An Extensible Workflow Designing 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 our system Arktis 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

An Extensible Workflow Designing 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 our system Arktis 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.

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. As more and more workflow management systems (WfMS) are available ([MQS], [JB96], [SL96], [STA97]), 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 [WfMC95]. 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 (or processes), a runtime environment, and a user-interaction abstraction. The implementation of all these components lies in the hands of the WfMS vendor, only the interaction interfaces between the components are defined by the standards.

To get a new workflow schema, i. e. the specification of a workflow (instance) executed, into a

WfMS, it has to be described in a system-understandable way.

Since designing a complex process, what workflows are without any doubt, is a complex task, the process designer is often supported by a set of tools assisting him in the process definition by means of script languages or equally powerful graphical representations. The workflow's control flow which defines the logical order of application invocations, is often described by the same means as used in structured programming languages, namely by a fixed set of control structures like, for example, IF and LOOP along with workflow-internal control variables. The same approaches are taken for the functional decomposition of a workflow. The concept of nested or sub-workflows is used, in analogy to procedure calls, with the restriction that only complete, previously defined workflow schemas can be used in the specification of a new workflow schema.

All those tools — albeit their easy to use GUI or language approach — are meant to be applied by IT specialists in order to specify the complete workflow schema before the first workflow instance (short: workflow) is invoked. Moreover, it is often difficult to express the ideas of a process by means of the target workflow model, since most (if not all!) features are fix, even for IT specialists, not to think of non-IT people.

For business processes, which should be implemented as workflows, the non-IT people ("engineers") are the ones which have the complete understanding of the processes, not the IT specialists who know the system. Therefore, it is desirable that the engineers can express their ideas of a workflow using an appropriate abstraction from its actual implementation.

The Arktis (our own prototypical WfMS) approach (see Figure 1) is to provide an extensible workflow model consisting of building blocks ("macros") for workflows and methods to combine them. The macros and the combination methods are described in an abstract manner, only their functional interface is known, implementation details are hidden to the engineers who specify the workflows. The implementation of the macros and their combination methods lies in the responsibility of the IT specialist.

As different engineers frequently identify different building blocks or combination methods as essential for the way they describe their workflows, extensibility is extremly important for this approach to be accepted. Therefore, the building blocks are not hardcoded in the Arktis ap-

proach, but are provided by a macro component. Hence, they are configurable and extensible any time, which leads to a high degree of adaptability for easy deployment and refinement of workflows in different domains.

Given the macros, everything (simple or complex) needed by an engineer (even IT specialist) can be pre-build by a few IT specialists. In analogy to the previously mentioned programming

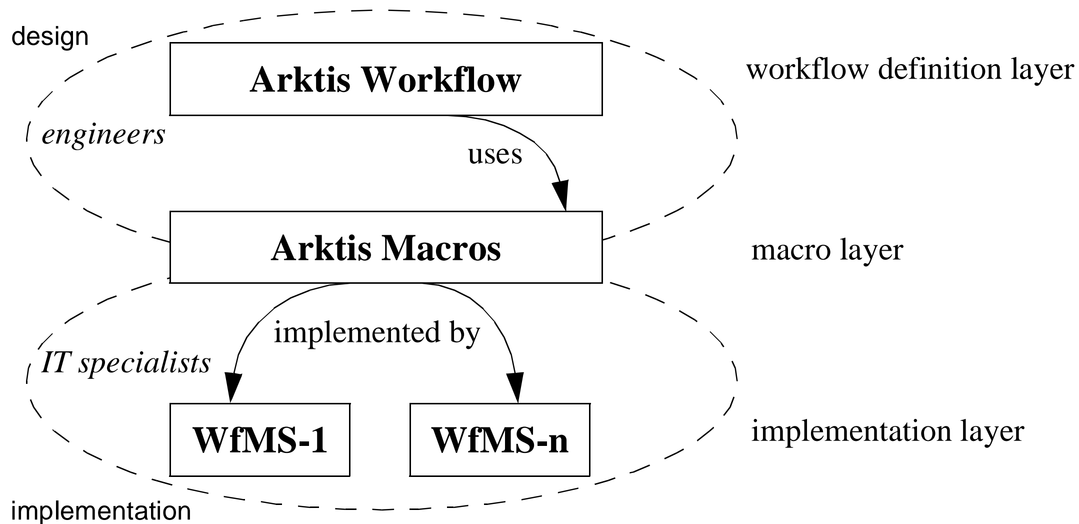


Figure 1 The multiple layers of Arktis

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 occuring) special problems which cannot be solved with the existing macros.

