Worklets: A Service-Oriented Implementation of Dynamic Flexibility in Workflows

Michael Adams[1], Arthur H. M. ter Hofstede[1], David Edmond[1], and Wil M. P. van der Aalst[1,2]

[1] Business Process Management Group
Queensland University of Technology, Brisbane, Australia
{m3.adams,a.terhofstede,d.edmond}@qut.edu.au
[2] Department of Technology Management
Eindhoven University of Technology, Eindhoven, The Netherlands
w.m.p.v.d.aalst@tm.tue.nl

Abstract. This paper presents the realisation, using a Service Oriented Architecture, of an approach for dynamic flexibility and evolution in workflows through the support of flexible work practices, based not on proprietary frameworks, but on accepted ideas of how people actually work. A set of principles have been derived from a sound theoretical base and applied to the development of *worklets*, an extensible repertoire of self-contained sub-processes aligned to each task, from which a dynamic runtime selection is made depending on the context of the particular work instance.

1 Introduction

Workflow management systems are used to configure and control structured business processes from which well-defined workflow models and instances can be derived [1, 2]. However, the proprietary process definition frameworks imposed make it difficult to support (i) dynamic evolution (i.e. modifying process definitions during execution) following unexpected or developmental change in the business processes being modelled [3]; and (ii) deviations from the prescribed process model at runtime [4–6].

Without support for dynamic evolution, the occurrence of a process deviation requires either suspension of execution while the deviation is handled manually, or an entire process abort. However, since most processes are long and complex, neither manual intervention nor process termination are satisfactory solutions [7]. Manual handling incurs an added penalty: the corrective actions undertaken are not added to 'organisational memory' [8, 9], and so natural process evolution is not incorporated into future iterations of the process. Other evolution issues include problems of migration, synchronisation and version control [4, 10].

These limitations mean a large subset of business processes do not easily map to the rigid modelling structures provided [11], due to the lack of flexibility inherent in a framework that, by definition, imposes rigidity. Process models are 'system-centric', or *straight-jacketed* [12] into the supplied framework, rather

than truly reflecting the way work is actually performed [13]. As a result, users are forced to work outside of the system, and/or constantly revise the static process model, in order to successfully support their activities, thereby negating the efficiency gains sought by implementing a workflow solution in the first place.

Since the mid-nineties many researchers have worked on problems related to workflow change (cf. Section 7). This paper is based on and extends the approach proposed in [14]. It introduces a realisation of *'worklets'*, an extensible repertoire of self-contained sub-processes and associated selection rules, grounded in a formal set of work practice principles called *Activity Theory*, to support the modelling, analysis and enactment of business processes. This approach directly provides for dynamic change and process evolution without having to resort to off-system intervention and/or system downtime. It has been implemented as a discrete service for the well-known, open-source workflow environment YAWL [15, 16] using a Service Oriented Architecture (SOA), and as such its applicability is not limited to that environment. Also, being open-source, it is freely available for use and extension.

The paper is organised as follows: Section 2 provides a brief overview of Activity Theory and lists relevant principles derived from it by the authors, then introduces the worklet paradigm. Section 3 describes the implementation of the discrete worklet service. Section 4 details the worklet service architecture. Section 5 discusses process definition methods, while Section 6 describes how the worklet approach utilises *Ripple Down Rules* (RDR) to achieve contextual, dynamic selection of worklets at runtime. Section 7 discusses related work, and finally Section 8 outlines future directions and concludes the paper.

2 Achieving Flexibility through Worklets

Workflow management systems provide support for business processes that are generally predictable and repetitive. However, the prescriptive, assembly-line frameworks imposed by workflow systems limit the ability to model and enact flexible work practices where deviations are a normal part of every work activity [12, 17]. For these environments, formal representations of business processes may be said to provide merely a contingency around which tasks can be formulated dynamically [18], rather than a prescriptive blueprint that must be strictly adhered to.

Rather than continue to try to force business processes into inflexible frameworks (with limited success), a more adaptable approach is needed that is based on accepted ideas of how people actually work.

A powerful set of descriptive and clarifying principles that describe how work is conceived, performed and reflected upon is *Activity Theory*, which focusses on understanding human activity and work practices, incorporating notions of intentionality, history, mediation, collaboration and development [19]. (A full exploration of Activity Theory can be found in [20, 21]). In [22], the current authors undertook a detailed study of Activity Theory and derived from it a

set of principles that describe the nature of participation in organisational work practices. Briefly, the relevant principles are:

1. Activities (i.e. work processes) are *hierarchical* (consist of one or more actions), *communal* (involve a community of participants working towards a common objective), *contextual* (conditions and circumstances deeply affect the way the objective is achieved), *dynamic* (evolve asynchronously), and *mediated* (by tools, rules and divisions of labour).
2. Actions (i.e. tasks) are undertaken and understood contextually. A *repertoire* of applicable actions is maintained and made available for each action of an activity; the activity is performed by making contextual choices from the repertoire of each action in turn.
3. A work plan is not a prescription of work to be performed, but merely a guide which may be modified during execution depending on context.
4. Deviations from a plan will naturally occur with every execution, giving rise to learning experiences which can then be incorporated into future instantiations of the plan.

