Black & White, Never Grey: On Interfaces, Synchronization, Pragmatics, and Responsibilities

Franz Puntigam
Technische Universität Wien
Institut für Computersprachen
Argentinierstraße 8, A-1040 Vienna, Austria
E-mail: franz@complang.tuwien.ac.at

***Abstract*—When composing systems from components we have to deal with involved aspects like synchronization or non-functional properties like performance. It is nearly impossible to clearly specify such aspects in interfaces. Looking behind the interfaces (into grey boxes) does not solve the problem because of lost substitutability. In this paper we explain on the example of synchronization, first, how pragmatic descriptions solve the problem in usual cases and, second, that moving and splitting responsibility for synchronization between components is helpful in further cases. We argue that there is a general pattern applicable to many functional and non-functional aspects behind this solution.**

I. Motivation and Overview

The most important concept in object-based programming is data abstraction, this is the encapsulation of data and routines in objects together with data hiding. We regard objects as black boxes the internal structure of which is hidden from the rest of the world. Since users of an object do not see and (more importantly) have no direct access to implementation details, we can safely change these details and even substitute the object for another similar one without breaking code. On this substitutability property we build successful concepts like inclusion polymorphism (with subtyping and dynamic binding) in object-oriented programming as well as the composition of systems from components.

While we use concepts and tool based on data hiding we often experience the fact that data hiding poses limits on the usability of objects and components. We are led into temptation to look inside the box and make use of some parts we find there, this is, we regard the box as being grey instead of black. Doing so usually gives us an immediate advantage: We have to write less code, get better performance, and can show useful properties. Often the pain follows later when we detect that our system does not work properly after substituting objects or components. Then, we remember why our tools required black boxes. Even worse, especially when composing components there are situations where we intentionally give up the idea of black boxes and use grey boxes instead, knowing that it will be difficult to replace them later on; sometimes the only other alternative would be to program everything from scratch. Such situations give raise to the question whether black boxes are really that successful when dealing with components.

Actually no component is completely black because interfaces give needed information to users. This information is considered to be stable and, therefore, we regard interfaces as white boxes. In this paper we distinguish between components as

- black boxes with white boxes at their entries where interfaces clearly specify which portions of the components are white and stable, and the unstable black rest is completely hidden from the user,
- and grey boxes where even unstable portions can become visible.

Of course, a balanced combination of black and white is preferable over grey since we can get substitutability and the needed stable information at the same time.

Conventional interfaces (= signatures) cannot give all the information needed by users. For example, users need information about synchronization in the component to avoid synchronization conflicts that can lead to deadlocks and livelocks. Hence, we often think of grey boxes as something we cannot avoid in practice.

In the author's opinion, rich interfaces and pragmatic models can be very effective to reduce the need of grey boxes. It is not even necessary to wait for new technologies (although new techniques slowly coming up promise to be quite helpful in this respect) because clear contracts and component descriptions written in a natural language are often all we need. However, the basics of many contracts in software are still not understood well enough. For example, even seemingly harmless synchronization primitives somewhere deep in the code of a component can cause synchronization conflicts. If we solved this problem by putting all synchronization information into the interface, we would have to consider much more program parts as stable than we like to, causing many useful component substitutions to be impossible. We will discuss such topics in this paper.

In Section II we point out problems with synchronization in black boxes and how they are usually addressed in practice. In Section III we argue that it can be quite useful to move responsibility for synchronization from a component to its user. Next, in Section IV we generalize what we have seen so far to further aspects. Finally, in Section V we give some concluding remarks.

II. Synchronization and Informal Interfaces

We deal with synchronization whenever we require an action to be performed before another one as well as when we simply perform actions in sequence. In sequential programs, synchronization is implicit in the control flow, and in concurrent programs we handle it explicitly.

Synchronization is a very complex topic because it can be conflicting. For example, if a component performs an action A only before another action B while a user insists on B to be performed before A (by invoking B before A) we have a deadlock. The user must know the behavior of the component to avoid such conflicts. Unfortunately, the requirement of "A before B" may not be statically determinable; it may occur only under certain conditions like full buffers that are not predictable in practice.

