Reasoning with Design Knowledge for Interactively Supporting Framework Reuse

W.De Meuter[1] M.D'Hondt[2] S.Goderis[1] T.D'Hondt[1]

[1]Programming Technology Lab, Vrije Universiteit Brussel, Belgium
[2]System and Software Engineering Lab, Vrije Universiteit Brussel, Belgium

wdmeuter | mjdhondt | sgoderis | tjdhondt@vub.ac.be

Abstract

The paper explains our efforts in using a rule-based reasoning engine to actively support reuse of object-oriented frameworks. The engine is embedded in the programming environment and is able to reason about the developed source code. By means of explicitly encoded design knowledge, it is able to actively guide the reuse process by intelligently assisting the programmer in taking the appropriate reuse steps. Reuse thus becomes an interaction between the programmer and the reasoning engine.

Two conducted experiments are explained in this paper, illustrating the applicability of such an embedded rule-based inference engine to explicitly guide both black box and white box reuse.

1 Introduction and Hypothesis

One of the major consequences of the popularity of object technology is the advent of *object-oriented frameworks* [Inc89] [CHSV97] [GAL97]. As opposed to a single software application, a framework is a reusable software system that offers a solution to a family of related problems. Repeatedly reusing a framework results in a number of similar software applications that only vary in well-defined places. These variations on a common theme are achieved when the framework is constructed as a well-designed *skeleton* of code that is instantiated by filling in the so-called *hot spots*.

Currently, a distinction is made between *white box reuse* and *black box reuse* of a framework [JF88]. In the case of the former the framework's hot spots are implemented as abstract classes or classes with default behaviour, and reusing the framework requires techniques such as overriding and subclassing. Hence the reuser needs to know the details of the framework design and implementation in order to make controlled changes. Black box reuse involves configuring highly parameterised existing parts of the framework rather than adding new behaviour. This requires techniques such as polymorphism and parameterisation. In this case design and implementation details do not need to be exposed

to the reuser, who nevertheless needs to be aware of the existing parts in order to select the correct ones and compose them.

The aforementioned reuse processes indicate that good framework reuse depends on the availability of *documentation* [Joh92]. Whether white box reuse or black box reuse is applied to a framework, the documentation should clearly describe the intended behaviour and use of the framework, and provide a thorough explanation of its design and implementation. The documentation has to be informal enough to be easily understood by framework reusers, without becoming ambiguous.

However, current state of the art does not provide satisfactory techniques for framework reuse documentation. One technique is documenting frameworks with *patterns* [Joh92] [RJ96], which state the purpose of the framework, describe how to reuse it, and explain its design and implementation. Another way framework reuse is supported, is by means of a *cookbook* [KP88] [Dig95] which provides standard recipes offering a step by step explanation of how the framework can be reused. In both techniques, several problems arise:

- The documentation is written down in a variant of natural language, even if some kind structuring is applied. This results in informal and thus ambiguous documentation.
- It is not always clear what pattern or recipe to use, or the reuser does not even realise there exists a suitable one. A mechanism, similar to a wizard, is missing to interactively guide the reuser through the correct reuse process.

We observe that documentation to guide framework reuse is *knowledge*. In artificial intelligence numerous successful knowledge representation schemes have been developed, accompanied by powerful inference engines that are able to reason with this knowledge [LS98]. Therefore, our hypothesis is that *an expert system whose knowledge base consists of the design and reuse knowledge of a specific framework, can interactively guide the reuser and enhance the quality of the reuse process.*

The next section explores the requirements of an interactive guiding system for framework reuse. Section 3 describes the technological choices we made corresponding to these requirements in order to develop a concrete system. Section 3 sketches two concrete experiments we conducted with this system regarding black box and white box reuse in order to prove our hypothesis. Section 5 concludes.

2 Requirements for Supporting Interactive Framework Reuse

A first requirement is that framework development should still be performed in a standard object-oriented programming language, since existing techniques

and methods ought to remain applicable. The expert system with design and reuse knowledge should be integrated in the development environment of this language. We elaborate on the language of choice, Squeak, for proving our hypothesis in Sec. 3.1.

