case differently are selected (e.g., the location of the claimant and the date the damage occurred). After identifying the differences, the administrator is presented with a list of possible worklet choices, if available, that may suit this particular context. The administrator may choose an appropriate worklet to invoke in this instance, or, if none suit, define a new worklet for the current instance. In either case, the identified differences form the conditional part of a new rule, which is added to the RDR tree for this task using the rule addition algorithm described earlier.

The principles derived from Activity Theory state that all work activities are mediated by rules, tools, and division of labor. Translating that to an organizational work environment, rules refer to business rules, policies and practices; tools to resources and their limitations (physical, financial, staff training, experience and capabilities, and so on); and division of labor to organizational hierarchies, structures, roles, lines of supervision, etc. Of course, these constraints apply to the creation of a new worklet, just as they would in any workflow management system. This means that the authority to create new worklets and add rules to rule sets would rest with the appropriate people in an organization, and the authority to reject inappropriate worklets would reside within the duties of a worker charged with performing the task – the “subdomain expert.” Of course, spurious rejection of worklets would be managed in the same way as any other instances of misconduct in the workplace.

In all future instantiations of a specification, the new worklet defined following a rejection would be chosen for that task if the same contextual conditions occur in a new instance’s case data. Over time, the RDR rule tree for the task grows towards specificity as refinements are added (cf. Fig. 4.1).

4.7 Service Interface

To enable the Worklet Service to serve the YAWL enactment engine, a number of events and methods must be provided by an interface between them. Being a web-service (cf. Chap. 7), the Worklet Service has been designed to enable remote deployment (i.e., deployed on a web server in a location remote to the YAWL Engine) and to allow a single instance of the service to concurrently manage the
flexibility and exception handling management needs for a number of disparate enactment engines that conform to the interface.

The interface requires a number of events and methods, some originating from the service side and others from the engine side. Some require a response, while others do not require any acknowledgment (such as event notifications that do not necessarily require action).

This section describes the selection interface requirements. The event notifications that must be provided to the service by the engine are the following:

- **A work item is enabled event**, where the engine notifies the service that a work item is ready to be executed. The Worklet Service will use this event as a trigger to query the rule set to discover if the enabled work item has a worklet repertoire and, if so, if one is appropriate to act as a substitute for the work item. If an appropriate worklet is found, this event becomes the catalyst for a service selection procedure. If an appropriate worklet is not found, the event is simply ignored, allowing the work item to be executed in the default manner for the engine. Since it is only necessary for an engine to notify the service if there may be available worklets for a work item, the service-aware YAWL Engine would allow tasks to be flagged as “service-enabled” and thus would operate more efficiently than an engine that sent events for every enabled work item, although this is not a necessary requirement of the enactment engine in general.

- **A work item is canceled event**, which is necessary to accommodate the situation where the service has substituted a worklet for a work item and the work item is canceled (e.g., if it is a member of a cancellation region of the parent process or if the parent process is canceled). In such cases, the service will need to take appropriate action. Again, a service-aware engine would only generate this event if the service has already substituted the work item with a worklet, but it is not a necessary requirement of an enactment engine.

- **A process instance has completed event**, which is required by the service so that it is aware when a worklet instance completes to enable the finalization of the substitution procedure and allow the parent process to continue to the next task.

Each of the events above do not require acknowledgments from the service back to the originating engine – they are simply notifications from the engine that may or may not be acted on.

In addition to the three events, a small number of interface methods are required to be made available to the interface to enable the service to communicate with the engine and take the appropriate action. The required methods are generic in their nature and so would typically be available in most workflow enactment engines. They are the following:

- **Connect to engine**: A connection would typically be required through the interface to enable messaging to pass between the engine and the Worklet Service.
- **Load specification**: The service requires the ability to load a worklet specification into the enactment engine. Since a worklet may be represented as a
“normal” specification of the host enactment engine, this method would already be available within the engine.

- **Launch case:** The service must have the ability to launch an instance of a loaded worklet specification.
- **Cancel case:** The service requires the means to cancel a launched worklet instance (e.g., if the work item it has substituted for is canceled, then the worklet instance would typically need to also be canceled).
- **Check-out work item:** When the service has been notified of an enabled work item, and there is a worklet to act as a substitute for it, the service needs a way to take control of the work item’s execution. Thus, by “checking out” the work item from the engine, the engine would pass responsibility for the execution of the work item to the service, and wait for the service to notify it that the execution has completed (i.e., when the worklet case completes).
- **Check-in work item:** When a worklet instance has completed, the service will notify the engine that execution of the original work item it acted as a substitute for has completed. The data gathered by the worklet would be mapped to the work item before it was checked in.