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. We refer to some related work, before we give a conclusion and an outlook.

2. The 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.

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 provide it.

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 specify 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 very fundamental point: macros only cover one aspect at a time, therefore they may describe either the flow of control or the flow of data, not both. This proceeding is supported by the observation in [Ja95] that these aspects are often modelled more or less independant from each other . There are only a few exceptions: the crossing points between the different flows, e. g. an application, must be modelled by a special type of macro. As this macro is involved in the flow of control as well as in the flow of data, it is called **hybrid macro** (for more information see [Bo99]).

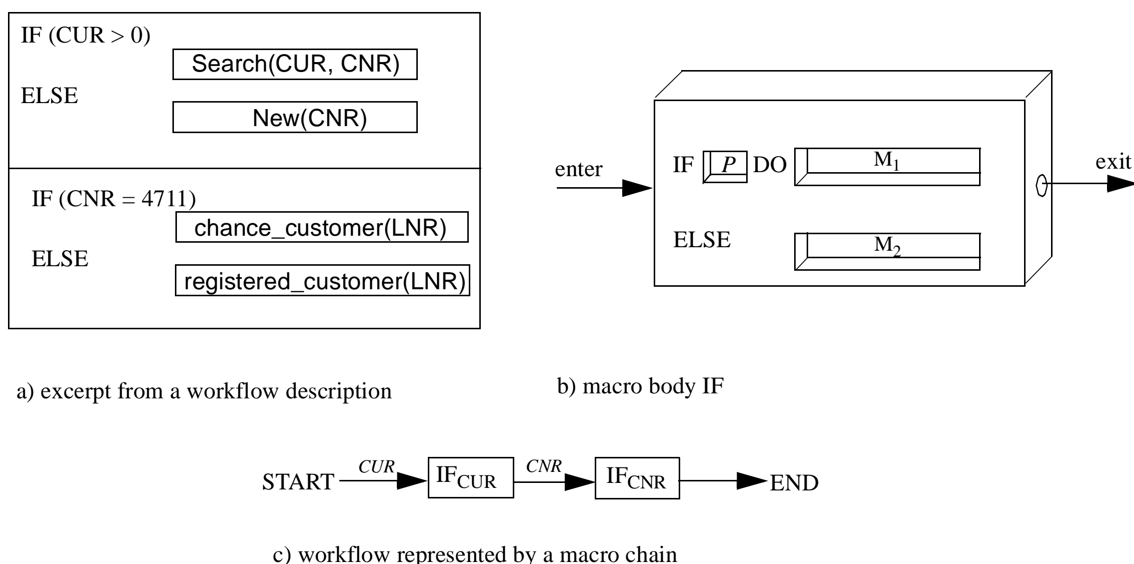


Figure 2 Transforming a workflow description into a macro chain

Figure 2b shows the resulting macro body for the conditional branch IF as used in our example. The variable parts are provided as buildtime 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 the both instantiated IF-macros. Hence, we have replaced (a part of) the workflow description by an equivalent macro chain.

To accomplish this transformation we need a kind of programming language to describe the use of macros. Such a language is very simple, and a suitable GUI to support the designer should not be too difficult to be developed. In the most common case a macro is selected, supplied with parameter values and called for execution. For this purpose, we use the keywords #MACRO_USE and #EXEC. The ‘#’ indicates a reserved word, so a macro name must not begin with ‘#’. But what is the difference between #EXEC and #MACRO_USE? It is quite simple. If we want to expand the actual code (as done with macros by the C-preprocessor), we will choose MACRO_USE. But if we have already got an executable and just want to call it like a subroutine, we will use #EXEC. No (textual) expansion is done, the macro called is used as a “black box”. The definition of IF_{CUR} is shown in Example 1, where “Search” and “New” are executable macros. A mechanism for determining the valid parameter types will be introduced in Section 2.1. In Section 3.1 the processing of #MACRO_USE will be explained in detail.

```
#MACRO_USE  (IF, “CUR > 0”,
                 #EXEC Search (CUR, CNR),
                 #EXEC New(CNR)
            );
```