Knowledge about framework reuse involves information on how a specific framework should be reused and on framework reuse in general. Because we want this knowledge to be easily maintainable upon evolution of the framework, the knowledge should be explicitly represented and separated from the reasoning algorithm that uses it to interactively guide the reuser. This reasoning algorithm should be a *forward chainer* because inferring the steps in the process of framework reuse, is a form of *goal-less reasoning*. Indeed, no goal is known beforehand that the reasoning algorithm needs to prove, as is the case with a backward chainer. The forward chainer will instead try to generate possible solutions starting from initial facts. Another crucial property of forward chainers is that access to their state is at all times possible because - as opposed to backward chainers - no backtracking is performed. The system we selected, KAN, is described in Sec. 3.2.

Another requirement of our system for guiding framework reuse, is that it has to be interactive, similarly to wizards. The control over the reuse process is in turn with the system and with the reuser. We thus envision a system to which a reuser can turn for help. Hereupon the system takes control and offers a number of possible reuse recipes to the reuser. Upon regaining control, the reuser reacts by taking steps to perform one of the recipes. When certain steps have been performed by the reuser, the system is able to take control for further instructions, remember what still needs to be done, and adapt its advice to the concrete situation the reuser is in. This illustrates another requirement for our system: it has to be *coupled* to the framework implementation, in other words as a meta-system that reasons about the code level. How we incorporated this coupling mechanism in our system is explained in Sec. 3.3.

3 Experimental Setup

The following describes the ingredients of a system for interactively supporting framework reuse that are necessary to fulfil the requirements listed in the previous section. The choices Squeak, KAN and the coupling mechanism discussed above are explained and motivated.

3.1 Squeak

The reasons for choosing Squeak [IKM+97], an open-source Smalltalk environment developed at Disney Imagineering, are:

- We decided to use Smalltalk because we consider it to be the pearl of object-oriented programming. This is especially important in framework development as this requires extreme flexibility in order to achieve the

highest possible reusability. The Squeak Smalltalk implementation is particularly interesting because it has a very active user community.

- The expert system needs to be plugged into the programming language in such a way that it can take control after certain actions of the reuser. Currently, only Smalltalk with its successful tradition of adaptable programming tools such as browsers and finders meets these high standards of openness. Although development environments for other languages (notably Java) are catching up swiftly, we still feel that there is currently no widely used environment that can stand up to the comparison with Smalltalk environments.

3.2 The KAN Forward Chainer

Since artificial intelligence has a multitude of knowledge technologies at hand, we made it our explicit goal not to try to find something new, but instead to select an existing solution. We opted for KAN, a 'lean and mean' expert system shell, and implemented it on top of Squeak.[1] We will not give a full account of KAN, but refer the interested reader to [Ste92] for the specification of KAN as a language and to [God00] for an elaboration of the version we implemented.

Figure 1 contains some code excerpts supporting our discourse. The code comes from a small expert system whose task is to classify finches based on their external properties.

```
(define (ruleset finch-type) finch-rules)

(define (rule finch-rules) diamond-firetail-rule
  (if
    (beak red)
    ... )

  (then
    (conclude (species diamond-firetail))
    (communicate "the species is diamond firetail")
    (investigate identity-determination-rules)))

(define problemsolver finch-classification-solver
  (object (a finch-type))
  (ruleset finch-rules)
  (goal species))
```

Figure 1: Some KAN code

[1] As a matter of fact, KAN was implemented *in* Squeak, but the implementation details are outside the scope of this paper.

KAN is an expert system shell whose entities (fact base, rules,...) are internally organised as frames of slots. For example, a rule is a frame containing an `if` slot and a `then` slot. Every slot needs a filler for that slot. The filler for an `if` slot is a conditional expression. For the `then` slot of a rule, the filler is a sequence of actions.

The main KAN frame is called a *problem solver*. A problem solver consists of a `ruleset` slot, an `object` slot[2] and a `goal` slot. As an example, Fig. 1 shows the declaration of a problem solver `finch-classification-solver`. A goal is a monitor on some fact that becomes satisfied when the fact is indeed concluded to be true. The rule set the problem solver will use for reasoning is the `finch-rules` set. The idea of rule sets is that the forward chainer cycles through the rules in the set until the goal is reached or until no more new facts are derived. Fig. 1 shows one exemplary rule `diamond-firetail-rule` of the `finch-rules` rule set. The rule concludes that a finch is a diamond firetail finch (and informs the user about this) whenever the finch has a red beak and a few other properties.