Consideration of these derived principles have led to the conception, development and implementation of a flexible workflow support system that:

- regards the process model as a guide to an activity's objective, rather than a prescription for it;
- provides a repertoire (or catalogue) of applicable actions to be made available for each task at each execution of a process model;
- provides for choices to be made dynamically from the repertoire at runtime by considering the specific context of the executing instance; and
- allows the repertoire of actions to be dynamically extended at runtime, thus incorporating unexpected process deviations, not only for the current instance, but for other current and future instantiations of the process model, leading to natural process evolution.

Thus, each task of a process instance may be linked to an extensible repertoire of actions, one of which will be contextually chosen at runtime to carry out the task. In this work, we present these repertoire-member 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)[1]. A top-level or parent process model is developed that captures the entire workflow at a macro level. From that manager process, worklets are contextually selected and invoked from the repertoire of each task when the task instance becomes enabled during execution.

In addition, 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, derived from the context of each process instance.

[1] In Activity Theory terms, a worklet may represent one action within an activity, or may represent an entire activity.

Importantly, the new worklet becomes part of the process model for all current and future instantiations, avoiding issues of version control. In this way, the process model undergoes a dynamic natural evolution.

3 The Worklet Custom Service for YAWL

The *Worklet Dynamic Process Selection Service* has been implemented as a YAWL Custom Service [15, 16]. The YAWL environment was chosen as the implementation platform since it provides a very powerful and expressive workflow language based on the workflow patterns identified in [23], together with a formal semantics. It also provides a workflow enactment engine, and an editor for process model creation, that support the control flow, data and (basic) resource perspectives. The YAWL environment is open-source and has a service-oriented architecture, allowing the worklet paradigm to be developed as a service independent to the core engine. Thus the deployment of the worklet service is in no way limited to the YAWL environment, but may be ported to other environments by making the necessary amendments to the service interface. As such, this implementation may also be seen as a case study in service-oriented computing whereby dynamic flexibility in workflows, orthogonal to the underlying workflow language, is provided.

Custom YAWL services interact with the YAWL engine through XML/HTTP messages via certain interface endpoints, some located on the YAWL engine side and others on the service side. Specifically, custom services may elect to be notified by the engine when certain events occur in the life-cycle of nominated process instantiations (i.e. when a workitem becomes enabled, when a workitem is cancelled, when a case completes). On receiving a workitem-enabled event, the custom service may elect to 'check-out' the workitem from the engine. On doing so, the engine marks the workitem as *executing* and effectively passes operational control for the workitem to the custom service. When the custom service has finished processing the workitem it will check it back in to the engine, at which point the engine will mark the workitem as *completed*, and proceed with the process execution.

The worklet service utilises these interactions by *dynamically substituting an enabled workitem in a YAWL process with a contextually selected worklet* – a discrete YAWL process that acts as a sub-net for the workitem and so handles one specific task in a larger, composite process activity.

An *extensible repertoire (or catalogue) of worklets is maintained for each nominated task in a parent workflow process.* Each time the service is invoked for an enabled workitem, a choice is made from the repertoire based on the data attributes and values associated with the workitem, using a set of rules to determine the most appropriate substitution (see Section 6). The workitem is checked out of the YAWL engine, the input variables of the original workitem are mapped to the net-level input variables of the selected worklet, and then the worklet is launched in the engine as a separate case. When the worklet has completed, its net-level output variables are mapped back to the output variables

of the original workitem, which is then checked back into the engine, allowing the original (parent) process to continue.

The worklet executed for a task is run as a separate case in the YAWL engine, so that, from an engine perspective, the worklet and its parent are two distinct, unrelated cases. The worklet service tracks the relationships, data mappings and synchronisations between cases, and creates a process log that may be combined with the engine's process logs via case identifiers to provide a complete operational history of each process.

Worklets may be associated with either an atomic task, or a multiple-instance atomic task. Any number of worklets can form the repertoire of an individual task, and any number of tasks in a particular specification can be associated with the worklet service. A worklet may be a member of one or more repertoires – that is, it may be re-used for several distinct tasks within and across process specifications. In the case of multiple-instance tasks, a separate worklet is launched for each child workitem. Because each child workitem may contain different data, the worklets that substitute for them are individually selected, and so may all be different.

The repertoire of worklets for a task can be added to at any time, as can the rules base used for the selection process, including while the parent process is executing. Thus the service provides for dynamic ad-hoc change and process evolution, without having to resort to off-system intervention and/or system downtime, or requiring modification of the original process specification.

4 Worklet Service Architecture

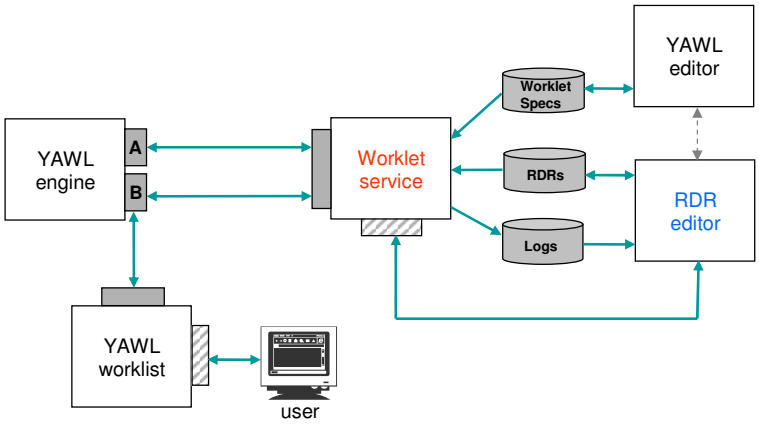

Fig. 1. External Architecture of the Worklet Service

Figure 1 shows the external architecture of the worklet service. As mentioned previously, the service has been implemented as a Custom YAWL Service

[16]. The YAWL engine provides a number of interfaces, two of which are used by the worklet service. Interface A provides endpoints for process definition, administration and monitoring; Interface B provides endpoints for client and invoked applications and workflow interoperability [16]. The worklet service uses Interface A to upload worklet specifications into the engine, and Interface B for connecting to the engine, to start and cancel case instances, and to check workitems in and out of the engine after interrogating their associated data.