Figure 4.4 summarizes the interface required by the service’s selection process.

![Required selection interface](image-url)

---

**Fig. 4.4** Required selection interface
4.8 Secondary Data Sources

When making a contextual choice of an appropriate worklet, it may be desirable or even necessary to seek data outside the parameters of the task/or case. For example, the current state of the process, or the states of individual tasks, may have some bearing on the choices made; available resources may be an important factor and so on. Thus, choosing the most appropriate worklet for a situation will be achieved by defining rules that use a combination of currently available data attribute values, both case-dependent and independent.

One method of accessing the current set of states for an instantiated process, resource data, and archival histories of previous instantiations of a specification may be deduced by mining the process log file. A series of predicates can be constructed to enable the extraction of the current state set and any relations between active worklets and tasks, as well as archival trends and relations. These predicates may then be used to augment the conditionals in RDR nodes to enable selection of the most appropriate worklet.

The kinds of information that may be extracted from the process log file using these predicates include the current status of a worklet, whether a worklet is a parent or child of another worklet, when a certain state was entered or exited for a particular worklet or task, the resource that triggered a state change, and so on. Chapter 11 provides an explanation of how such predicates may be constructed and used in practice.

Figure 4.5 shows an ORM diagram for the log kept by the Worklet Service. The entity “Parent Case” in the worklet log corresponds to the case identifier of the parent process that a worklet was launched for, which can be drawn from the engine’s process logs. Hence, the entity would map to a record of the parent case instance in
the engine process logs. The worklet log entities “Task” and “Case” would also map to corresponding records in the engine process logs. Thus, by mapping those entity values, a complete view of the process, inclusive of any actions taken on behalf of it by the Worklet Service, can be constructed.

It should be noted that the event types for the engine log differ from those of the worklet log. Since worklets are launched as separate cases in the workflow engine, the engine’s log records the progress of the process instance, so that it is not necessary to record those types of events in the Worklet Service logs – indeed, to do so would be a duplication. Thus, the worklet log concentrates on the event details that are not recorded on the engine side, such as CheckOutWorkItem, WorkletLaunched, and so on.

4.9 Conclusion

The worklet approach to dynamic workflow presents several key benefits, including the following:

- A process modeler can describe the standard activities and actions for a workflow process and the worklets for particular tasks using the same modeling methodology
- It allows reuse of existing process and worklet components. Removing the differentiation between dynamically inserted worklets and the “normal” workflow aids in the development of fault tolerant workflows out of preexisting building blocks
- Its modularity simplifies the logic and verification of the standard model, since individual worklets are less complex to build and therefore to verify than monolithic models
- It provides for workflow views of differing granularity, which offers ease of comprehensibility for all stakeholders
- It allows for gradual and ongoing evolution of the model, so that global modification to the model each time a business practice changes or an exception occurs is unnecessary
- In the occurrence of an unexpected event, an administrator needs simply to choose an existing worklet or build a new one for the particular context, which can be automatically added to the repertoire for future use as necessary, thus avoiding complexities including downtime, model restructuring, versioning problems, and so on.

Most importantly, the worklet approach is built on the solid theoretical foundations of Activity Theory, and so fully supports the set of derived principles of organizational work practices, and the criteria for a workflow support system based on those principles.
Exercises

Exercise 1. How does a worklet process model differ from a normal YAWL process model?

Exercise 2. Name three principal insights into human work activity offered by Activity Theory that have application in the development of support for dynamic flexibility in Process-aware Information Systems.

Exercise 3. Construct a Ripple-Down Rule tree (on paper) for the following items. In each case, the condition will be the type of item, and the conclusion will be “can fly” if the item can fly, or “can’t fly” if it cannot, generally speaking. The rule tree is to be constructed in the sequence in which the items are listed, which proceed from the more general to the more specific.

(a) A bird
(b) A cat
(c) A baby bird
(d) An airplane
(e) A penguin
(f) A penguin inside an airplane

Exercise 4. Construct a Ripple-Down Rule tree (on paper) for the following rules. The rule tree is to be constructed in the sequence in which the items are listed.