One of the actions the conclusion of a rule can contain is `investigate`, permitting rule sets to be hierarchically organised. In [Ste92] this hierarchical organisation of rule sets is aligned with the approach of organising expert systems around 'tasks' to be solved. A rule set corresponds to such a task and an `investigate` action corresponds to the spawning of a sub task. Rules can also contain `communicate` actions to display a message to the user, and `conclude` actions that conclude a new fact in the fact base of the problem solver. In Fig. 1 we can see that when the `diamond-firetail-rule` succeeds, it will conclude the species of the observed finch, communicate this finding to the user and call the `identity-determination-rules` rule set to investigate whether the observed finch was already encountered before and if so, which specimen is being observed.

3.3 Coupling KAN to Smalltalk

We extended the original KAN design with facilities to access the underlying framework implementation written in Smalltalk from within a running expert system. Two constructions where added to the original KAN language:

1. The first construction is the `(smalltalk <name>)` primitive. This expression looks up a name in the Smalltalk runtime[3] and injects the corresponding Smalltalk object into KAN. This is achieved by simply wrapping the Smalltalk object as a KAN value at the level of the KAN evaluator.

2. The second construction is a primitive `(send o selector [o1 ... ok])`, which can appear in conditionals expressions as well as in the conclusion actions of rules. It requires that all sub expressions `o`, `o1`, ..., `ok` evaluate

[2] In KAN terminology, an object is just a fact base. This has nothing to do with the term 'object' in the object-oriented sense.

[3] More precisely, the name is looked up in the global dictionary `Smalltalk` which contains all the global variables.

to Smalltalk objects as described above. When this is the case, the message `selector` is effectively sent to the Smalltalk counterpart of `o` with `o1` to `ok` as arguments. This is accomplished by unwrapping the receiver and the arguments in order to get actual Smalltalk objects, which in turn are used as parameters in the `perform:withArguments:` primitive from the Smalltalk meta object protocol. Finally, the resulting Smalltalk object is wrapped again in order to inject it back into KAN.

These two constructions grant full control over Smalltalk from within KAN, as in Smalltalk *everything* is done using messages (even making classes e.g.).

4 Experiments

We employed the technology described in the previous section to support our hypothesis that rule-based expert system technology indeed enables active framework documentation. We achieved by designing two tiny expert systems that guide a programmer in the reuse process.

4.1 Experiment 1: Black Box Reuse

The first experiment involves the applicability of our approach to guide a black box reuse process. To confirm this conjecture we used our KAN shell to conceive a problem solver that actively guides reuse of the Smalltalk collection hierarchy. This hierarchy consists of more than 80 Squeak classes which offer a wide variety of containers programmers might need. The stereotypical reuse of this hierarchy is simple black box reuse: programmers identify the right collection kind and use it as is. The problem with this is that most programmers are only aware of some frequently used general collections such as `Array` or `Dictionary`, while the collection hierarchy contains many highly specialised ones like `B3DColor4Array` to name just one. Even though the classes in the collection hierarchy *do* document their intended use, this documentation is passive. Programmers unaware of some useful collections will never use them unless other programmers inform them of the existence of these collections. The setup of this experiment was to assist this occasional passer-by with an expert system that actively guides the reuser in selecting the right collection hierarchy.

In order to reuse the collection hierarchy, it suffices to send the `reason` message to the `Collection` class which is the root of the collection hierarchy. This triggers a class method that will create and launch a problem solver. The problem solver uses pop-up menus like the one shown in Fig. 2 to interrogate the user about the properties the target collection class is supposed to have. E.g., the question in Fig. 2 is needed because different collection implementations have been optimised depending on the kind of objects they will have to contain. When the target collection kind is known, the reasoning process finishes by opening a browser for that collection class. This is possible because the `send` primitive we discussed in Sec. 3.3 funnels that message to the Smalltalk level.

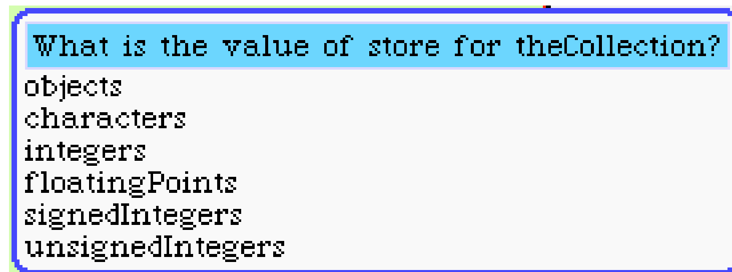

Figure 2: A pop-up menu for multi-valued questions

The problem solver has several rule sets, some of which are purely administrative such as the topmost rule set which only initialises the problem solver, then calls the 'real topmost rule set' and finally terminates the problem solver by opening a browser and concluding the goal. The other rule sets actually implement a traversal through the inheritance hierarchy. Based on yes-no questions and multi-valued questions (like the one in Fig. 2), the rule sets 'walk down' the inheritance hierarchy until a concrete collection kind is concluded. The topmost rule set figures out which kind of functionality is required (dictionary, set,...). In the case of different variants of such a collection (i.e. when it has subclasses) a dedicated rule set is spawned to refine that initial conclusion.