Example 1   Using the definition language

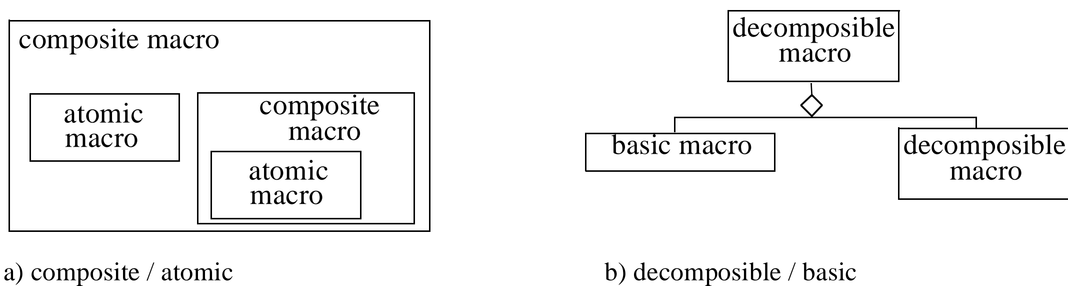
All macros and macro bodies defined are collected in libraries. This makes them easy to use, edit, or reuse for the definition of new macros. Fully instantiated macros may be executed within a special engine.

Reusability is one of the key qualities macros offer. As stated before, macros may be built hierachically using already defined macros. We distinguish between basic and decomposible mac-

ros. Basic macros have a direct counterpart at the implementation layer, decomposible macros are constructed using basic and/or decomposible 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 decomposible macros. Every fundamental macro owns an implementation or implementation description.
- **Decomposible macros** are composed of fundamental and/or other decomposible macros.

Definition 1 different macro concepts



a) composite / atomic b) decomposible / basic

Figure 3 Relationships between different macro-types

Figure 3 shows the relationship between different macro types. We have 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 a good many times, but finally there must be an atomic macro to complete the definition (Figure 3a).

At the implementation level, macros are built up hierachically, too (Figure 3b). **Decomposible**

macros may be transformed into a representation using only **basic macros** (this work is done by a parser, see Section 3.1). 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 Passing Parameters

To declare and to instantiate macros requires answering the following questions: What do the connection points between macros look like? How many entry and exit points (we call them "transitions") exist? Which plugin macros may be used? What are the data types of the parameters passed?

All this information is kept in a special declaration which – according to the construct offered by the programming language JAVA[TM] [GJS96] – is called an "Interface". Thus, we can separate the interface definition from the specification of a macro's behaviour (which is is done in the macro body). This makes sense, because different macros belong to a similiar class and may use an identical interface. Every macro body is dedicated to an interface; during the definition of the macro body the elements declared in the interface may be used. Example 2 shows an interface describing macros doing some kind of iteration. It has one input and one output transition, both are used in the flow of control (CF) without restrictions for the data accepted. Furthermore, the interface has a function returning a boolean value used as iteration predicate. The last parameter is the macro forming the iteration body. The macro bodies REPEAT and WHILE shown in Figure 4, both match with this interface .

```
INTERFACE NEW   IterationInterface;
IN              ENTER [CF, DOMAIN *];
OUT             EXIT  [CF, DOMAIN *];
FUNCTION        p OF BOOLEAN;
MACRO           m IS MacroInterface;
END;
```

Example 2 declaration of an interface used for iteration macros (e. g., WHILE, REPEAT)

The elements of an interface definition are:

- input transitions
- output transitions
- types of plugin macros
- declaration of simple-typed parameters
- declaration of collections

The first four items are self-explanatory, so we only have to discuss **collections**. A collection is used to handle an arbitrary number of parameters. For example, the macro PAR gets macros which shall be executed simultaneously. Only the macro type (its interface) must be known in advance, the number of macros may remain unknown until the actual instantiation.

