# An Extensible Workflow Design Kit

Markus Bon, Jürgen Zimmermann
Universität Kaiserslautern
(bon, jnzimmer)@informatik.uni-kl.de

**Abstract:** Using workflows is a quite convenient matter. The right piece of work is done at the right time by the right person. Unfortunately, many workflow management systems are tailored to specific tasks and (in spite of nice GUIs) hard to handle for non-IT-specialists. However, most of the analysts/designers are non-IT-people. In this paper, we introduce a mechanism which provides the possibility of building workflow systems "as needed" in a very flexible way. We present a macro component offering building blocks to the workflow designer, so he may easily compose a workflow by adapting predefined elements. New building blocks may be added by the IT specialists on demand.

**keywords:** Workflow, extensibility, macros

## 1. Introduction & Motivation

Workflow management has grown to everybody's attention, as it supports the execution and coordination of complex business processes in a handy manner. Most business processes may be structured very well, but nevertheless, producing an adequate description is non an easy task to perform, mainly because of the large amount of steps that have to be accomplished in the modelling process. As more and more workflow management systems (WfMS) are available ([7], [8], [9], [10]), the fact that a common workflow model is missing becomes obvious. Where a given system and its workflow model prevail in a primary target domain (e. g., office automation), they fail or become hard to utilize in another one, e. g., manufacturing, due to deficiencies in their workflow model.

A number of organizations joined the "Workflow Management Coalition" (WfMC), which tries to establish some common standards for WfMSs [11]. Most of these standards origin from analysing and merging the concepts found in existing WfMSs. As a result, a "reference model" for a WfMS has been established, which consists of build-time tools, a runtime environment, and a user-interaction abstraction.

For the definition of workflows some kind of workflow definition language (WDL) is provided by the WfMSs. Although these languages are sufficient for representing a business process as a workflow, they are not very comfortable to use. Complex dependencies have to be expressed with simple constructs. Unfortunately, knowledge about business processes is provided by employees which are no trained IT-people. Asking them to describe the process using the WDL is like demanding a bookkeeper to write the software he needs for work. On the other hand, the IT-people knowing how to use the WDL have no detailed knowledge about the business processes. Therefore, an extended WDL is needed which may easily be used by non-IT-people, powerful enough to express complex dependencies in a simple way, easy to be tailored for special interests and extendible for further tasks unknown by now.

In short, a useful WDL has to meet the following demands:

- easy to use and to maintain
- expressive power
- extensible
- adaptable
- independent of underlying WfMS

Offering these properties, the WDL is likely to be accepted by non-IT-people, as they may focus on their actual job – to build an appropriate representation of a real life process – without struggling with cryptic languages. In fact, they get exactly the language *they* want. Being independent of the actually used WfMS, the modelled workflows have not to be changed even if the WFMS is exchanged. On the other hand, the IT-people may concentrate on the technical aspects like implementing a proper mapping of the macro language to the WDL provided by the underlying WfMS(s). Therefore, everybody does the job he is qualified for.

The Arktis (A reliable kernel for technical information systems) approach used in our own prototypical WfMS (see Figure 1) is to provide an extensible workflow model consisting of building blocks ("macros") for workflows and methods to combine them. The macros are described in an abstract manner, only their functional interface is known, implementation details are hidden to the engineers who specify the workflows. The combination methods for macros enforce sound macro usage in workflow definitions. The implementation of the macros and their combination methods lie in the responsibility of the IT-specialist. Furthermore, using macros hides the WfMS actually used. Even though the underlying WfMS is changed, the workflow definition using macros may remain unmodified.