The disk entities 'Worklet specs', 'RDRs' and 'Logs' in Figure 1 comprise the *worklet repository*. The service uses the repository to store rule sets and load them for enabled workitems; to store worklet specifications for uploading to the engine; and to store generated process and audit logs. The YAWL editor is used to create new worklet specifications, and may be invoked from the RDR (Ripple Down Rules) Editor. The RDR Editor is used to create new or augment existing rule sets, making use of certain selection logs to do so, and may communicate with the worklet service via a JSP/Servlet interface to override worklet selections following rule set additions (see Section 6).

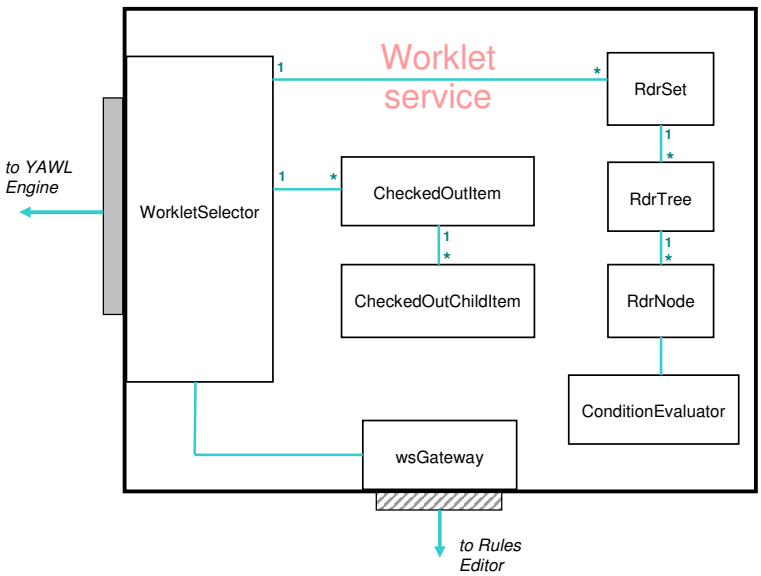

Fig. 2. Internal Architecture of the Worklet Service

Figure 2 shows a representation of the internal architecture of the worklet service. The *WorkletSelector* object handles all interactions with the YAWL engine, and administrates the service. For each workitem that it checks out of the engine, it creates a *CheckedOutItem* object. In YAWL, each workitem is a 'parent' of one or more child items – one if it is an atomic task, or a number of child items in the case of a multiple instance atomic task. Thus,

the role of each CheckedOutItem object is to create and manage one or more *CheckedOutChildItems*, which hold information about worklet selection, data associated with the workitem and the results of rules searches.

The WorkletSelector, for each workitem that is checked out from the engine, also loads from file the set of rules pertaining to the specification of which the workitem is a member into an *RdrSet* object. At any time, there may be a number of RdrSets loaded into the service, one for each specification for which a workitem has been checked out. Each RdrSet manages one or more *RdrTree* objects, each tree representing the rule tree for a particular task within the specification, of which this workitem is an instance. In turn, each RdrTree owns a number of *RdrNode* objects, which contain the actual rules, conclusions and other data for each node of the rule tree.

When a rule tree is evaluated against the data set of a workitem, each of the associated nodes of that tree has its condition evaluated by the *ConditionEvaluator* object, which returns the boolean result to the node, allowing it to traverse to its true or false branch as necessary. Finally, the *wsGateway* object provides communications via a JSP/Servlet interface between the service and the Rules Editor (see Section 6 for more details).

5 Process Definition

Fundamentally, a worklet is nothing more than a workflow specification that has been designed to perform one part of a larger, parent specification. However, it differs from a decomposition or sub-net in that it is dynamically assigned to perform a particular task at runtime, while sub-nets are statically assigned at design time. So, rather than being forced to define all possible branches in a specification, the worklet service allows the definition of a much simpler specification that will evolve dynamically as more worklets are added to the repertoire for particular tasks within it.

Figure 3 shows a simple example specification (in the YAWL Editor) for a Casualty Treatment process. Note that this process specification has been intentionally simplified to demonstrate the operation of the worklet service; while it is not intended to portray a realistic process, it is desirable to not camouflage the subject of this paper by using a more complex process specification.

In this process, the *Treat* task is to be substituted at runtime with the appropriate worklet based on the patient data collected in the *Admit* and *Triage* tasks. That is, depending on each patient's actual physical data and reported symptoms, we would like to run the worklet that best treats the patient's condition.