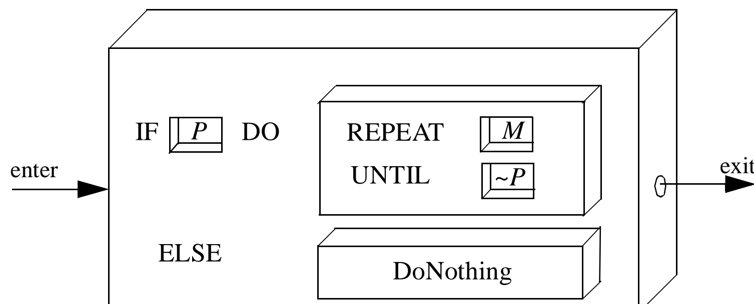


Figure 4 Building WHILE with IF and REPEAT

Different macros may be derived from the same interface. The macros PAR (for executing macros in parallel) and SER (for serializing macros) have identical interface declarations. Therefore, both will be suitable for using them as parameter, if the corresponding interface type is requested by the calling macro.

2.2 Handling an Arbitrary Number of Parameters

There are two techniques for expanding macros: **recursion** and **expansion**. Recursion means that in every step the first element of the collection is processed, and the macro itself is used as plugin taking the rest. This leads to deeply nested macros. In contrast, if we use expansion, one separate branch is created for every element.

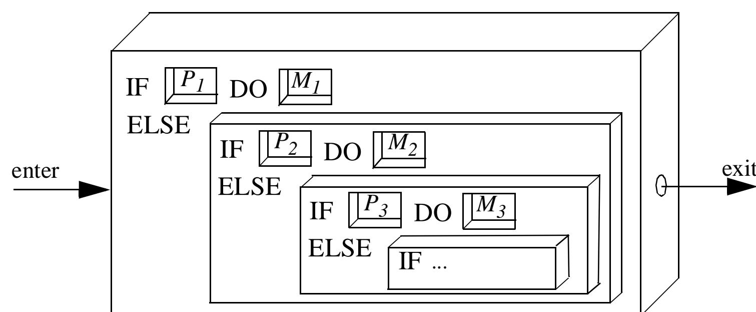


Figure 5 Building macro SWITCH using IF

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 shown in Figure 5. The associated “code” would look like illustrated in Example 3.

Another macro used quite often is PAR providing the execution of several macros in parallel. A first idea for the implementation of PAR could be to use recursion. The fundamental macro (stopping the recursion) is PAR_2 as shown in Figure 6a. But how should the recursion be defined? There are several solutions, each one suitable and completely correct, but not very elegant as shown in Figure 6b and Figure 6c. Using n macros leads to 2(n-1) connections which may be spread asymmetrically.

```
MACRO    SWITCH IS DynamicParmInterface;
BODY   #IF (  #IS_EMPTY (collectionName) = #FALSE)
                 #MACRO_USE IF(#FIRST(collectionName).p,
                                         #FIRST(collectionName).m,
                                         #MACRO_USE SWITCH(#REST(collectionName))
           #ELSE #EXEC DEFAULT();
           #END;
END.
```

Example 3 Body definition of macro SWITCH

Much more desirable is the solution shown at the bottom of Figure 6. Executing n macros is modelled by n separate branches. This can be achieved by using the expansion technique. The resulting macro will then be generated dynamically. It is also possible to guard the different branches, to synchronize or to exclude them from each other. The originating description is quite simple:

```
MACRO PAR IS DynamicParmInterface;
BODY:  #FOREACH (collectionName)
              #ON p #EXEC m #BLOCKED #ONCE ONLY;
           #END;
END.
```

Example 4 Body definition of macro SWITCH

So you may ask, why do we need recursion at all? This is because sometimes order does matter! Using algorithmical expansion leads to several branches not distinguishable anymore.
Especially we lose information about the order the parameters have been passed. However, SWITCH needs this information.The predicates guarding the different branches of SWITCH do not have to be disjunctive (e. g., $x > 2$, $x > 3$, $x > 4$, ...), so using expansion may lead to undesirable results. Only using recursion like we did before creates a macro providing the desired behaviour.