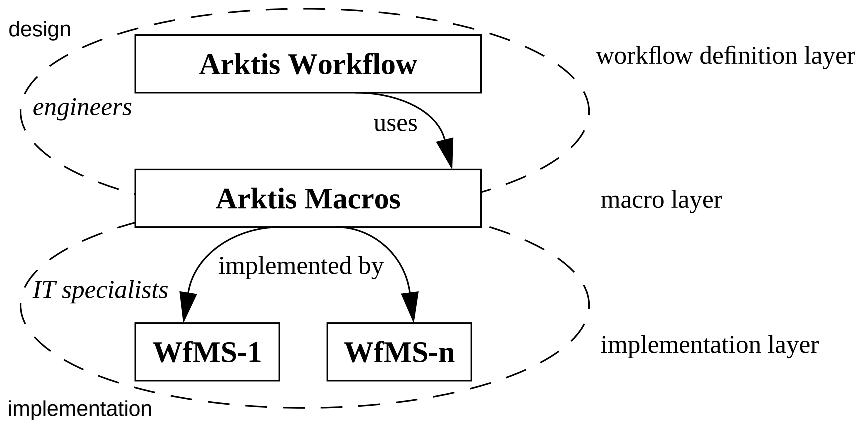


Figure 1 The multiple layers of Arktis

As different Wf-designers frequently identify different building blocks or combination methods as essential for the way they describe their workflows, extensibility is extremely important for this approach to be accepted. Therefore, the building blocks are not hard-coded in the Arktis approach, but are provided by a macro component. Hence, they are configurable and extensible at any time, which leads to a high degree of adaptability for easy deployment and refinement of workflows in different domains. The macros themselves can have algorithmic elements, so that an entire problem class can be subsumed by one macro. Given the macros, everything (simple or complex) needed by a Wf-designer (even IT-specialist) can be pre-build by a few IT-specialists.

In analogy to the previously mentioned programming languages, the macro component allows for new combinations of existing macros without exposing the implementation details. In addition, it facilitates the introduction of new macros needed to face (dynamically occurring) special problems which cannot be solved with the existing macros. Consequently, the collection of all available macros forms a new WDL, which meets the demands we stated earlier.

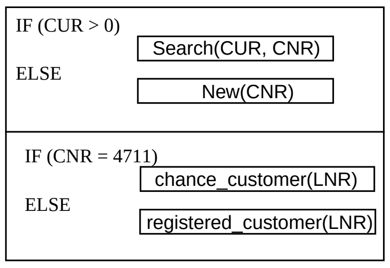
In this paper, we concentrate on the macro abstraction and its implementation in the macro component. We describe the abstractions provided by macros along with a definition language for macros. Before we give a conclusion and an outlook, we refer to some related work.

## 2. Macro Abstraction

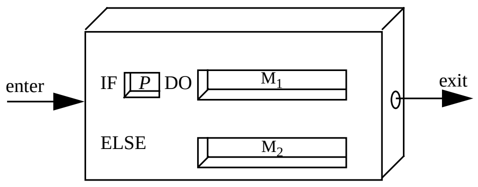
The macro component — the server which provides the macro abstraction — is one of the most important parts of the Arktis architecture with respect to flexibility, reusability and expressive power. It offers a set of building blocks which can easily be used without bothering about implementation details. All the analyst/workflow designer has to do is to select appropriate macros and adjust the parameters. No IT-specific knowledge is necessary.

In order to accomplish this, the macro component must at first abstract from implementation details of the underlying workflow engine, so that the workflow designer has only to deal with one workflow definition language, namely the one provided by the macro component. When for example the workflow designer wants to express alternative execution paths in the workflow design, he just uses a design element “IF” and adjusts the parameters accordingly, regardless of the target workflow engine and the design constructs offered by this engine.

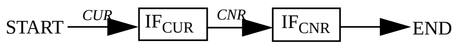
But how do we identify those design elements? Figure 2a shows a short excerpt of a workflow instance. Using the value of CUR, an existing customer number is searched or a new one is created, respectively. In the next step, this customer number is used to decide how to proceed. Obviously, the two IF-blocks are very similar to each other, so we extract the common elements and offer them as a template to be adopted by passing adequate parameters. These templates are called **macro bodies**, the instantiation is named **macro**. Macro bodies are workflow modules which may be used in a very flexible way. Nevertheless, they differ from sub-workflows in a fundamental point: macros describe the inner struc-



a) excerpt from a workflow description



b) macro body IF



c) workflow represented by a macro chain

Figure 2 Transforming a workflow description into a macro chain

ture of a workflow, like e. g., the flow of control or the flow of data, whereas sub-workflows can be used as predefined building blocks within this structural description. Most macros just model exactly one aspect of a workflow definition, for example the flow of control, the flow of data or organizational matters. This is supported by the observation that these aspects may often be modelled more or less independently from each other [6]. There are only a few exceptions: the crossing points between the different flows, e. g., an application or sub-workflow call, must be modelled by a special type of macro. As this macro combines different kinds of aspects, it is called **hybrid macro**.

