A functional specification of reusing software components

John van den Elst†, Frank van Harmelen †, Guus Schreiber ‡, Monique Thonnat †

† INRIA Sophia Antipolis
BP 93, F-06902 Sophia Antipolis CEDEX, France
Tel.: (+33) 93 65 76 57    Fax: (+33) 93 65 76 43
email: elst@sophia.inria.fr

‡ University of Amsterdam
Faculty of Psychology, Social Science Informatics
Roetersstraat 15, 1018 WB Amsterdam, The Netherlands
Tel.: (+31) 525 6121    Fax: (+31) 525 6896

Abstract
We take the first steps towards a functional specification of reusing predefined software components. We do this by presenting a case study: a description of the knowledge contained in OCAPI, using the KADS expertise model. OCAPI is a system that implements knowledge about how to plan and control the execution of a specific kind of software components, namely image processing programs. The KADS expertise model allows to specify this problem solving behaviour in an implementation independent way, and to give a categorization of the knowledge required to generate this behaviour (i.e., functional specification). Based on this case study we will evaluate the structure and contents of the knowledge, and so identify reusable parts and less developed knowledge structures. This will serve as a basis for abstracting from the OCAPI domain (image processing), to a functional specification of reusing predefined software components in general.

1 Introduction
Software libraries consist of a large number of programs. We use these programs as our predefined software components. Reusing them means that they have to be planned and their execution has to be controlled to reach a certain goal. [1] and [2] categorize the reuse of software components into four steps: abstraction, selection, specialization, and integration. From our point of view, we also look at a supplementary step: execution. Reusing a library of software components first of all requires an abstract definition of the components (abstraction). Then, given a certain problem to solve, a subset of the components is chosen and ordered (selection), initialized (specialization), integrated (integration), and applied (execution) to actually solve the problem. Our aim is to specify this behaviour and give a categorization of the knowledge required to generate this behaviour.

We have taken the first steps towards a functional specification that fulfills this aim. These first steps are taken by using the system OCAPI (Outil de Contrôle Automatique de Procédures Images [3] [4]) as a case study. A knowledge base for OCAPI contains knowledge about a specific kind of software components, namely image processing programs. Information about the goals of the components, the conditions under which they are applicable and the relations between them, are all part of the knowledge base. With this information, OCAPI is able to plan and control the execution of image processing programs to reach a certain goal (e.g., object detection). In this article we present a functional specification of the information contained in the concepts and mechanisms of OCAPI. We do this using the KADS expertise model [5], which models the problem solving behaviour of an agent in terms of knowledge that is being applied in carrying out a certain task (knowledge level description [6]). The functional specification we present, is a semi-formal definition of this knowledge.

Then, based on the functional specification, we are able to evaluate the aptness of the structure of the knowledge, identify missing or irrelevant knowledge, and extend the knowledge description. Furthermore, OCAPI is just a possible symbol level representation of the knowledge needed to perform supervision of image processing programs, and (re)creating its functional specification allows us to analyze the knowledge requirements for the reuse of software components independently from their rather incidental representation in OCAPI. Finally, abstracting from OCAPI’s specific domain (image processing) provides us with the basis for a functional specification of reusing predefined software components in general.

In section 2, we briefly present OCAPI. Then, in section 3, we explain the KADS expertise model. In
section 4 we present the specification of the knowledge contained in OCAPI. Section 5 explains what insights we gained from this work. Finally, section 6 contains conclusions.

2 OCAPI

2.1 Introduction

OCAPI is a tool to semantically integrate image processing programs. A semantical integration means that not only the programs are integrated, but also information about the goals of the programs, the conditions under which they are applicable and the relations between them. This makes OCAPI a tool to perform the supervision of image processing programs.

2.2 OCAPI concepts

In order to model the knowledge involved in semantical integration, the following concepts are used in OCAPI:

Goals. When we process an image we always have in mind an objective or goal to reach. In OCAPI a goal represents an image processing functionality (e.g. segmentation).