From playing with this and similar simple examples we can learn the following:

- We expect good components not to cause deadlocks. Unfortunately, this statement is more of a wish than reality: Deadlocks often result from the interaction between components. We usually find no deadlocks when testing components in isolation. Of course, good components make it more unlikely to experience deadlocks in realistic environments.
- Static deadlock prevention is usually no option because of its restrictive nature. Actually it is easy to find a set of syntactic constraints to ensure programs to be free of deadlocks. However, each such approach restricts the set of programs conforming to the constraints in a severe way that is not acceptable in most cases. Statically ensuring other liveness properties (like livelock prevention) restricts conforming programs even further.
- In the few cases where we absolutely need static deadlock prevention we have to design the whole system (or subsystem) with this goal in mind. This means, components can be used only if designed with the same goal in mind; then, interfaces statically specify the synchronization behavior. A simple analysis of a component (as a grey box) is rarely helpful in this respect.
- Without static deadlock prevention we will sometimes (hopefully not too often) experience deadlocks. To find out the cause of them it is helpful to inspect the source code of the component as a grey box. Without a proper documentation and without access to the source code it can be quite difficult to find the reason for the deadlock. Once we know the problem we can usually avoid it without any need to change the component (except if we found a problem in the component itself).
- It is difficult to describe all cases that can lead to a deadlock when interacting with a component. Most interface specifications do not tell much in this respect. One of the simplest ways to specify all possible deadlocks is to describe the complete control flow by, for example, an automaton. However, such approaches turn black into nearly white boxes.
- Without clear interface specifications it is always an open question who is responsible for a problem, the component or its user.

These considerations do not give much hope that we can avoid grey boxes by giving appropriate interfaces in the context of synchronization although such interfaces are important to arrange responsibilities. Other liveness properties (like livelocks) are even more difficult to handle. Fortunately, there is another side of the coin as we will see below.

Deadlocks do not occur that often in practice. Obviously, users and developers of components already employ some sort of (informal) interface specification that includes synchronization behavior. In general we have a description that tells users what a component does and how to use it (often in the form of an application programmer interface). Even if the description does not specify the component's synchronization behavior in detail, it shows the overall structure, and experienced users are able to derive the most likely synchronization behavior just from this description. Only in cases of unexpected synchronization patterns the documentation gives hints on them. This pragmatic and mostly implicit kind of communication between developers and users works surprisingly well.

Simplicity of synchronization patterns and their pragmatic use is an important precondition for this communication. Theoretically we can imagine a huge number of possible synchronization patterns, but only a handful of them actually occurs. For example, with a mechanism to ensure mutual exclusion ("synchronized" in Java) and another one to avoid underflows and overflows we cannot just develop buffers, but almost everything that needs synchronization. Furthermore, it is not necessary to mention mutual exclusion in an interface of a component: If the code running in mutual exclusion terminates in finite (and for practical reasons short) time, then mutual exclusion cannot cause synchronization conflicts [10]. (Of course we must specify in interfaces if a routine expects to be invoked in mutual exclusion and does not ensure mutual exclusion by itself.) Synchronization to avoid overflows and underflows in a component can be in conflict with (often unintended and accidental) synchronization by the user. Fortunately, it is usual to describe in the interface what happens in the case of an underflow or overflow although this description often does not refer to synchronization.

By using "wait" and "notify" we can produce arbitrarily complex synchronization patterns. However, it is better not to do so. The implicit communication between developers and users works well in the simple cases mentioned above (and in some further simple cases), but not in general. It is extremely difficult to describe complex synchronization patterns in interfaces. Moreover, complex synchronization patterns are often less stable. Since they should be specified in interfaces so that users can avoid conflicting synchronization, there is a danger that we have to either keep inappropriate synchronization behavior in future versions or change interfaces and thereby lose substitutability. Good components keep their synchronization behavior simple and as usual (this is, pragmatic).

There is a tendency to support more restrictive synchroniza-

tion primitives that can be better expressed in an interface. For example, in Polyphonic C# [2] (based on the Join calculus [3]) we combine routines like “put” and “get” in a buffer to a chord to be executed as a single unit. Users see in the interface how routines in a chord are synchronized. Since only one routine in a chord is executed synchronously and all other routines are asynchronous, synchronization with chords is less expressive but more visible than that with “wait” and “notify”. However, even a component written in Polyphonic C# can have synchronization not visible in the component interfaces. The interfaces show only synchronization that happens when invoking component routines from outside, but not that occurring when invoking them from inside. Such languages help to avoid the worst cases. However, they cannot solve the problem.