Figure 2b shows the resulting macro body for the conditional branch IF as used in our example. The variable parts are provided as build-time parameters (namely predicate P and two macros $M_1$ and $M_2$ as “plug-ins”). Binding these parameters to actual values leads to “executable” macros. These macros may be combined and thus be used to model the flow of control. Figure 2c shows both instantiated IF-macros. Hence, we have replaced (a part of) the workflow description by an equivalent macro chain.

Fortunately, many design elements reappear again and again, so providing these elements allows the composition of most of the desired workflows. In addition, more complex macros composed of existing macros may be saved in a macro database for future use. Only if a new macro is needed, the IT specialist has to be involved.

All macros and macro bodies defined are collected in libraries. This makes them easy to use, edit, or reuse for the definition of new macros. Fully instantiated macros may be executed within a special engine.

Reusability is one of the key qualities macros offer. As stated before, macros may be built hierarchically using already defined macros. We distinguish between basic and decomposable macros. Basic macros have a direct counterpart at the implementation layer, decomposable macros are constructed using basic and/or decomposable macros.

- **Macro bodies** are workflow modules which can be instantiated as macros by passing parameters.
- **Macros** are instantiated macro bodies. They are simple, executable workflows.
- **Atomic macros** are macros not using other macros as plugin macros.
- **Composite macros** are macros using other macros as plugin macros.
- **Plugin macros** are macros used as sub-macros by a composite macro.
- **Basic macros** are those macros which can be used to define decomposable macros. Every basic macro owns an implementation or implementation description.
- **Decomposable macros** are composed of basic and/or other decomposable macros

Definition 1 Macro Types

Figure 3 shows the relationship between different macro types. There are two similar composition principles at the design level and at the implementation level (see Figure 1). A stock of simple modules is used to compose more complex ones. At the design level, new composite macros may be produced by combining **composite** and **atomic macros**. Composite macros are the nodes of the originating building tree, atomic macros are the leaves. Macros may be nested arbitrarily many times, but finally there must be an atomic macro to complete the definition (Figure 3a).

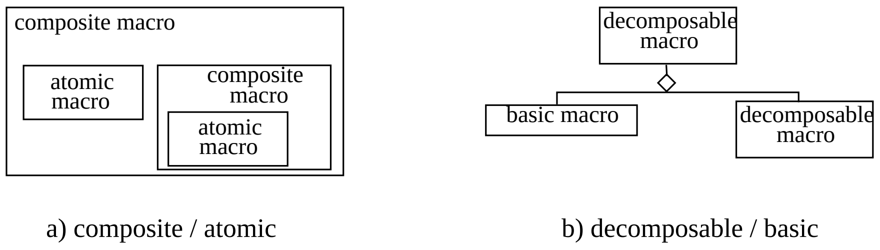


Figure 3 Relationships between different macro types

At the implementation level, macros are built up hierachically, too (Figure 3b). **Decomposable macros** may be transformed into a representation using only **basic macros** (this work is done by a parser, see Section 4). Fortunately, a few basic macros suffice to achieve this goal. For example, most common constructs can be built by using only a few elements like conditional branch, iteration, sequence, parallelization and serialization. Basic macros can be directly mapped into the implementation layer, for they own a direct counterpart (this is done by a compiler). If the implementation layer changes, only the implementation of the basic macros must be adapted.

In most cases, building up new macros is quite simple. In Figure 4, the definition of a new macro body WHILE is shown using the macros IF, REPEAT, and DoNothing defined in an earlier step. The parameters of WHILE (P, M) are mapped into the parameters needed by IF (P) and REPEAT (M, ~P). DoNothing and REPEAT are used as plugin macros for IF. More complex is the handling of macros accepting an arbitrary number of parameters, this will be discussed later.