Operators. Actions that can be performed are called operators. They contain the information to satisfy a certain goal. The image processing programs are referred to as primitive or elementary operators (e.g. a primitive operator for thresholding). A combination or sequence of operators is expressed in a complex operator. In this case we speak of an operator decomposition. A complex operator for primitive extraction for example, could be a decomposition into subsequently edge detection, segmentation and chaining. Every operator (primitive and complex) also has a set of characteristics, which express the typical properties of the operator, like for example information about the efficiency or speed of an operator.

Arguments. Operators (primitive or complex) and goals are associated with a list of input and output arguments. We distinguish two classes of arguments, data and parameter arguments. Data arguments have fixed values which are given or computed, like e.g. the input image for a certain operator. Parameter arguments (parameters) are tunable arguments, they have adjustable values and are always input arguments, like e.g. the threshold of the thresholding operator hysteresis.

Requests. A request states which goal has to be reached and what is the data of the particular case to work on. In a request we define the initial values of the arguments (e.g. the filenames of the input and output images). Thus, a request is an instantiated goal.

Data descriptions. Information about a data argument is described in the data description. It has two parts, an implementation description and a semantical description. In the implementation description the information about the data argument itself is described (e.g. information about an image, like the size of an image, or the maximal pixel value). In the semantical description the information about the contents of a data argument is described (e.g. what an image contains/what is seen in the image, like that it contains detected edges).

Context. General information concerning a certain application is described in the context (for example the type of cameras we use, or user constraints for the all-over image processing process).

In the integration of image processing operators, the following phases can be distinguished: how to choose among methods, how to initialize input arguments, how to evaluate results and possibly how to adjust the processing with the determination of new input values for programs or selection of other programs.

Choice criteria are used to choose, among all the available operators, the operator(s) which is (are) the most pertinent, according to the data description, the context and the characteristics of the operators.

The initialization criteria contain information how to initialize values of input arguments.

Evaluation criteria state the information how to assess the results provided by the chosen operator after its execution.

The adjustment criteria express information how to modify values of parameters after a negative evaluation.

The initialization and the adjustment criteria are attached to a certain operator, while the choice and the evaluation criteria are attached to a goal.

The static or descriptive concepts presented above (i.e. goals, operators, context, requests, arguments and data descriptions) are implemented in OCAPI with frames (or structured objects). The heuristic criteria concerning choice, evaluation, initialization, and adjustment, are expressed in production rules.

If we combine all these concepts we obtain a tree, which we call the operator decompositions tree. At the root of this tree there are the initial request, its goal, and the operators to achieve this goal. Every complex operator is a decomposition into substeps. Each of these substeps contains a request stating which subgoal to reach, together with the operators
to achieve it. The leaves of the tree are primitive operators (representing the image processing programs).

2.3 Reasoning mechanism

The main task of the control structure of OCAPI is to control the steps for solving a set of requests. The set of requests can be just the initial request, or a set of requests obtained by the decomposition of a complex operator. The algorithm that controls the execution of requests, is as follows:

1. global matching  
   while not all requests have been processed
2. selection of the request to be processed
3. classification of the operators (using choice criteria)  
   while the request is not satisfied
4. (A) initialization or adjustment of the parameters
5. (B) execution of the operator
6. evaluation of the results (using evaluation criteria)

Step 1 is a global match to verify that there exists at least one operator to solve the goals of the requests. Step 2 selects one request to process. Step 3 eliminates invalid operators, and performs a classification of the valid ones using the choice criteria. Step 4 chooses one operator among the valid ones. In step 5A the values of its parameters are determined using initialization criteria for the first attempt, and adjustment criteria for the next ones. Then, the chosen operator is executed (step 5B). If the operator is a primitive one, the execution is done by the interface with programs; in the case of a complex one, this algorithm is recursively called to control the execution of the requests of its decomposition. Then, the evaluation of the results (step 6) is achieved automatically using the evaluation criteria. In case of unsatisfactory results, another execution is performed after adjustment of the input parameters of the operator, or after choice of another operator.

3 Basis for the functional specification

3.1 Motivation

The desired problem solving behaviour we want to specify, is the reuse (planning and control of execution) of predefined software components, as performed by the OCAPI system described above. We are especially interested in describing the knowledge and knowledge structures that are required to generate this behaviour. The KADS expertise model has proved to be a successful approach for this purpose.