We can argue that it is absolutely necessary to specify the complete synchronization behavior of a component in its interface (thereby turning a black box into a nearly white one) because users need this information. Once users rely on such information we cannot easily change the communication structure anyway. Perhaps there is much truth in this argumentation if we consider arbitrarily complicated communication patterns. Fortunately, current software engineering practice shows that in many situations users have all needed synchronization information without turning black into white boxes. It is an open question whether we have to show all synchronization behavior in the remaining cases. Of course, we want to avoid that if possible. In the next section we deal with required synchronization as a way to specify partial synchronization information (as opposed to complete synchronization information) in an interface. It is our goal to keep as much information as possible hidden in the black box.

III. Required Synchronization

Under required synchronization we understand the synchronization that a component requires from the user when invoking routines.[1] For example, many components require from a user to invoke an initializing routine before any other routine. This is clearly a kind of synchronization although we need no synchronization primitive like “synchronized”, “wait”, and “notify” to ensure it.

Users can provide the required synchronization only if they know exactly in which ordering routines are invokable. The synchronization depends only on data available to both the component and the user. Thus, it is much easier to specify required synchronization in interfaces than other kinds of synchronization. Required synchronization is specified in the interface and usually only there. Such properties cannot be derived from the implementation (since there are no explicit synchronization primitives), and the component is always a black box in this regard.

[1]Required synchronization has nothing to do with require interfaces of components. There is just an unintended similar naming. In this paper, a require interface does not differ from any other interface and, hence, can also specify required synchronization. Required synchronization in a require interface specifies synchronization to be ensured by the component itself because the component acts as client of the unit referred to by the require interface.

Required synchronization can be very expressive. We can specify arbitrary sequential orders of routine invocations, alternatives (where always the user selects the alternative to be taken), and of course arbitrary interleavings thereof. This means, we can express each possible prefix-closed set of routine invocation sequences (or trace set). The expressiveness need not depend on the language we use in the interface – a plain natural language or any formal language able to express trace sets. Substitutability of required synchronization is also easy and well-defined: A subtype must essentially support all routine invocations in all orderings supported by the supertype, this is, the trace set corresponding to the subtype includes that corresponding to the supertype (just set inclusion).

The difficulty is rather at the side of the user who must ensure that routines are invoked only in an ordering specified in the interface. At a first glance this seems to be easy since the user has all needed information. However, in most cases there is not just one component with a single well-defined user. In systems composed from components, each component can provide services to and use services from several other components. If several users interact with the same component, the users must coordinate themselves to invoke routines in a required linear ordering. There are several ways to ensure linearity:

- The simplest solution is to require only a single user whenever there are constraints on the ordering of routine invocations. In simple cases (like initializing routines to be invoked before others) this solution is absolutely appropriate. It is easily enforceable and expressible in a natural language.
- In the case of several users we can ensure that at each point in time only a single user interacts with the component. This solution requires more effort on coordinating users, but it is doable. However, without tool support and without a clear concept how to do it the approach is error prone. An obvious disadvantage is the strict linearity (this is, a single user at a time) even for independent routine invocations.
- To improve the approach we add support for splitting a single linear thread into several independent linear threads and combining them again when threads become dependent. In this context a thread can be an independent instance of any mechanism to ensure linearity, and it can also be a concurrent thread of control in concurrent programs. This improved approach is the most flexible one, but it is difficult and error prone without proper tool support.

Tool support has been proposed for single as well as multiple linear threads [1], [4], [5], [6], [7], [8], [9], [10]. However, such support is currently not widely available. At the moment it is only possible to simulate the corresponding techniques by writing annotations in natural language into the users’ code and check them by hand.

Although the most advanced approaches are possibly not yet mature, required synchronization is in many cases preferable

over more complex synchronization using synchronization primitives in components (except of mutual exclusion and a handful simple synchronization patterns). In a large majority of cases we need to ensure linearity only in small scale (which is doable by hand).