**2.1 Declaration and Usage of Macros**

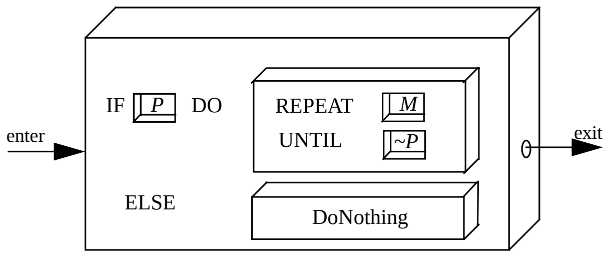


Figure 4 Building WHILE using IF and REPEAT

We choose XML for the representation of our workflow definition language (WDL) as well as for the macros themselves. Thereby we do not only achieve the structure and readability of the definition, but also the advantage of standardized tools for parsing, analysing, operating, and transforming of the definition. Therefore, the macros can be integrated seamlessly into the workflow definition language, so that every new macro improves the readability, if not also the expressiveness, of the WDL.

Before a macro can be used, it must be registered in the macro component. This involves three steps. Firstly, the macro's composition rules for the workflow definition language, i. e., the rules where in the workflow definition language the macro use is allowed and where it is forbidden. For example, it would make no sense to use a data-flow macro in the description part of the control flow.

The composition rules are accomplished by the notion of an interface, which is very similar to the interface construct offered by the programming language JAVA$^{TM}$ [4]. For example, many macros just let pass the flow of control through themselves, despite of the inner working. Therefore, there is one special point to enter and another one to leave the macro. A suitable interface for this kind of macro may be called "WDL_INTERFACE:LINEAR". Examples satisfying this interface are "IF" and "LOOP" which both have different inner workings, but just pass the flow of control through themselves.

Obviously, different macros may be derived from the same interface, as different macros comply with the same composition rules accomplished with an interface.

## 3. Macro Representation

After the macro's interface is defined, we must declare its representation in the WDL. This describes the internal structure of the macro itself. As we have already mentioned, the WDL is XML-based, so a workflow definition itself is an XML document. Hence, the natural description of a macro declaration/body is a DTD fragment which just describes the syntactical structure the macro supplements to the WDL. Moreover, the DTD fragment also uses the interface idea in order to describe what macros are allowed to be plugged into the macro body.

```
<!ELEMENT IF (CONDITION, ACTION, ELSE?)>
<!ELEMENT CONDITION (WDL:EXPRESSION)>
<!ELEMENT ACTION (WDL_INTERFACE:LINEAR)>
<!ELEMENT ELSE (WDL_INTERFACE:LINEAR)>
```

Example 1 Macro representation of the "IF" macro as DTD fragment

The definition of IF is shown in Example 1, the usage in the workflow definition is shown in Example 2, where "SEARCH" and "NEW" are macros, which comply to the "WDL_INTERFACE:LINEAR" interface, too.

Remember, the collection of all defined macros and interfaces makes up the complete WDL which may be used to define workflows. Therefore, all interfaces and macros can be used to define new ones and thus extend the WDL. As stated before, the definition is done by describing the new macro using XML. As precondition of the translation process, the new document must be well-formed and valid w.r.t. the DTD containing all DTD fragments from the macros already defined. This document may be parsed to transform it in a basic form, where only basic macros are used (Section 4). Therefore, after this translation process the resulting document must be valid for the DTD formed from the description of the basic macros only.

```
<IF>
   <CONDITION>
      <WDL:EXPRESSION>"CUR > 0" </WDL:EXPRESSION>
   </CONDITION>
   <ACTION>
      <SEARCH>
         <CURRENT>      CUR </CURRENT>
         <CUSTOMER>     CNR </CUSTOMER>
      </SEARCH>
   </ACTION>
   <ELSE>
      <NEW> <CUSTOMER> CNR </CUSTOMER> </NEW>
   </ELSE>
</IF>
```

Example 2 Using the definition language

## 4. Macro Implementation

The third and last step involved in the definition of the macro is its implementation. The macro implementation deals with the processing of the macro within the macro component, as outlined by the algorithm shown in Figure 5.

```
MacroProcessor:
   oldDocument, newDocument: workflow_definition_document;
   newDocument := oldDocument;
   // expand all composite macros
   REPEAT
      FOR composite_macro IN MacroComponent.getCompositeMacros() DO
         newDocument:= composite_macro.process (newDocument);
      END;
   UNTIL (newDocument = oldDocument);

   // translate to the target machine using all basic macros
   Translate (newDocument);
END.
```