Each task in a process specification may be flagged to notify the worklet service when it becomes enabled. In this example, only the *Treat* task is flagged so; the other tasks are handled directly by the YAWL environment. So, when a Casualty Treatment process is executed, the YAWL Engine will notify the worklet service when the *Treat* task becomes enabled. The worklet service will

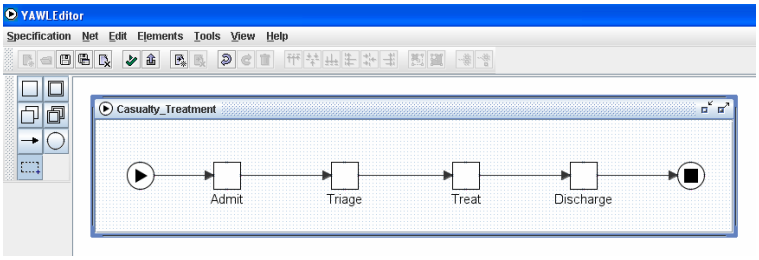

Fig. 3. Parent 'Casualty Treatment' Process

then examine the data of the task and use it to determine which worklet to execute as a substitute for the task.

A worklet specification is a standard YAWL process specification, and as such is created in the YAWL Editor in the usual manner. Figure 4 shows a very simple example worklet to be substituted for the *Treat* top-level task when a patient complains of a fever.

In itself, there is nothing special about the Treat Fever specification in Figure 4. Even though it will be considered by the worklet service as a member of the worklet repertoire and may thus be considered a "worklet", it also remains a standard YAWL specification and as such may be executed directly by the YAWL engine without any reference to the worklet service, if desired.

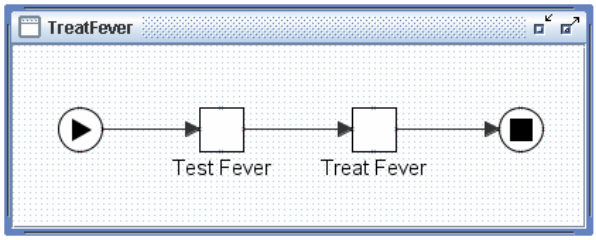

Fig. 4. The 'Treat Fever' Worklet Process

The association of tasks with the worklet service is not restricted to top-level specifications. Worklet specifications also may contain tasks that are associated with the worklet service and so may have worklets substituted for them, so that a hierarchy of executing worklets may sometimes exist. It is also possible to recursively define worklet substitutions - that is, a worklet may contain a task that, while certain conditions hold true, is substituted by another instance of the same worklet specification that contains the task.

Any number of worklets can be created for a particular task. For the Casualty Treatment example, there are currently five worklets in the repertoire for the

Treat task, one for each of the five conditions that a patient may present with in the *Triage* task: Fever, Rash, Fracture, Wound and Abdominal Pain. In this example, which worklet is chosen for the *Treat* task depends on which of the five is given a value of True in the *Triage* task.

6 Context and Worklet Selection

The consideration of context plays a crucial role in many diverse domains, including philosophy, pragmatics, semantics, cognitive psychology and artificial intelligence [24]. In order to realise the worklet approach, the situated contextual factors relevant to each case instance were required to be quantified and recorded [25] so that the appropriate worklet can be 'intelligently' selected from the repertoire at runtime.

The types of contextual data that may be recorded and applied to a business case may be categorised as follows (examples are drawn from the *Casualty Treatment* 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 created when, 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:** that 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: patient name and id, blood pressure readings, height, weight, symptoms and so on; deadlines both approaching and expired; and diagnoses, treatments and prescribed medications.
- **Case dependent with no *a-priori* knowledge:** that 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 in a patient's condition after triage, allergies to certain medications and so on.

Each worklet is a representation of a particular situated action, the runtime selection of which relies on the relevant context of each case instance, derived from case data. The worklet selection process is achieved through the use of Ripple Down Rules (RDR), which comprise a hierarchical set of rules with associated exceptions, first devised by Compton and Jansen [26].

The fundamental feature of RDR is that it avoids the difficulties inherent in attempting to compile, *a-priori*, a systematic understanding, organisation and assembly of all knowledge in a particular domain. Instead, it allows for general rules to be defined first with refinements added later as the need arises [27].

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. 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 subsequent rule is evaluated; if it is not satisfied, the false branch is taken and its rule evaluated [28]. 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. This tree traversal provides implied *locality* - a rule on an exception branch is tested for applicability only if its parent (next-general) rule is also applicable.