As a simple example let us consider a window on a screen that is shown either in full or as a small icon. The window component supports two methods (`iconify` acceptable only when the window is shown in full, and `uniconify` acceptable only when being an icon) changing the state of the window. To have a further level of complexity we assume the methods to take parameters and return results so that we cannot simply ignore invocations that occur when the window is in the wrong state. Synchronizing executions of such methods in the window component is not simple because of the state dependence. Users need detailed information about the synchronization and the window's state to avoid undesirable program behavior like no immediate reaction and some undesirable delayed reaction when pressing the "iconify" button. This information must be expressed in the interface. If we require the synchronization to be provided by users (this is, users have to ensure that method invocations occur only in appropriate window states), then we need no synchronization at all in the component, and it is probably easy for the users to ensure proper synchronization: We have to ensure linearity (this is, always only one user can invoke `iconify` or `uniconify`) and let the corresponding user know the window's (initial) state. It is quite natural to provide this required synchronization simply by having only a single "iconify" button while the window is shown in full and a single "uniconify" button while the window is an icon; we need no synchronization primitives at all to get proper synchronization.

Systems based on required synchronization

- avoid the grey-box view of components,
- clearly regulate responsibilities for synchronization,
- usually do not suffer from deadlocks,
- and are often simpler and promise higher performance.

In the author's opinion, few safe synchronization patterns together with required synchronization are almost always sufficient to achieve the desired synchronization behavior. There is no need to regard a component as a grey box for synchronization. The desire to look inside a component to determine its synchronization behavior is often just a sign of bad component design.

IV. How to Avoid the Grey Area

Availability of source code is independent of regarding a component as grey box. For example, offering customers a look at the code (or supplying open source code) can be an effective way of establishing trust into software. Code inspection need not cause a black box to become grey. The code can tells us something about the developers, company, or product line, and it is a back-up in case the software is no longer supported. However, modifications adapting code to own needs or just making use of implementation details decouples the software from further developments and is undesirable.

Sometimes we regard a component as a grey box even without looking at its code. This happens, for example, if we measure the performance (or any other non-functional property) of a system composed of components. Even if the performance is satisfactory we cannot rely on it after substituting a component with a new one. Although there exist approaches to specify non-functional properties we are currently far away from being able to specify them as part of a practical software contract. Of course, it is always possible to give some kind of guarantee based on average usages. However, it remains an open question if the one application we are concerned with fits into the class of average usages.

In this scenario we can apply a similar pattern as we did in Section II (based on the pragmatic support of usual cases): The description of the component clearly explains what the component is supposed to be used for and how to use it. Most users who correspond to this rather narrow description will most likely experience no dramatic performance decrease with future releases; they fit into the class of average usages. However, each other user is let alone. Hence, we have a pragmatic solution for the usual cases and an unpleasant position in other cases. We can improve the situation by a deep analysis of the problem and the development of appropriate tools, especially tools to improve the communication and to shift responsibility from components to their users (similarly as we did in Section III). As a simple example, we can allow users to influence the performance of components by setting parameters or providing specialized routines.

Probably this is a general pattern for a large number of aspects:

- We are able to describe the aspect in an informal interface in a very pragmatic way. The notion of pragmatic description means that component developers and users have some common understanding of the aspect and the usual solution space for corresponding problems; the description mentions only which solutions have been taken (if there are several alternatives) as well as deviations from the usual solutions. Interface specifications can be very brief (or even non-existing) especially in usual cases. Quite often such specifications are abstract in the sense that they only refer to some notion and give no details at all. For example, an interface specifies that a component behaves as a "buffer" without clarifying what this notion means. Although we can imagine many different kinds of buffers this simple word often gives us enough information to correctly use the component. Furthermore, an interface often explicitly specifies a component to be usable as replacement of another component without giving details. Such specifications relate abstractions based on common understanding. To have a common understanding is the basic idea behind the "simulation of the real world" which is a key concept in object-oriented programming. When dealing with involved aspects like synchronization we need a common understanding even

if there is no direct counterpart in the real world.