Figure 5 Algorithm for macro transformation within the macro processor

```
<!ELEMENT SWITCH (CASE+)>
<!ELEMENT CASE (CONDITION, ACTION)>
```

Example 3 Macro representation of the “SWITCH” macro as DTD fragment

The whole process is specified as a document transformation from a workflow definition document to another workflow definition document which at least has some parts of the originating document’s macro usage replaced by other (simpler) composite macros or basic macros. When there are no more replacements possible, the expanded workflow definition must only use basic macros. Afterwards, the workflow definition is ready to be translated to the target workflow engine.

The macro transformation itself is done by the means of expansion processors which are configured to perform the transformation a macro defines.

As a generic expansion processor, the macro component integrates an XSLT processor [13] like, e. g., SAXON ([15]) or Xalan-Java([14]). The generic expansion processor takes XSLT as its configuration language and transforms (stepwise) the workflow definition by applying the macro’s associated XSLT stylesheet. As XSLT is a widely adapted technology, powerful tools exist which help in the design of such a transformation stylesheet interactively. Hence, the generic expansion processor is a good starting point for the development of macros.

Let us look at an example. A very useful macro is SWITCH, a 1-out-of-n selection. Suppose, only IF macros are available (just like in ancient programming languages), then we may build up SWITCH as illustrated in Figure 6. The definition of the SWITCH macro is shown in Example 3. Note that as the SWITCH is built upon the IF macro, it “reuses” its

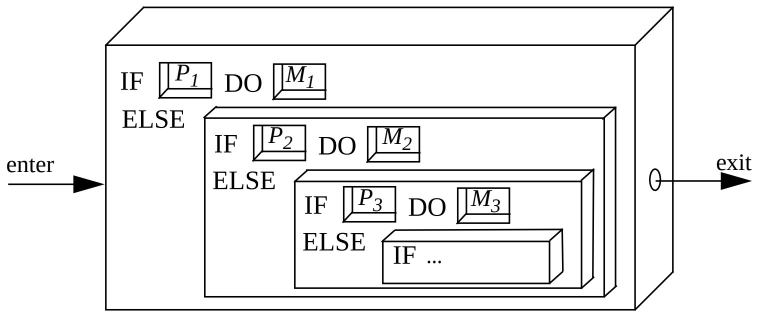


Figure 6 Building macro SWITCH using IF

definition for CONDITION and ACTION. The transformation of the SWITCH macro is done by the generic expansion processor, the associated XSLT stylesheet looks like the one outlined in Example 4.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"versi-
on="1.0">
   <xsl:template match="SWITCH">
      <xsl:choose>
         <xsl:when test="count(./CASE) >= 2">
            <IF>
               <xsl:apply-templates select="./CASE[1]"/>
               <ELSE>
                  <SWITCH>
                     <xsl:for-each select="./CASE">
                        <xsl:if test="position() > 1">
                           <xsl:element name="CASE">
                              <xsl:apply-templates/>
                           </xsl:element>
                        </xsl:if>
                     </xsl:for-each>
                  </SWITCH>
               </ELSE>
            </IF>
         </xsl:when>
         <xsl:when test="count(./CASE) = 1">
            <IF>
               <xsl:apply-templates select="./CASE[1]"/>
            </IF>
         </xsl:when>
      </xsl:choose>
   </xsl:template>
</xsl:stylesheet>
```

Example 4 Body definition of macro SWITCH as an XSLT stylesheet

If we use the SWITCH macro in the workflow-definition document like in Figure 7, the generic expansion processor transforms the document stepwise using the XSLT stylesheet introduced in Example 4. The first CASE is taken to the IF macro, whereas the rest goes