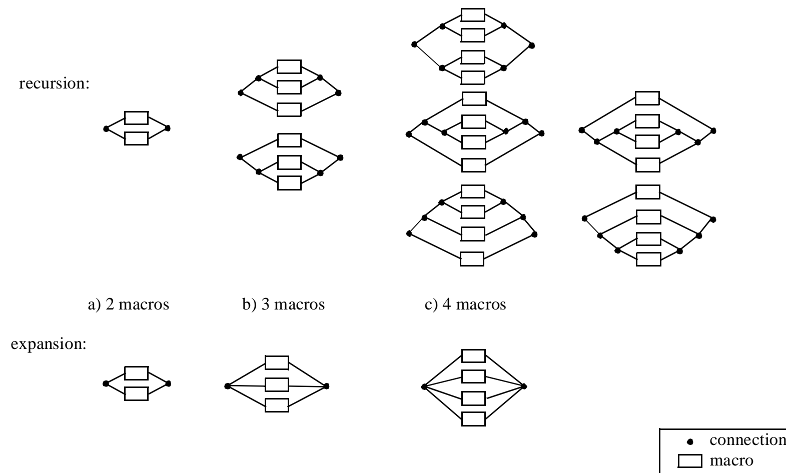


Figure 6 Resolving an arbitrary number of parameters

2.3 Refining Data Flow Macros

Another problem occurs when flow-of-data macros are used. Figure 7, for example, shows a macro used to form a new document ("C") out of two smaller ones ("A" and "B"). The processing of data is always the same, but unfortunately the data itself is not! As soon as the document's layout is specified, the macro can only consume documents having the same type as "A" or "B". All documents produced will have the same type as "C". The solution is named "refinement": A kind of prototypical macro (in OO-terminology it would be called an "abstract macro") is defined that only specifies the number and names of expected documents and the behaviour of the macro, while omitting the types of the documents processed. A new macro may now be easily derived from the prototypical macro by adding an appropriate interface definition.

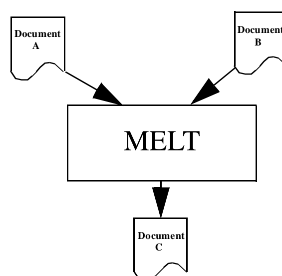


Figure 7 A macro melting two documents into a new one

How can we express refinement using our definition language. Just look at Example 5. The macro “protoMelt” is defined having two input transitions “in1” and “in2”, both accepting any kind of data (“DOMAIN *”), and one output transition “out”. DocumentC_melt refines the proto-macro and substitutes the transitions by adding appropriate DOMAIN definitions.

```
MACRO   protoMelt IS NEW protoMeltInterface;
IN:     in1     [DF, DOMAIN *],
        in2     [DF, DOMAIN *];

OUT:    out     [DF, DOMAIN *];
END.


MACRO DocumentC_melt REFINES protoMelt;
IN:     in1     [DF, DOMAIN DocumentA],
        in2     [DF, DOMAIN DocumentB];
OUT:    out     [DF, DOMAIN DocumentC];
END.
```

Example 5 Refining a macro melting documents

3. Macro Translation

In order to actually execute a macro it has to be translated into a form understood by the implementation layer which, for example, may be another 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 also need a parser to preprocess the macro definition. Figure 8 shows the process of translating a macro definition into an executable. After defining the macro (1) the definition is passed to the parser. The parser expands

all complex macros using their definition from the macro library (2), until only fundamental macros remain. Errors occuring are reported to the user. If the parser succeeds, the resulting definition is syntactically correct and fully expanded. It may be transferred to the compiler. The compiler translates this definition into an implementation-layer-dependent form. The translation rules are taken from the implementation library (3). Every fundamental macro is associated with a matching translation rule managed by this library. Again, errors are reported. If the compiler succeeds, a valid executable has been produced. This executable is stored within an "executable server" (4) providing executables for direct invocation. It may also be replicated in the implementation library in order to shorten future translations.