The issue of maintaining the documentation rules upon changes in the hierarchy remains largely open. Concerning the rule sets, there are two possible extensions that can be made to the hierarchy. Making a first variant of a class (i.e. making the first subclass) implies adding a new rule set and making sure this new rule set is called from within the existing rule set. Making a new variant of a certain collection (i.e. making a subclass of a class that already has subclasses) implies adding a rule to the rule set that decides which subclass to use. We experience this artisanal maintenance of the documentation as inadequate. It is one of the issues on our current research agenda.

4.2 Experiment 2: White Box Reuse

In our second experiment the expert system shell was used to actively assist white box reuse of a framework. Before moving on to the experiment itself, we first elaborate on the framework on which the reuse experiment is based. We chose a LAN framework [Luc97] we developed to serve as didactic artefact in the courses on reuse we teach at our university. This framework simulates a simple circular LAN network. The main elements of the framework are nodes (with subclasses for workstations and printers) and packets. Nodes work together to make packets go around the LAN. Packets have the responsibility to ensure that they are addressed to those nodes. There are different kinds of reuse we could think of. The existing family can be extended with new kinds of nodes, output-servers, packets, packet delivery systems (e.g. broadcasting), addressing

schemes, and so on. However, in order to make these extensions correctly, the reuser has to follow explicit reuse recipes that reflect the design of the framework. These reuse recipes are encoded in the active cookbook we conceived. The idea of the active cookbook is that there is a problem solver telling the user gradually which classes or methods to create, after which it is suspended to allow the user to effectively perform these tasks. At this point the user has full control. He can program whatever he wants, but as soon as these tasks have been carried out, the problem solver restarts. As such, the problem solver is continuously being suspended (by itself) and restarted (by the programming environment in which the user is operating).

In order to comfortably express this interactive behaviour, we had to enrich the theoretical coupling of Sec. 3.3[4]. First, we extended the KAN formalism with a new kind of action (`suspend`) to be used in rules. This suspends the problem solver, thereby keeping its internal state for later resumption. Second, we extended the Smalltalk programming environment with a so-called *task pool* containing 'tasks the programmer still has to do'. From the side of the problem solver, the **`send`** construction can be used to post tasks in the task pool. Currently, we use two kinds of tasks, one to express that a certain method has to be added to a class, and the other one to express that a subclass has to be made of a certain class. Furthermore, both tasks have a flag associated to them which expresses whether or not the problem solver has to be resumed after the task has been executed. From the side of the programming environment, each time the programmer writes a new method or class, the task pool is consulted. If it appears to contain a task that matches the method or class just added, the task is removed from the pool and if its flag requires so, the problem solver is resumed. Based on the knowledge that the task has been fulfilled, it can continue the reuse process it was guiding by posting new tasks in the pool. Notice that it is thanks to Smalltalk's openness that we were able to change the programming environment such that it calls our problem solver when needed[5].

The problem solver we implemented is started by sending `reuse` to any class of the framework. It then works in two phases. In the first phase, a rule set interrogates the user and determines which reuse recipe the reuser has to use. Of course, if there is no recipe that fits the need of the programmer, the problem solver stops. When the right recipe is determined however, a rule set that goes through the different reuse steps (adding methods and classes) is executed. This rule set posts the reuse tasks yet to be performed in the task pool and subsequently suspends itself.

An excerpt of the problem solver guiding reuse of the LAN framework is shown in Fig. 3. It concerns three rules from the recipe that describes how to add a new kind of printer. To do this the user needs to create a new subclass

[4]This extension is not fundamental to our approach. The construction explained in Sec. 3.3 allows us to send *any* message to any Smalltalk object from within KAN. So, in theory, we can do everything in KAN. However, KAN as a paradigm is unsuitable to implement things which are extremely 'sequential' in nature.