```
<SWITCH>
   <CASE>
      <CONDITION> pred0 </CONDITION>
      <ACTION> act0 </ACTION>
   </CASE>
   <CASE>
      <CONDITION> pred1 </CONDITION>
      <ACTION> act1 </ACTION>
   </CASE>
   <CASE>
      <CONDITION> pred2 </CONDITION>
      <ACTION> act2 </ACTION>
   </CASE>
</SWITCH>
```

Figure 7 A simple example using SWITCH

to the ELSE part and forms a new SWITCH macro with one CASE less. After the first transformation step, a definition emerges as the one shown in Figure 8.

As already being basic, the macro IF is ignored in the expansion step. Therefore, the SWITCH macro is the only candidate for expansion in the next step. After this, the CASEs of the SWITCH macro are all substituted by IFs. The expansion step ends here, because every macro used is a basic one. At this point, the macro component starts generating code for all basic macros, which is outlined in Section 5.

```
<IF>
   <CONDITION> pred0 </CONDITION>
   <ACTION> act0 </ACTION>
   <ELSE>
      <SWITCH>
         <CASE>
            <CONDITION> pred1 </CONDITION>
            <ACTION> act1 </ACTION>
         </CASE>
         <CASE>
            <CONDITION> pred2 </CONDITION>
            <ACTION> act2 </ACTION>
         </CASE>
      </SWITCH>
   </ELSE>
</IF>
```

Figure 8 First expansion step

Despite the power of XSLT, sometimes it is not easy—albeit the powerful tools which exist—to create an XSLT stylesheet that does exactly the transformation needed. In this situation, the processing is defined as an external processor which performs all the processing of a new macro. Consider for example the macro “SERIAL”, which executes a set of the plug-in macros (which must belong to the interface WFL_INTERFACE:LIN-