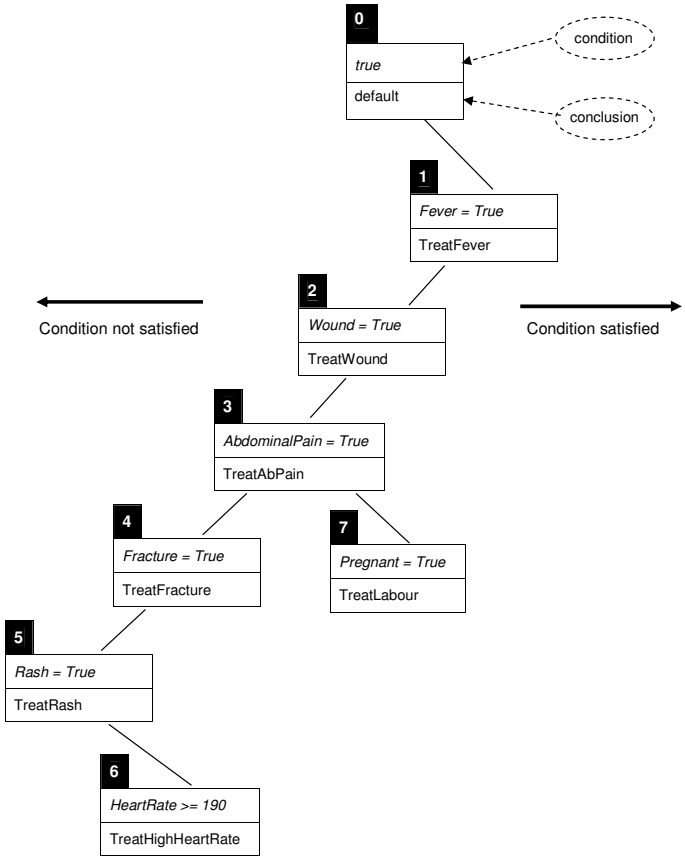

Fig. 5. Conceptual Structure of a Ripple Down Rule (*Casualty Treatment* Example)

A workflow process specification may contain a number of tasks, one or more of which may be associated with the worklet service. For each specification that

contains a worklet-enabled task, the worklet service maintains a corresponding set of ripple down rules that determine which worklet will be selected as a substitute for the task at runtime, based on the current case data of that particular instance. Each worklet-enabled task in a specification has its own discrete rule set. The rule set or sets for each specification are stored as XML data in a disk file that has the same name as the specification, except with an ".xrs" extension (XML Rule Set). All rule set files are stored in the worklet repository.

Occasionally, the worklet started as a substitute for a particular workitem, while the correct choice based on the current rule set, is considered by a user to be an inappropriate choice for a particular case. For example, if a patient in a Casualty Treatment case presents with a rash *and* a heart rate of 190, while the current rule set correctly returns the TreatRash worklet, it may be more desirable to treat the racing heart rate before the rash is attended to. In such a case, when the worklet service begins an instance of the TreatRash process, a user may reject it by advising an administrator (via a button on their worklist) of the inappropriate choice. Thus the administrator would need to add a new rule to the rule set so that cases that have such data (both now and in the future) will be handled correctly.

If the worklet returned is found to be unsuitable for a particular case instance, 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 worklet returned was the conclusion of a satisfied terminal rule, then the new rule is added as a local exception node via a new true branch from the terminal node.
- If the worklet returned was the conclusion of a non-terminal, ancestor node (that is, the condition of the terminal rule was not satisfied), then the new rule node 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 KB by "sub-domain" 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 [29].

Each rule node also incorporates a set of case descriptors that describe the actual case that was the catalyst for the creation of its rule. This case is referred to as the 'cornerstone case'. The descriptors of the cornerstone case refer to essential attributes of a case, in this example, the sex, heart rate, age, weight and so on of a patient. The condition for the new rule is determined by comparing the descriptors of the current case to those of the cornerstone case of the returned worklet and identifying a sub-set of differences. Not all differences will be relevant – to define a new rule 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. The identified differences are expressed as attribute-value pairs, using the normal conditional operators. The current case descriptors become the cornerstone case for the newly formulated rule; its condition is formed by the

identified attribute-values and represents the context of the case instance that caused the addition of the rule.

A separate Rules Editor tool has been developed to allow for the easy addition of new rules and associated worklets to existing rule sets, and the creation of new rule sets.

Each time the worklet service selects a worklet to execute as a substitute for a specification instance's workitem, a file is created that contains certain descriptive data about the selection process. These files are stored in the worklet repository, again in XML format. Thus to add a new rule to the existing rule set after an inappropriate selection, the particular selection file for the case that was the catalyst for the rule addition is first loaded into the Rules Editor.

Figure 6 shows the Add New Rule screen of the Rules Editor with a selection file loaded. The Cornerstone Case panel shows the case data that existed for the creation of the original rule for the TreatRash selection. The Current Case panel shows the case data for the current case - that is, the case that is the catalyst for the addition of the new rule. The New Rule Node panel is where the details of the new rule are added. Notice that the ids of the parent node and the new node are shown as read only - the Rules Editor takes care of where in the rule tree the new rule node is to be placed, and whether it is to be added as a true child or false child node, using the algorithm described above.

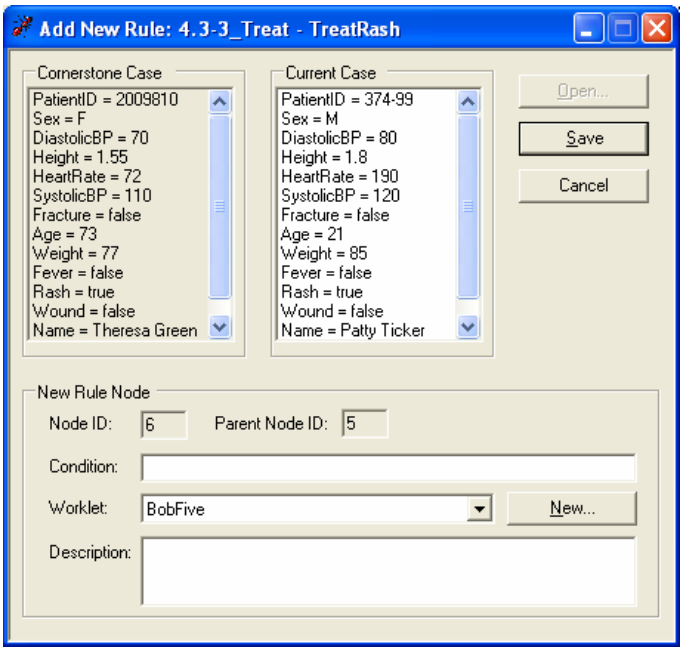

Fig. 6. Rules Editor (Add New Rule Screen)