(a) If it is warm, then play tennis
(b) If it is raining, then stay indoors
(c) If it is dry, then water the garden
(d) If it is snowing, then go skiing
(e) If it is fine, then go on a picnic
(f) If it is cool, then go to a movie
(g) If it is snowing heavily, then stay indoors
(h) If it is fine and cool, then go shopping
(i) If it is warm and raining, then stay indoors

Exercise 5. Construct a Ripple-Down Rule tree (on paper) for the following (fictional) requirements. “A patient may present to the Casualty Department with one of five general conditions: fever, wound, rash, abdominal/chest pain, or fracture. Each of those conditions should be referred to the corresponding department for treatment. If, however, a patient with abdominal/chest pain has a pulse over 150 bpm, they should be treated for a heart condition. If a patient has a fever and a rash, they should be quarantined in the infectious diseases department. If a patient has a wound and a fracture, treatment of the wound takes precedence. If a patient with abdominal/chest pain is pregnant, they are to be referred to the maternity ward. If a patient has a wound with high blood loss, they should be immediately referred to the ER. If a patient’s fever is mild, they should be given a pill and sent home.”
Chapter Notes

Worklets

The worklet approach arose from an investigation into why workflow systems had difficulties supporting flexible processes. A strong disconnect was found between the modeling and execution frameworks supplied, and the way human work was actually performed. Simply put, existing systems had grown from programming or linear, sequential execution bases, and most work activities are far from linear. These findings were first reported in [24], which also detailed a set of criteria against which the systems could be measured for their ability to support flexible processes. The idea of worklets was first published in [26], and the worklet approach to dynamic flexibility was further detailed in [27]. A full exploration of the worklet approach, including a complete formalization and exemplary studies, can be found in [23].

For the interested reader, a discussion of the use of worklets in very creative working environments may be found in [237].

Ripple-Down Rules

Ripple-Down Rules were first devised by Compton and Jansen [62]. While on the surface it may seem that the quality of RDR sets would be dependent on the insertion sequence of new rules, and may be open to the introduction of redundant and/or repeated rules, this has been shown to be not the case. In terms of the correct selection of the appropriate rule based on case context, it is always the case that the correct rule is chosen, regardless of the insertion sequence of rules into the tree. In terms of the potential for redundancy and repetition of rules throughout the tree, studies have shown that the issue is far less serious than first perceived [63, 156] and that the size of an RDR set which includes a normal distribution of redundant rules compares favorably in size with other various inductively built Knowledge Based Systems.

In terms of the number of computational steps required to reach the finally chosen rule, it has been empirically shown that Ripple-Down Rules are able to describe complex knowledge systems using less rules than conventional “flat” rule lists [62, 99, 234]. So, when comparing insertion sequences between RDR sets and traditional decision lists, RDR sets will have the higher quality in this regard. In terms of the potential for degradation of computational time taken to traverse a rule set due to the growth of rule sets over time, a number of algorithms exist for the reordering and optimization of rule trees (cf. [99, 211, 234]). Thus, trees may occasionally be optimized and thus the highest quality can be maintained over time.

Activity Theory

For more information on Activity Theory, the interested reader is directed to [24, 39, 139].
Other Approaches

Most workflow systems use their own unique conceptual framework, which is usually based on programming constructs rather than founded on theories of work practice. Consequently, since the mid-nineties, much research has been carried out on issues related to dynamic flexibility and exception handling in workflow management systems. Such research was initiated because, generally, commercial workflow management systems required the model to be fully defined before it could be instantiated, and that any changes must be incorporated by modifying the model statically. These typically flat, monolithic, single-schema architectures make it difficult to fully capture flexible business processes [46, 117].

While there have been many proposed and/or implemented approaches to flexibility, this section discusses a sample of the more interesting ones.

An optional component of the Tibco iPProcess Suite is the Process Orchestrator [102], which allows for the dynamic allocation of subprocesses at runtime. It requires a construct called a “dynamic event” to be explicitly modeled that will execute a number of subprocesses listed in an “array” when execution reaches that event. Which subprocesses execute depend on predefined data conditionals matching the current case. The listed subprocesses are statically defined as are the conditionals. There is no scope for dynamically refining conditionals, nor adding subprocesses at runtime.