```
<IF>
   <CONDITION> pred0 </CONDITION>
   <ACTION> act0 </ACTION>
   <ELSE>
      <IF>
         <CONDITION> pred1 </CONDITION>
         <ACTION> act1 </ACTION>
         <ELSE>
            <IF>
               <CONDITION> pred2 </CONDITION>
               <ACTION> act2 </ACTION>
            </IF>
         </ELSE>
      </IF>
   </ELSE>
</IF>
```

Figure 9 Final result

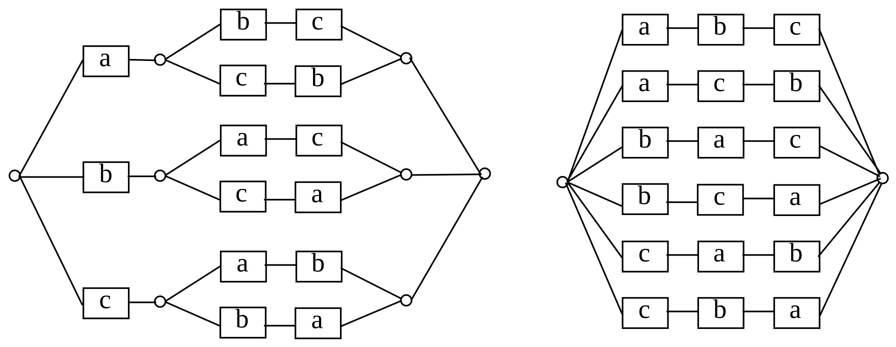
EAR) in an arbitrary order. The macro uses two already defined macros “SEQUENCE”, which executes in a fixed order, and “ALTERNATIVE”, which executes only one of the paths given. If we want to express “SERIAL(a,b)”, the expansion rule is described by:

SERIAL (a,b) => ALTERNATIVE ( SEQUENCE(a,b), SEQUENCE(b,a) ).

But how should we express “SERIAL(a,b,c)”? Possible solutions are shown in Figure 10. One way is using recursive substitution, leading to a deeply nested result (Figure 10a). The other way shown in Figure 10b seems to be more preferable. Unfortunately, the stylesheet for this transformation is very complicated and error-prone to design. It is more feasible to use some kind of external processor to perform the transformation. As a model for the external processor we choose the transformation of a DOM [12], as it can be seen as the abstract syntax tree (AST) of the input document, and the processor can do all transformations needed on this AST according to the macro’s semantics.

## 5. Code Generation

In order to actually execute a macro it has to be translated into a representation understood by the implementation layer which, in general, is some WfMS with an associated workflow engine. Therefore, we need an appropriate compiler to handle this translation. As we have seen before, a few basic macros suffice to build up all the decomposable ones. Only these basic macros have to be processed by the compiler. For this reason, we have to expand all decomposable macros as described in Section 4 using the macro processor. Doing so, only basic macros remain. Errors occurring are reported to the user. If the macro processor succeeds, the resulting document is valid w.r.t. the DTD including only the basic macros’ definitions. Thus, it may be transferred to the compiler. The compiler translates this definition into an implementation-layer-dependent representation. Every basic macro is associated with a matching translation rule mapping it into the WDL used by the WfMS. Again, errors are reported. If the compiler succeeds, a valid “executable” is delivered as result. This executable is stored within an “executable server” providing executables for direct invocation.



a) recursive expansion

b) algorithmical expansion

Figure 10 Two different solutions for expanding SERIAL(a,b,c)

As stated before, the compiler depends on a chosen implementation layer. Therefore, we do not describe any implementation details in this paper. If the implementation layer is changed, only the compiler has to be changed, too. Fortunately, a new compiler only has to translate basic macros, and there really are only a few of them. Parser and macro definitions may remain unchanged.

## 6. Related Work

In this section, we outline two projects having similar ideas. First, we have a closer look at the *workflow patterns* described by W. van der Aalst et al. [1]. Subsequently, we present some ideas considering the representation of flow of control introduced by Markus Böhm in [2].

### 6.1 Workflow Patterns

Although there are many WfMSs with more or less expressive power, there are nevertheless certain requirements recurring quite frequently during the analysis phases of workflow projects. These requirements can be described in an abstract form called *workflow pattern* (similarly to [5]). In [1], the following collection of patterns is presented:

- basic control patterns (Sequence, Parallel Split, Simple Merge, ...),
- advanced branching and synchronization patterns (Multi-choice, Synchronizing Merge, ...),
- structural patterns (Arbitrary Cycles, Implicit Termination),
- patterns involving multiple instances,
- temporal relations,
- state-based patterns,
- inter-workflow synchronization.

Our macro approach is quite compatible with this idea of workflow patterns. Likewise, macros (without their implementation) are an abstract description of workflow parts, too. Many of the patterns described have a direct macro counterpart in our system.

There are two main differences between the workflow patterns and our macro approach. The first one is that we do not only provide macros for control flow, but also for data flow and other tasks using our extensibility mechanism. The second difference is that we provide some of the patterns only in valid combinations (XOR-Split combined with XOR-Merge), so that the workflow designer cannot model semantically incorrect combinations (XOR-Split with AND-Merge). As a consequence, one major source for specification errors is eliminated in a satisfactory manner.

Nevertheless, the macro approach may be seen as (partial) realization of patterns, so they may be easily used by workflow developers.

### 6.2 Flow of Control Within Workflow Types

According to [2] there are three different classes of flow-of-control definitions. Böhm distinguishes between *primitives*, *constructs*, and *execution directives*.

Primitives are used to correlate the states of two workflows (e.g., the rule “when the first workflow is done, the second one becomes ready”)

Constructs are combinations of primitives to gain a certain functionality (e. g., SPLIT, ALT or PAR). While using different constructs, the designer has to pay attention to certain rules. Obviously, there is no sense at all in combining an XOR-SPLIT with an AND-JOIN, to give a simple example.