In this example, there are many data values that differ between the two case data sets shown in Figure 6, such as PatientID, Name, Sex, Blood Pressure readings, Height, Weight and Age. However, the only differing data item of relevance here is HeartRate - that is the only data item that, in this case, makes the selection of the TreatRash worklet inappropriate. Selecting the HeartRate line in the list of Current Case data items will insert it to the condition field, where it may be modified as necessary. In this case, the new rule would become, as an example, "HeartRate $\geq$ 190".

It is not necessary to define a conjunctive rule such as "Rash = True AND HeartRate $\geq$ 190", since this new rule will be added as an exception to the true branch of the TreatRash node. By doing so, it will only be evaluated if the condition of its parent, "Rash = True", first evaluates to True. Therefore, any rule nodes added to the true branch of a parent node become *exception* rules, and thus refinements, of the parent rule.

After defining a condition for the new rule, the name of the worklet to be executed when this condition evaluates to true must be entered in the Worklet field of the Editor (refer Figure 6). This input is a drop-down list that contains the name of all the worklets currently in the worklet repository. An appropriate worklet for this rule may be chosen from the list, or, if none are suitable, a new worklet specification may be created.

After a new rule is added, the Editor provides an administrator with the choice to replace the previously started (inappropriate) worklet instance with an instance of the worklet defined in the new rule. If the administrator chooses to replace the worklet, the Rules Editor contacts the worklet service via HTTP and requests the change. The service responds with a dialog similar to Figure 7.

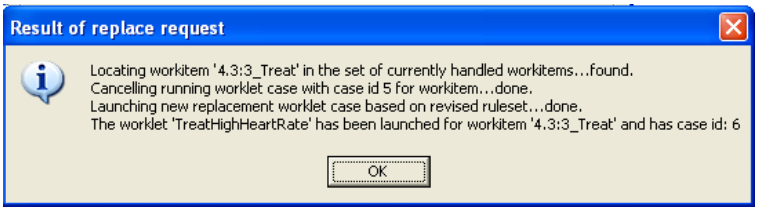

Fig. 7. Example Dialog Showing a Successful Dynamic Replacement

7 Related Work

Since the mid-nineties much research has been done on issues related to flexibility and change in workflow management systems (cf. the classification into ad-hoc, administrative, and production workflows in [30]). While it is not the intention of this paper to provide a complete overview of the work done in this area,

reference is made here to a number of quite different approaches to providing dynamic flexibility in workflows.

Generally, commercial workflow management systems provide various levels of support for the decomposition of tasks and sub-processing. However, each of the products require the model to be fully defined before it can be instantiated, and changes must be incorporated by modifying the model statically. Staffware provides 're-usable process segments' that can be inserted into any process. SAP R/3 allows for the definition of 'blocks' that can be inserted into other 'blocks', thus providing some support for encapsulation and reuse. COSA supports parent-sibling processes, where data can be passed to/from a process to a sub-process. MQ Workflow allows sub-processes to be defined and called statically from within a process. Clearly, all of these static forms of decomposition do not offer support for dynamic flexibility.

Among the non-commercial systems, *ADEPT* [31] supports modification of a process during execution (i.e. add, delete and change the sequence of tasks) both at the type (dynamic evolution) and instance levels (ad-hoc changes). Such changes are made to a traditional monolithic model and must be achieved via manual intervention. The *WASA* [32] system provides some support for dynamic change, mainly focusing on scientific applications. It allows an administrator to modify a (monolithic) specification and then restart a task, but then only at the instance level. A catalog of 'skeleton' patterns that can be instantiated or specialised at design time is supported by the *WERDE* system [5]. Again, there is no scope for specialisation changes to be made at runtime. *AgentWork* [33] 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 Ripple Down Rules, and changes are limited to individual tasks, rather than the process-for-task substitution provided by the worklet service. Also, the possibility exists for conflicting rules to generate incompatible actions, which requires manual intervention and resolution.

It should be noted that only a small number of academic prototypes have had any impact on the frameworks offered by commercial systems [34]. Nevertheless, there are some interesting commercial products that offer innovative features with respect to flexibility. *Caramba* [35] supports virtual teams in their ad hoc and collaborative processes by enabling links between artifacts (for example, documents and database objects), business processes (activities), and resources (persons, roles, etc.). *FLOWer* supports the concept of case-handling; the process model only describes the preferred way of doing things and a variety of mechanisms are offered to allow users to deviate in a controlled manner [1].