COSA (version 5.4) [66] provides for the definition of external “triggers” or events that may be used to start a subprocess. All events and subprocesses must be defined at design time, although models can be modified at runtime (but only for future instantiations). COSA also allows manual ad-hoc runtime adaptations such as reordering, skipping, repeating, postponing, or terminating steps. SAP Workflow (version 6.20) [227] supports conditional branching, where a list of conditions (each linked to a process branch) is parsed and the first evaluating to true is taken; all branches are predefined. FLOWER (version 2.1) [45, 191] is described as a “case-handling” system; the process model (or “plan”) describes only the preferred way of doing things and a variety of mechanisms are offered to allow users to deviate in a controlled manner [22].

There have been a number of academic prototypes developed in the last decade (although activity was greater during the first half); very few have had any impact on the offerings of commercial systems [174]. Several of the more widely acknowledged are discussed here.

The eFlow system [56] supports flexibility in e-Services by defining compensation rules for regions, although they are static and cannot be defined separately to the standard model. The system allows changes to be made to process models, but such changes introduce the common difficulties of migration, verification, consistency and state modifications.

ADEPT [118, 206] supports modification of a process during execution (i.e., add, delete, and change the sequence of tasks) both at the model (dynamic evolution) and at the instance levels (ad-hoc changes). Such changes are made to a traditional monolithic model and must be achieved via manual intervention, abstracted to a high
level interaction. The system also supports forward and backward “jumps” through a process instance, but only by authorized staff who instigate the skips manually [204].

The AdaptFlow prototype [111] provides a hybrid approach to flexibility. It supports the dynamic adaptation of process instances, although each adaptation must be confirmed manually by an authorized user before it is applied (alternate manual handling to override the dynamic adaptation offered is also supported). Also, the rule classifications and available exception handling actions are limited to medical treatment scenarios. The prototype has been designed as an overlay to the ADEPT system, providing dynamic extensions.

The ADOME system [57] provides templates that can be used to build a workflow model, and provides some support for (manual) dynamic change; it uses a centralized control and coordination execution model to initiate problem solving agents to carry out assigned tasks. A catalog of “skeleton” patterns that can be instantiated or specialized at design time is supported by the WERDE system [54]. Again, there is no scope for specialization changes to be made at runtime.

AgentWork [176] provides the ability to modify process instances by dropping and adding individual tasks based on events and ECA rules. However, the rules do not offer the flexibility or extensibility of the YAWL approach, and changes are limited to individual tasks, rather than the task-process-specification hierarchy. Also, the possibility exists for conflicting rules to generate incompatible actions, which requires manual intervention and resolution.

The ActivityFlow specification language described in [151] divides workflows into different types at design time (including ad-hoc, administrative, or production), and provides an open architecture that supports interaction and collaboration of different workflow systems. The system, like ADEPT, advocates the use of a dedicated (human) workflow coordinator/administrator to monitor workflows with an eye on deadlines, handle exceptions, prioritize, stop, resume and abort processes, dynamically restructure running processes, or change a specification.

An approach that uses a “society of intelligent agents” that work together to execute flexible processes is found in [257], and another that uses BPBots (Business Process Robots) to perform the roles of service requesters, providers, and brokers in the formation of a hierarchical community for the execution of a process instance is introduced in [279]. A further approach using incompletely specified process definitions is found in the SwinDeW (Swinburne Decentralized Workflow) project [278]. SwinDew is a peer-to-peer based decentralized model, where a process definition is split into a set of task partitions and distributed to peers, and on-the-fly process elaboration is performed at runtime. Thus, a multi-tiered process modeling and execution framework is provided.

CBRFlow [258] uses a case-based reasoning approach to support adaptation of predefined workflow models to changing circumstances by allowing (manual) annotation of business rules during runtime via incremental evaluation by the user. Users must be actively involved in the inference process during each case. An approach, which integrates CBRFlow into the ADEPT framework, is described in [213]. In doing so, semantic information about the reasons for change, and traceability data,
are presented to the ADEPT user/administrator to support decision making processes. The information can also be used to facilitate reuse of ad-hoc changes from similar scenarios. When deviations from a process schema are required, the case-based reasoning component assists the user to find similar previous cases through a series of questions and answers, one of which may then be applied to the current instance [213]. While the process is quite user-intensive, the approach does provide a good example of the combination of contextual information with flexibility techniques.
Modern Business Process Automation
YAWL and its Support Environment
 ter Hofstede, A.H.M.; van der Aalst, W.; Adams, M.; Russell, N. (Eds.)
2010, XVIII, 676 p., Hardcover
ISBN: 978-3-642-03120-5