In contrast to some other methods for functional specification, KADS provides us with a predefined knowledge structure. As we will see in the next sections, this structure imposes a separation of the knowledge into three categories, namely domain knowledge, inference knowledge and task knowledge. Such a categorization enables us to use existing generic theories about different types of knowledge (like task level theories about planning), and to identify the reusability of different parts of the knowledge (see also section 5).

3.2 The KADS expertise model

The KADS expertise model [5] describes the problem solving behaviour of an agent in terms of knowledge that is being applied in carrying out a certain task. It specifies the desired problem solving behaviour of an agent and gives a categorization of the knowledge required to generate this behaviour. The model thus fulfills the role of a functional specification of the problem solving part of an artifact.

The categories or levels in which the expertise knowledge is analyzed and described are domain knowledge, inference knowledge, and task knowledge [7].

3.2.1 Domain knowledge

Domain knowledge is static domain specific knowledge. It is a declarative theory of the domain, represented as much as possible independently from the way it is used. It contains terms referring to the entities that must be distinguished as categories in the domain, and terms referring to the relations that can hold between the entities (domain ontology).

3.2.2 Inference knowledge

The inference knowledge describes the different types of inferences that can be made in the domain theory. It expresses how to use the domain knowledge. This is done in two ways: it specifies (1) the basic inference steps that can be made using the domain knowledge, and (2) the roles (knowledge roles) that the elements of the domain knowledge can play in the inference process. Although the inference knowledge specifies the basic inference steps, it does not specify any control
knowledge: no order is imposed on the various inference steps.

An inference structure is a combination of basic inference steps via input/output arguments. It describes what inferences can be made, but not how and when they are made.

3.2.3 Task knowledge

The task knowledge specifies how elementary inferences can be combined to achieve a certain goal, i.e. the control over the execution of the basic inference steps. It does this by imposing an ordering on these steps in terms of execution sequences, iterations, conditional statements, etc.

Three different types of tasks are distinguished. Primitive problem solving tasks are directly related to inferences (part of the inference knowledge). Composite problem solving tasks are further decomposed in sub-tasks (task decomposition). Transfer tasks are tasks of interaction with an external agent.

3.3 Knowledge description language

To describe these three knowledge categories and their interrelations, we use the semi-formal notation CML (Conceptual Modeling Language). CML [7] is especially developed for formulating KADS expertise models. The semi-formal descriptions in CML can be formalized using the language (ML)² [8]. In this article we present the descriptions in pictures with a graphical notation. This graphical notation is one-to-one translatable to semi-formal CML. CML on its turn, can be translated to (ML)².

4 Towards a functional specification of software reuse: OCAPI as a case study

In this section we present a description of the knowledge contained in OCAPI. We do this by using the KADS expertise model. First we introduce a hypothesis justifying this approach. Then we propose a possible knowledge description.

When needed, the knowledge description will be illustrated with some particular application domain dependent examples. Because this serves just illustrative purposes, they will be kept simple. We refer to the PROMETHEE knowledge base [9] for more details about the used application domain.

4.1 Hypothesis for justification of description

OCAPI (which is implemented in LeLisp) resides on the implementation level, and its concepts on the symbol level. The KADS expertise model resides on the knowledge level [6]. So we assume an isomorphism between the symbols of OCAPI and the epistemological meaning/sense of these symbols on the knowledge level. This assumption is the a priori hypothesis we make.

4.2 Description of the knowledge contained in OCAPI

In this section we present a description of the knowledge contained in the concepts and mechanisms of OCAPI.

4.2.1 Domain knowledge in OCAPI

Because OCAPI is a system that manipulates image processing operators, the domain layer of OCAPI consists of these operators plus the data they operate on. This information is presented in OCAPI with notions like goals and operators (abstraction), requests (integration), rules, data descriptions, and context. Let’s take a look at some examples² in the domain ontology.

Data and data description. Figure 1 contains a graphical representation of the CML definitions of the data. Remember that CML is a semi-formal notation for formulating KADS expertise models (see section 3.3). In this article we use a graphical notation that is one-to-one translatable to CML. Boxes represent concepts (classes). A box contains the name of the concept (above the straight line) and the properties with their value set (below the straight line). Arrows represent a subtype relation between concepts (or relations). Properties are automatically inherited from the parents, and could be differentiated (like e.g. filename of image is a differentiation of value OF data). Diamonds represent relations. An oval and an arrow indicate an expression, with the arrow pointing to the operand of the expression.