The implementation discussed in this paper differs considerably from the above approaches. Worklets dynamically linked together by extensible Ripple Down Rules provide an alternative method for the provision of dynamic flexibility. An approach with some similarities to worklets is is the *Process Orchestrator*, an optional component of Staffware [36], which provides for the dynamic allocation of sub-processes at runtime. It requires a construct called a "dynamic event" to be explicitly modelled that will execute a number of sub-processes listed in an

'array' when execution reaches that event. Which sub-processes execute depend on predefined data conditionals matching the current case. Unlike the worklet approach, the listed sub-processes are statically defined, as are the conditionals – there is no scope for dynamically refining conditionals, nor adding sub-processes at runtime.

8 Conclusion and Future Work

Workflow management systems impose a certain rigidity on process definition and enactment because they use frameworks based on assembly line metaphors rather than on ways work is actually planned and carried out. An analysis of Activity Theory provided principles of work practices that were used as a template on which a workflow service has been built that better supports flexibility and dynamic evolution. By capturing contextual data, a repertoire of actions is constructed that allow for contextual choices to be made from the repertoire at runtime to efficiently carry out work tasks. These actions, or worklets, directly provide for process evolution and flexibility, and mirror accepted work practices.

The worklet implementation presents several key benefits, including:

- A process modeller can describe the standard activities and actions for a workflow process, and any deviations, using the same methodology;
- It allows re-use of existing process components and aids in the development of fault tolerant workflows using pre-existing building blocks [37];
- Its modularity simplifies the logic and verification of the standard model, since individual worklets are less complex to build and therefore easier to verify than monolithic models;
- It provides for a variety of 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 each time a business practice changes or a deviation occurs is unnecessary; and
- In the occurrence of an unexpected event, the process modeller needs simply to choose an existing worklet or build a new one for that event, which can be automatically added to the repertoire for current and future use as necessary, thus avoiding manifold complexities including downtime, model restructuring, versioning problems and so on.

This implementation used the open-source, service-oriented architecture of YAWL to develop a service for dynamic flexibility independent to the core engine. Thus, the implementation may be viewed as a successful case study in service-oriented computing. It is the first instalment of a comprehensive approach to dynamic workflow and is intended to be extended in the near future to also provide support for dynamic handling of process exceptions using the same service paradigm. One of the more interesting things to be investigated and incorporated is the application of process mining techniques to the various logs collected by the Worklet service; a better understanding of when and why

people tend to "deviate" from a work plan is essential for providing better tool support.

All system files, source code and documentation for YAWL and the worklet service, including the examples discussed in this paper, may be downloaded via `www.yawl-system.com`.

References

1. W.M.P. van der Aalst, Mathias Weske, and Dolf Grünbauer. Case handling: A new paradigm for business process support. *Data & Knowledge Engineering*, 53(2):129–162, 2005.
2. Gregor Joeris. Defining flexible workflow execution behaviors. In Peter Dadam and Manfred Reichert, editors, *Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications*, volume 24 of *CEUR Workshop Proceedings*, Paderborn, Germany, October 1999.
3. Alex Borgida and Takahiro Murata. Tolerating exceptions in workflows: a unified framework for data and processes. In *Proceedings of the International Joint Conference on Work Activities, Coordination and Collaboration (WACC'99)*, pages 59–68, San Francisco, CA, February 1999. ACM Press.
4. S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems: A survey. *Data and Knowledge Engineering*, 50(1):9–34, 2004.
5. Fabio Casati. A discussion on approaches to handling exceptions in workflows. In *CSCW Workshop on Adaptive Workflow Systems*, Seattle, USA, November 1998.
6. C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In N. Comstock, C. Ellis, R. Kling, J. Mylopoulos, and S. Kaplan, editors, *Proceedings of the Conference on Organizational Computing Systems*, pages 10–21, Milpitas, California, August 1995. ACM SIGOIS, ACM Press, New York.
7. Claus Hagen and Gustavo Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, October 2000.
8. Mark S. Ackerman and Christine Halverson. Considering an organization's memory. In *Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work*, pages 39–48. ACM Press, 1998.
9. Peter A. K. Larkin and Edward Gould. Activity theory applied to the corporate memory loss problem. In L. Svennson, U. Snis, C. Sorensen, H. Fagerlind, T. Lindroth, M. Magnusson, and C. Ostlund, editors, *Proceedings of IRIS 23 Laboratorium for Interaction Technology*, University of Trollhattan Uddevalla, 2000.
10. W.M.P. van der Aalst. Exterminating the dynamic change bug: A concrete approach to support workflow change. *Information Systems Frontiers*, 3(3):297–317, 2001.
11. Jakob E. Bardram. I love the system - I just don't use it! In *Proceedings of the 1997 International Conference on Supporting Group Work (GROUP'97)*, Phoenix, Arizona, 1997.
12. W.M.P. van der Aalst and P.J.S. Berens. Beyond workflow management: Product-driven case handling. In S. Ellis, T. Rodden, and I. Zigurs, editors, *International ACM SIGGROUP Conference on Supporting Group Work*, pages 42–51, New York, 2001. ACM Press.
13. I. Bider. Masking flexibility behind rigidity: Notes on how much flexibility people are willing to cope with. In J. Castro and E. Teniente, editors, *Proceedings of the CAiSE'05 Workshops*, volume 1, pages 7–18, Porto, Portugal, 2005. FEUP Edicoes.