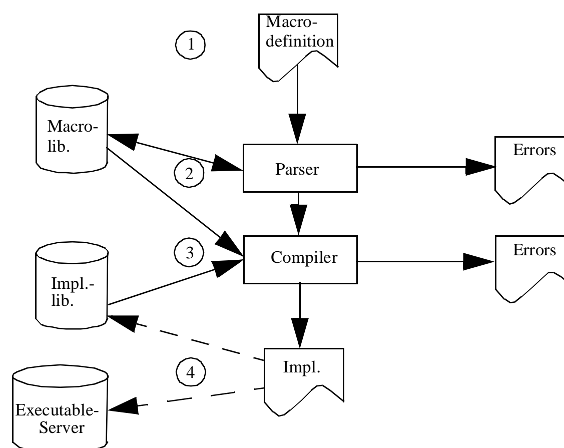


Figure 8 Translating a macro definition into an executable

3.1 Expanding the Macro Definition

A crucial parser task is the expansion of a macro definition. We will illustrate how this is accomplished by examining one step in the transformation of SWITCH. Assuming we use again the collection "collectionName" with its elements (p_i, m_i). If the definition processed contains a line like the one shown in Figure 9, it will be processed according to the body definition of macro SWITCH introduced in Example 3. First the #IS_EMPTY-predicate is evaluated. Hence, there are further elements, a new IF clause is generated, using another SWITCH macro as its ELSE branch. After the first transformation step, a definition emerges as the one shown in Figure 10 .

```
...
#MACRO_USE SWITCH (((p1 m1), (p2 m2)));
...
```

Figure 9 Using SWITCH

```
#MACRO_USE IF (p1, m1,
        #MACRO_USE SWITCH (((p2 m2), (p3 m3))))
END.
```

Figure 10 First expansion step

The macro IF is already a basic macro that can be directly processed by the compiler. Therefore, the SWITCH macro is the only candidate for expansion in the next step. Applying the body definition again leads to the definition shown in Figure 11.

```
#MACRO_USE IF (p1, m1,
        #MACRO_USE IF (p2, m2,
                #MACRO_USE SWITCH (())))
END.
```

Figure 11 Second expansion step

Then, the given collection is exhausted. In a final step, the SWITCH macro is substituted by #EXEC DEFAULT, a call to an executable macro. The processing ends and the compiler can start its work of translation.

```
#MACRO_USE IF(p1, m1,
        #MACRO_USE IF(p2, m2,
                #EXEC DEFAULT()))
END.
```

Figure 12 Final result

3.2 Compiler

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.

4. Related Work

In this section, we outline two projects having similiar ideas. First we will have a closer look at the *workflow patterns* described by W. van der Aalst et al. [AHK+00]. Subsequently, we present some ideas considering the representation of flow of control introduced by Markus Böhm in [Bö00].

4.1 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 [GHJV95]). In [AHK+00], the following collection of patterns is presented:

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

Our macro approach is quite compatible with this idea of workflow patterns. 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.

4.2 Flow of Control Within Workflow Types

According to [Bö00] 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., first workflow: done -> second workflow: 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 opportunity anymore 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 arbitrary number of plugin macros. Moreover, we do not only concentrate on flow-of-control definitions, but also on data flow and other tasks.

5. Conclusion & Future Work

WfMS 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 WfMS 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. 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. Especially, we examined the possibility to define macros with an arbitrary number of parameters. Additionally we looked at the parser and how a macro definition is expanded, until it only consists of fundamental macros and hence may be translated by the compiler.

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. We plan to use a construct named “hybrid macro” to accomplish this. We also plan to test our model by using an existing WfMS as implementation layer (e. g. MQSeries/Workflow).

6. Literature

[AHK+00] 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.

[Bö00] Markus Böhm
Entwicklung von Workflow-Typen
Springer Verlag Berlin, Heidelberg, New York, 2000

[Bo99] Markus Bon
ARKTIS/Makros - Eine Makrokomponente als Basis für flexible, erweiterbare WfMS.
Diplomarbeit, Universität Kaiserslautern, Fachbereich Informatik
Kaiserslautern, Germany, 1999

[GJS96] James Gosling, Bill Joy, Guy Steele
The Java™ Language Specification
Addison-Wesley, 1996

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides
Design Patterns: Elements Of Reusable Object-Oriented Software
Addison-Wesley, Reading, Massachusetts, 1995

[Ja95] Stefan Jablonski
Workflow-Management-Systeme – Modellierung und Architektur
Thomsons Aktuelle Tutorien, Int. Thomson Publ.
Bonn, Germany, 1995

[JB96] S. Jablonski, C. Bussler
Workflow Management: Modeling Concepts, Architecture, and Implementation
International Thomson Computer Press
Bonn, Germany, 1996

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

[SL96] Software-Ley
COSA User Manual,
Software-Ley GmbH,
Pullheim, Germany, 1996

[STA97] Staffware
Staffware 97 / GWD User Manual
Staffware plc
Berkshire, United Kingdom, 1997

[WfMC95] Workflow Management Coalition,
The Workflow Reference Model
Document Number WfMC TC00-1003, Jan. 1995
www.aiim.org/wfmc/mainframe.htm