The data is simply named data. It has several properties, like a name, a value, a default value, etc. OCAPI identifies two types of data description, the implementation description and the semantical description. They are described with the has description relation.

²To be brief and comprehensible, no details about the information contained in a request or goal are presented.
In paragraph 2.2 we mentioned the concept argument as being part of the OCAPI concepts. What is meant there by the word argument, is not the data itself, but the role of the data with respect to an operator. Some data has the role of data argument, and other data has the role of parameter argument. Later, when we define an operator, this will become explicit.

**Context.** The ontology chosen for the context is shown in figure 2. A context has different parts. A context part can be a description of a real world entity (e.g. the camera that is used), or any other information concerning a certain application (e.g. user constraints for the all-over image processing process).

**Operator and arguments.** An operator is the central notion of the description, because it is the object that is manipulated in OCAPI. It is represented in figure 3. A circle in the upper right of a concept or relation means that the actual definition of that concept or relation is not in this figure, but somewhere else.

An operator is described as a relation between data. At one side, the data has the role of input data argument, at the other side of output data argument. There is also data that has the role of parameter argument. The operator characteristics are described as the properties of the operator relation.

Primitive operator and complex operator are both subtypes of an operator. A primitive operator is a representation of a concrete operator (that is part of a library of operators). An example of a primitive operator is hysteresis, which has an image as input data argument, a threshold as parameter argument, and an image as output data argument. Complex operators have a decomposition into substeps.

**Relations between data, data descriptions and context.** The domain ontology must also provide us with the terms to talk about the relations that
hold between data, data descriptions and the context. Some of these relations are necessary to perform the initialization, the evaluation, and the adaption. For the initialization we must e.g. be able to express that information about the data description is related to information concerning a parameter value. For example that the amount of noise in an image is related to the initial hysteresis threshold value. We must also be able to express that information concerning a context part is related to information about the data description. For example that the type of camera that is used is related to the amount of noise in an image. Furthermore, relations between information concerning operator properties (their characteristics) and data or data descriptions are needed to choose the most pertinent operators.

Finally, the domain knowledge contains statements about operator decompositions, providing the information contained in the operator decompositions tree.

### 4.2.2 Inference knowledge in OCAPI

How to use the domain knowledge is expressed by defining the different types of inferences that can be made with it. Some inferences directly concern the operator, like e.g. classify operators, select operator and decompose operator. Other inferences are e.g. initialize parameters, evaluate operator results, and adapt parameters.

![Figure 4: Initialization inference structure](image)

As an example, the inference structure concerning selection and initialization is shown in figure 4. In this figure ovals represent basic inference steps, and boxes represent knowledge roles. A set of operators is classified by the classify operators inference and becomes an ordered set of operators. Of an ordered set of operators the most pertinent operator is selected by the select operator inference. Both these inferences also use status information with knowledge roles pointing to (relations between) context part, data, data description and data format. The initialize parameters inference uses the parameters of an operator (with knowledge roles pointing to data), and again status information. The call operator inference is a transfer task, which we also added to this inference structure [7]. Other inferences and inference structures that are part of the functional specification are not presented, due to the limited space in this article. More details can be found in [10].

### 4.2.3 Task knowledge in OCAPI

![Figure 5: The control structure of OCAPI at the task level](image)

The task knowledge specifies how the elementary inferences can be combined to achieve a certain objective. This corresponds to the knowledge represented in the control structure of OCAPI. We defined the task decomposition presented in figure 5. The numbers in the figure correspond to the numbers in the algorithm that controls the execution of requests (presented in section 2.3).

### 5 Insights gained from the functional specification

In this section we explain the importance of having a functional specification (i.e. an implementation independent description) of the knowledge contained in OCAPI.

#### 5.1 General versus application specific knowledge

The description of the different types of knowledge and the structure we imposed on this knowledge, can be
used to identify different parts of knowledge with different functionalities. If we look at the domain knowledge, we can identify the following parts:

1) General information concerning the domain of planning and supervision of programming constructs. For example the data, data description, context, context part, operator, primitive operator, etc.