14. Michael Adams, Arthur H. M. ter Hofstede, David Edmond, and Wil M.P. van der Aalst. Facilitating flexibility and dynamic exception handling in workflows through worklets. In Orlando Bello, Johann Eder, Oscar Pastor, and João Falcão e Cunha, editors, *Proceedings of the CAiSE'05 Forum*, pages 45–50, Porto, Portugal, June 2005. FEUP Edicoes.
15. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
16. W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and implementation of the YAWL system. In A. Persson and J. Stirna, editors, *Proceedings of The 16th International Conference on Advanced Information Systems Engineering (CAiSE 04)*, volume 3084 of *LNCS*, pages 142–159, Riga, Latvia, June 2004. Springer Verlag.
17. Diane M. Strong and Steven M. Miller. Exceptions and exception handling in computerized information processes. *ACM Transactions on Information Systems*, 13(2):206–233, 1995.
18. Jakob E. Bardram. Plans as situated action: an Activity Theory approach to workflow systems. In *Proceedings of the 1997 European Conference on Computer Supported Cooperative Work (ECSCW'97)*, pages 17–32, Lancaster U.K., 1997.
19. Bonnie A. Nardi. *Activity Theory and Human-Computer Interaction*, pages 7–16. In Nardi [21], 1996.
20. Y. Engestrom. *Learning by Expanding: An Activity-Theoretical Approach to Developmental Research.* Orienta-Konsultit, Helsinki, 1987.
21. Bonnie A. Nardi, editor. *Context and Consciousness: Activity Theory and Human-Computer Interaction.* MIT Press, Cambridge, Massachusetts, 1996.
22. Michael Adams, David Edmond, and Arthur H.M. ter Hofstede. The application of activity theory to dynamic workflow adaptation issues. In *Proceedings of the 2003 Pacific Asia Conference on Information Systems (PACIS 2003)*, pages 1836–1852, Adelaide, Australia, July 2003.
23. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.
24. Paolo Bouquet, Chiara Ghidini, Fausto Giunchiglia, and Enrico Blanzieri. Theories and uses of context in knowledge representation and reasoning. *Journal of Pragmatics*, 35(3), 2003.
25. Debbie Richards. Combining cases and rules to provide contextualised knowledge based systems. In *Modeling and Using Context, Third International and Interdisciplinary Conference, CONTEXT 2001*, volume 2116 of *Lecture Notes in Artifical Intelligence*, pages 465–469, Dundee, UK, July 2001. Springer-Verlag, Berlin.
26. P. Compton and B. Jansen. Knowledge in context: A strategy for expert system maintenance. In J.Siekmann, editor, *Proceedings of the 2nd Australian Joint Artificial Intelligence Conference*, volume 406 of *Lecture Notes in Artificial Intelligence*, pages 292–306, Adelaide, Australia, November 1988. Springer-Verlag.
27. Tobias Scheffer. Algebraic foundation and improved methods of induction of ripple down rules. In *Procceedings of the Pacific Rim Workshop on Knowledge Acquisition*, Sydney, Australia, 1996.
28. B. Drake and G. Beydoun. Predicate logic-based incremental knowledge acquisition. In P. Compton, A. Hoffmann, H. Motoda, and T. Yamaguchi, editors, *Proceedings of the sixth Pacific International Knowledge Acquisition Workshop*, pages 71–88, Sydney, December 2000.
29. Byeong Ho Kang, Phil Preston, and Paul Compton. Simulated expert evaluation of multiple classification ripple down rules. In *Proceedings of the 11th Workshop on*

Knowledge Acquisition, Modeling and Management, Banff, Alberta, Canada, April 1998.
30. Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modelling to workflow automation infrastructure. In *Distributed and Parallel Databases*, volume 3, pages 119–153. Kluwer Academic Publishers, Boston, 1995.
31. Clemens Hensinger, Manfred Reichert, Thomas Bauer, Thomas Strzeletz, and Peter Dadam. ADEPT$_{workflow}$ - advanced workflow technology for the efficient support of adaptive, enterprise-wide processes. In *Conference on Extending Database Technology*, Konstanz, Germany, March 2000.
32. G. Vossen and M. Weske. The WASA approach to workflow management for scientific applications. In A. Dogac, L. Kalinichenko, M.T. Ozsu, and A. Sheth, editors, *Workflow Management Systems and Interoperability*, volume 164 of *ASI NATO Series, Series F: Computer and Systems Sciences*, pages 145–164. Springer, 1999.
33. Robert Muller, Ulrike Greiner, and Erhard Rahm. AgentWork: a workflow system supporting rule-based workflow adaptation. *Data & Knowledge Engineering*, 51(2):223–256, November 2004.
34. Michael zur Muehlen. *Workflow-based Process Controlling. Foundation, Design, and Implementation of Workflow-driven Process Information Systems*, volume 6 of *Advances in Information Systems and Management Science*. Logos, Berlin, 2004.
35. S. Dustdar. Caramba - a process-aware collaboration system supporting ad hoc and collaborative processes in virtual teams. *Distributed and Parallel Databases*, 15(1):45–66, 2004.
36. Michael Georgeff and Jon Pyke. Dynamic process orchestration. White paper, Staffware PLC `http://is.tm.tue.nl/bpm2003/download/WP%20Dynamic%20Proce%ss%20Orchestration%20v1.pdf`, March 2003.
37. Claus Hagen and Gustavo Alonso. Flexible exception handling in process support systems. Technical report no. 290, ETH Zurich, 1998.