Last but not least, the execution directives describe the order the sub-workflows are executed in. The actual sub-workflows are hidden and new connection points are offered. Therefore, the designer has no more the opportunity to create “senseless” connections. This idea of encapsulating sub-workflows is also closely related to our way using macros. Like Böhm we offer only the interface, hiding away the innards.

Our macro approach can be classified in this context as a combination of Böhm’s constructs and execution primitives, but enriched by algorithmic elements within the macros, which allow for an arbitrary number of plugin macros. Moreover, we do not only concentrate on flow-of-control definitions, but also on data flow and other tasks.

## 7. Conclusion & Future Work

WfMSs are more and more accepted, since they provide an efficient way to control and monitor complex processes. But unfortunately there are many different software products called WfMS and no common workflow model exists. These systems are very powerful for tasks they were built for, but they are not applicable universally. The Workflow Management Coalition (WfMC) tried to figure out the commonalities of different WfMSs and to define at least some standards.

Nevertheless, most systems may only be used by specialists, are difficult to handle, and do not offer the flexibility and extensibility desired by process designers. We have primarily designed Arktis to improve this situation. The component described in this paper provides high-level constructs called “macros” for defining flow of control and flow of data. Only few macros are offered directly, but they may be used to define new, more complex ones and thereby extend the possibilities for defining workflows.

We then had a closer look at the macro component and showed how complex macros may be derived from fundamental macros. This concept enables the designer to build up his own stock of macros, custom tailored exactly for him and his needs. The more macros he builds the better complex constructs may be expressed in a simple way. Hence, we gain adaptability and increasing expressive power.

Furthermore, we examined the possibility to define macros with an arbitrary number of parameters. Additionally, we showed how a macro definition is expanded, until it only consists of basic macros and hence may be translated to the WDL offered by the underlying WfMS through the compiler. Therefore, if the WfMS is exchanged, only a new compiler has to be provided, the macro definitions remain the same.

As we have seen, our Arktis approach promises to relieve non-IT-people while modelling real life processes as workflows. Therefore, they can use a WDL adapted to their special needs, powerful enough to express complex facts in an easy way. As the actual WfMS is hidden by the macro component, the workflow designer has not to care about technical details. Even if this WfMS is exchanged, the designer does not notice at all. On the other hand, the IT-specialists are used for the tasks they are qualified for: to handle the technical aspects (like maintaining the macro compiler) of the workflow system.

Since macros describe *either* flow of control *or* flow of data, we have to provide a way to combine flows at the points where applications are started. This will be accomplished by the integration of “hybrid macros”. We also plan to test our model by using an existing WfMS as implementation layer (e. g., MQSeries/Workflow).

## References

[1] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, A.P. Barros: Workflow-Patterns, BETA Working Paper Series, WP 47, Eindhoven University of Technology, Eindhoven, 2000.

[2] M. Böhm: Entwicklung von Workflow-Typen, Springer Verlag Berlin, Heidelberg, New York, 2000

[3] M. Bon: ARKTIS/Makros – Eine Makrokomponente als Basis für flexible, erweiterbare WfMS. Diplomarbeit, Universität Kaiserslautern, Fachbereich Informatik, Kaiserslautern, Germany, 1999

[4] J. Gosling, B. Joy, G. Steele: The Java$^{TM}$ Language Specification, Addison-Wesley, 1996

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: Design Patterns: Elements Of Reusable Object-Oriented Software, Addison-Wesley, Reading, Massachusetts, 1995

[6] S. Jablonski: Workflow-Management-Systeme – Modellierung und Architektur, Thomsons Aktuelle Tutorien, Int. Thomson Publ., Bonn, Germany, 1995

[7] S. Jablonski, C. Bussler: Workflow Management: Modeling Concepts, Architecture, and Implementation, International Thomson Computer Press, Bonn, Germany, 1996

[8] IBM: MQSeries Workflow (distributed platforms) Version 3.2.2, http://www-4.ibm.com/software/ts/mqseries/library/manuals/#Workflow

[9] Software-Ley: COSA User Manual, Software-Ley GmbH, Pullheim, Germany, 1996

[10] Staffware: Staffware 97 / GWD User Manual, Staffware plc, Berkshire, United Kingdom, 1997

[11] Workflow Management Coalition, The Workflow Reference Model, Document Number WfMC TC00-1003, Jan. 1995, http://www.aiim.org/wfmc/mainframe.htm

[12] W3C, Document Object Model (DOM) Level 3 Core Specification, W3C Working Draft 1, 3 September 2001, http://www.w3.org/TR/2001/WD-DOM-Level-3-Core-20010913/

[13] W3C, Extensible Stylesheet Language (XSL) Version 1.0, W3C Recommendation 15 October 2001, http://www.w3.org/TR/2001/REC-xsl-20011015/

[14] The Apache XML project, Xalan-Java, Version 2, http://xml.apache.org/xalan-j/index.html

[15] M. H. Kay: SAXON The XSLT Processor, Version 6.4 http://saxon.sourceforge.net/