2) Information concerning a certain type of programming constructs (like image processing, mathematical routines, etc.). In the presented ontology these are image, threshold, histogram, camera, image processing operator, etc.

3) Application domain dependent information (like a certain image processing application), e.g. road image, obstacle detection, car detection, etc.

This categorization helps us to identify the reusability of different parts of the knowledge. Some parts of the knowledge can be defined as fixed or static, and others as reusable. We could e.g. define the general information concerning the domain of planning and supervision of programming constructs together with the information concerning the planning and control of execution of a certain type of programming constructs, as static (e.g. knowledge about how to plan and control the execution of image processing programs). Then the application domain dependent information is exchangeable with other application domain dependent information. It has become an interchangeable, reusable module (e.g. a module for car detection in road scenes, a module for mathematical morphology, or a module for astronomical image analysis).

Another possibility is to define only the general information concerning the domain of planning and supervision of programming constructs as static. Then the information concerning a certain type of programming constructs is interchangeable with information concerning another type of programming constructs. In that case we could e.g. exchange a module containing image processing routines (including the knowledge about how to use them), with a module containing numerical programs, or a module containing signal processing programs.

5.2 Planning capabilities

As we saw in paragraph 3 and 4 the task knowledge contains different aspects of how to plan and control the execution of image processing programs. The control structure of OCAPI (see figure 5) is based on an existing operator decompositions tree. This approach corresponds to hierarchical skeletal based planning [9]. The dynamic aspects of this kind of planning are limited, and can be improved by introducing a form of nonlinear planning, in which the software components do not have a predefined fixed ordering with respect to each other. This, on its turn, gives need for adapted domain knowledge descriptions, like e.g. adapted descriptions of the operators and the data, and the use of pre- and postconditions.

5.3 Formalization

Although the graphical representations of figure 1 and further are all directly translatable to the KADS conceptual language CML, this language is still informal. A formalization of the knowledge description helps to validate the knowledge, and to identify wrong, missing or superfluous definitions. (ML)” [8] fills the gap between symbol level and knowledge level, being a formal language that allows modeling at the knowledge level. It allows the formalization of the KADS expertise model. A formalization of the knowledge contained in OCAPI, can thus help us to better identify different (reusable) parts of the knowledge and missing knowledge for making it more generally applicable. Our analysis of OCAPI in terms of a conceptual KADS model is a first and necessary step towards such a formal specification of OCAPI.

5.4 General insights

So, looking at the functional specification we can see that:

- It helps to make implicit structures explicit, like an OCAPI-frame (e.g. operator) that plays more than one role in the inference process (e.g. as item to plan, and as item that manipulates data).

- It allows to identify missing and irrelevant knowledge, like missing knowledge about operator planning (as we saw in paragraph 5.2).

- It enables to use existing generic theories about different types of knowledge. We can e.g. compare the task and inference structures of OCAPI with generic KADS models for planning and monitoring.

- It enables to evaluate the aptness of the structure of the knowledge, like e.g. if the structures of data description (implementation and semantical) are sufficiently expressive for the inferences that make use of them.
- It helps to formalize the description (see paragraph 5.3).
- It is easier to comprehend than the implementation description. For example the data representation in figure 1 is much easier to comprehend than the OCAPI code in which this is implemented with representation details of frames, slots, pointers, etc.

6 Conclusion

We have taken the first steps towards a functional specification of reusing predefined software components, by presenting a case study: a description of the knowledge contained in OCAPI, using the KADS expertise model. With the results of this case study we were able to evaluate the structure and contents of the knowledge, and could so identify reusable parts, and less developed knowledge structures. Hereby we have provided ourselves with the basis for abstracting from the OCAPI domain (image processing), to a functional specification of reusing predefined software components in general.

More steps must be taken to reach this aim. The description of the different types of knowledge and the structure we imposed on this knowledge have to be investigated in more detail, using information of the field of software reuse [2], [11], [12], problem solving [13], meta-architectures [14], planning and machine learning. Especially the planning (and replanning), but also the data and the way the data are represented and used are important research topics. In addition to this, the formalization of the knowledge is an item on the agenda.

References