[5]For the technical details we refer to [God00]. The essence of the adaptations consists of changing the browser at the point of class and method accepts.

```
(define (rule printerrules) addNewPrinter
  (if (not printerclassdone))
  (then (send smalltalk TaskPool classNeeded:solve:[Printserver true])
        (conclude printerClassDone)
        (suspend)))
(define (rule printerrules) printerAdded
  (if (printerClassDone)
      (not printerMethodsDone))
  (then (send smalltalk TaskPool methodNeeded:forClass:solve:
              [isDestinationFor:
                 (send smalltalk taskPool lastClassAdded []) true])
        (conclude printerMethodsDone)
        (suspend)))
(define (rule printerrules) newDocument
  (if (printerMethodsDone)
      (not documentClassDone))
  (then (send smalltalk TaskPool
              classNeeded:solve: [Document true])
        (conclude documentClassDone)
        (suspend)))
```

Figure 3: A part of the 'add new printer' reuse recipe

of `Printserver` with a method `isDestinationFor:` that will be called by the framework. Furthermore, a new subclass of `AbstractDocument` is needed. This is clearly stated in the rules of Fig. 3. The first rule simply says that the task pool needs to know that a new subclass of `PrintServer` is needed and that the problem solver has to resume when this is done. It subsequently suspends the solver. After the solver has been restarted, the second rule dictates that when this class was indeed added, the task pool has to be informed a new method has to be added to that class. A reference to the newly added class is asked from the task pool using `lastClassAdded`. The final rule tells the reuser to create a new subclass of `AbstractDocument`.

Notice that, because the problem solver's state is preserved, properties like `printerClassDone` are still known. Thus, the first rule will no longer fire but the second rule will, since one of the preconditions of this rule is `printerClassDone`.

Future work in this research area consists of making the reuse documentation active so that it automatically evolves together with the framework. Currently, the reuse documentation - represented in the knowledge base of the expert system shell - is able to dictate where and how the framework should be adapted for reuse, but unable to adapt itself accordingly.

5 Conclusion

Although software reusability is spectacularly enhanced as a result of the use of object-oriented framework technology, good documentation is still indispensable to help it attaining its full potential. The fundamental problem is that documentation has to be unambiguous and interactive, the latter because it is often not clear what framework features to reuse, or that there exists a suitable feature in the first place.

We argued that expert system technology is a good stepping stone to alleviate this problem. To prove this hypothesis, we built an expert system shell on top of a Smalltalk programming environment, designed in such a way that it allows the construction of interactive support for framework reuse to guide the reuser through the reuse process. We conducted two experiments that are representative for two typical reuse processes, black box and white box reuse.

Although impossible to prove in a mathematical sense, we feel that our experiments convincingly show how both black box and white box reuse of frameworks are supported and enhanced by this blend of state of the art object technology, meta-programming and artificial intelligence.

References

[CHSV97] W. Codenie, K. De Hondt, P. Steyaert, and A. Vercammen. Evolving custom-made applications into domain-specific frameworks. *Communications of the ACM*, 1997.

[Dig95] ParcPlace Digitalk. Visualworks cookbook. 1995.

[GAL97] A. Goldberg, S.T. Abell, and D. Leibs. The learningworks development and delivery frameworks. *Communications of the ACM*, 1997.

[God00] Sofie Goderis. A reflective forward-chained inference engine to reason about object-oriented systems. Technical report, Programming Technology Lab, Vrije Universiteit Brussel, 2000.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *OOPSLA '97 Proceedings*, 1997.

[Inc89] Apple Computer Inc. Macapp programmers guide. 1989.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Object-Oriented Programming*, 1988.

[Joh92] R.E. Johnson. Documenting frameworks using patterns. In *OOPSLA'92 Proceedings*, 1992.

[KP88] G.E. Krasner and S.T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Object-Oriented Programming*, 1988.

[LS98] G.F. Luber and W.A. Stubblefield. *Artificial Intelligence, Structures and Strategies for Complex Problem Solving*. Addison-Wesley, 1998.

[Luc97] Carine Lucas. *Documenting reuse and evolution with reuse contracts*. PhD thesis, Programming Technology Lab, Vrije Universiteit Brussel, 1997.

[RJ96] D. Roberts and R. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. At URL: http://st-www.cs.uiuc/ edu/users/droberts/evolve.html, 1996.

[Ste92] Luc Steels. *Knowledge-based systems*. Addison-Wesley, 1992.