- The pragmatic approach has its limits. It works fine as long as all taken solutions are close to the usual solutions (this means that there is a common understanding) and the users need no information about other aspects than usually considered in the component description. However, it breaks down in unusual cases. For example, the pragmatic approach fails where we need static deadlock prevention because this aspect is usually not considered in the description.
- Tools and formal techniques (based on a deep problem analysis) can help in cases where the pragmatic approach fails. For example, we can apply a specific type checker to ensure deadlock-free programs. Such tools and techniques are very specific to aspects they consider and require a deep understanding of the aspects, and their use often costs much time and/or severely restricts the flexibility of the solutions. Therefore, they are often not widely used. They will be used only if the limits of the pragmatic approach begin to hurt.
- There is a common pattern behind many of these tools and techniques: Primarily they describe components in greater detail so that users get more information through interfaces. For example, type checkers to ensure deadlock-free programs impose specific program structures and make them visible in the interface. However, because of needed substitutability it is important to keep implementation details hidden. For this reason it is often no good idea to specify the complete synchronization behavior in an interface.
- To avoid unnecessary exposion of implementation details we take an additional way to strengthen connections between components and users: We move responsibilities from components to users, for example, by introducing required synchronization as we did in Section III. Then, users can take advantage of all knowledge about the environment they have. It is likely that these responsibilities rather belong to components than to their users in the usual understanding; otherwise we would not have considered them to be part of the components. Hence, the tools and techniques are mainly concerned with the support of responsibility for aspects at places where they do not naturally belong to without these tools and techniques. They give us more freedom in moving things around. They also allow us to split (otherwise not divisible) responsibility so that we can deal with each sub-aspect where we have the needed information. For example, techniques mentioned in Section III handle aliases and ensure linearity to allow us to move synchronization between components and to divide responsibility for it between all users of a component.

Historically, many developments in object-oriented and component-based programming had one common goal: Support programmers in arranging (or moving) code and data such that it becomes easy to add new parts and replace existing parts without any need to change unrelated parts. The application of tools and techniques to arrange or move further aspects (besides code and data) seems to be the natural next step. Aspect-oriented programming (as often advertised today) can probably not achieve this goal because it is not specific enough to the particularities of certain aspects. We will have to put effort into each aspect separately. Much hard work remains to be done in this area. Unfortunately, most results of such work will only be used in small scale because they are especially useful in exceptional cases where pragmatic solutions fail. Nonetheless, such work is important. Today object-oriented and component-based programming are already well-established. Progress will be made by many small improvements, not by radically new ideas as was the case years ago. Sometimes new techniques will become standard and part of the pragmatic solution.

V. Conclusions

The black-box view of components is practically important and we shall avoid to give it up even if it causes troubles. In most cases it is possible to develop components based on a common understanding of how to handle certain aspects. Interface specifications in a natural language are all we need in these cases to keep the black-box view. In more unusual situations it often helps to move responsibility for certain aspects to other components and/or to divide responsibility to several components. However, depending on the considered aspects, today we have only few tools and techniques that support us in doing so. That is an open area of research in the tradition of object-oriented and component-based programming.

References

[1] Farhad Arbab. Abstract behavior types: A foundation model for components and their composition. *Science of Computer Programming*, 55(1–3):3–52, 2005.

[2] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):269–804, September 2004.

[3] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, 1996.

[4] Naoki Kobayashi, Benjamin Pierce, and David Turner. Linearity and the pi-calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[5] Edward A. Lee and Yuhong Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16(3):210–237, August 2004.

[6] Elie Najm and Abdelkrim Nimour. A calculus of object bindings. In *Proceedings FMOODS'97*, Canterbury, United Kingdom, July 1997. Chapman & Hall.

[7] Franz Puntigam. Type specifications with processes. In *Proceedings FORTE'95*, Montreal, Canada, October 1995. IFIP WG 6.1, Chapman & Hall.

[8] Franz Puntigam. Coordination requirements expressed in types for active objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP'97*, number 1241 in Lecture Notes in Computer Science, pages 367–388, Jyväskylä, Finland, June 1997. Springer-Verlag.

[9] Franz Puntigam. *Concurrent Object-Oriented Programming with Process Types*. Der Andere Verlag, Osnabrück, Germany, 2000.

[10] Franz Puntigam. Internal and external token-based synchronization in object-oriented languages. In *Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006*, number 4228 in Lecture Notes in Computer Science, pages 251–270, Oxford, UK, September 2006. Springer-Verlag.