How dark should a component black-box be? The Reuseware Answer

Jakob Henriksson and Florian Heidenreich and Jendrik Johannes and Steffen Zschaler and Uwe Aßmann
Technische Universität Dresden
Email: {jakob.henriksson|florian.heidenreich|jendrik.johannes|steffen.zschaler|uwe.assmann}@tu-dresden.de

Short answer: Jet-black with plenty of holes, some of which are not visible to everyone.

Long answer:

I. Introduction

The Software Technology Group at TU Dresden has long experience with component-based software development and techniques. For a recent addition to the public debate, see the book entitled *Invasive Software Composition* [1]. Currently, the group is involved in projects (e.g. European NoE REWERSE, IP ModelPlex, feasiPLe etc.) addressing composition for *declarative languages*. More precisely, languages important for the development of the Semantic Web and in software modeling are addressed. Such languages include, for example, rule languages (Xcerpt, R2ML), Web query languages (XQuery), ontology languages (OWL, Notation3) and general modeling languages (MOF, UML, Ecore). To enable component-based development for such languages, the composition framework *Reuseware*[1] is being developed [3], both as a conceptual framework and as a tool.

Szyperski [4] defines a software component as follows:

> ”A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” [4]

This definition calls for components to be black-box components where no information can be inferred beyond the explicitly specified interfaces of the component. Such an approach enforces strong encapsulation and is very useful for reuse of components by third parties as these third parties need only rely on the—relatively little—information provided in the interface specifications.

For declarative languages, a pure black-box approach cannot always be taken. We currently see two reasons for this. First, not all declarative languages describe processing entities (e.g. ontology languages). As such, there is not even a notion of well-defined inputs and outputs to interface components, which is an assumption made for black-boxes. Thus, a different composition paradigm is needed to address certain declarative languages. We argue that the grey-box and fragment-based component paradigm is more suited for these languages.

[1]http://www.reuseware.org

The second reason is related to our desire to reuse existing tools for the different languages. To achieve this, all referenced components need to be composed before applying the tools. As the tools are assumed to not have a prior understanding of components (in the fragment-based sense of the word), they do not understand the need for their restricted interaction (essential for proper component encapsulation). For this reason the components needs to be *opened up* such that their interactions can *statically* be ensured in the composition result. Again, this is not an idea supported by black-box component environments, but possible in composition systems based on grey-box approaches.

For some general-purpose languages (e.g. C++, C#, Java), components can be described on different abstraction-levels—either as run-time entities or as static source-code snippets (as done for aspects in Java). But for most languages used in software modeling or on the Semantic Web we do not have much choice. Thus, for the languages in these important fields, components necessarily consist of source artifacts—snippets of descriptions—which can play roles in more complex, complete, coherent and usable descriptions or declarative programs. However, one should take care not to meddle with one of the most powerful notions in component-based development: *the power of abstraction*. Thus, it is of utmost importance to properly encapsulate components—hide their details—and to access them via *well-defined composition interfaces*. Here, again, we are in line with Szyperski’s definition from above: All access to the component should occur through well-defined interfaces, all dependencies should be explicit. We shall return to the issue of interfaces shortly.

Our work has its roots in Invasive Software Composition (ISC) [1]. ISC takes a grey-box composition approach where components, or *fragments*, are static source-code entities with well-defined interfaces using the notion of *hooks*. A hook is a location in a component which may effectively be replaced by another component, thus, composed. As such, the hooks of a component define its interface. The replacement of a hook with some existing component constitutes the basic composition technique of ISC. One of the conclusions from work on ISC was the identification of a set of *primitive composition operators* implementing the described composition technique. Two of the identified primitive composition operators are *bind* and *extend*, where *bind* replaces a hook with some component once and *extend* possibly multiple times. The composition technique and operators defined for ISC are very general and applicable to many different languages and situations. It should be noted that ISC is able to realize existing composition approaches and techniques, such as aspect-orientation, view-based programming, hyperspaces etc.

Our experience with ISC and component-based development for declarative languages has refined our requirements for composition interfaces. Most importantly, we argue that components for declarative languages shall indeed be grey-boxes, but with tailored and *refined composition interfaces* to answer the call from language-specific needs and language-specifically developed composition operators.

II. Dedicated composition systems and environments

Many domain-specific languages in software modeling and on the Semantic Web do not provide sufficient constructs for defining reusable entities—components. Many languages do have some form of abstraction and reuse idea, but it is often limited, inflexible and most of all—fixed. For example, rule languages on the Semantic Web often allow rule chaining; the possibility of sequencing rules in different chains of computations. As such, the notion of the *rule* is the level of reuse made possible by the language itself. No other entities are reusable; there is no other level of *abstraction*. That is, the set of abstractions provided by the language is fixed. Thus, once the language has been designed and its relevant tools have been developed, the language as such is very inflexible to be changed for new abstractions. It should be noted that the expressiveness provided by languages is usually adequate, since the languages certainly were developed against use-cases and specific requirements. We exploit the fact that appropriate expressiveness is provided for by reusing existing tools developed for the different languages when processing the composition results. However, we will address the issue that a flexible level of abstraction is not to be found.

We argue that instead of redesigning an individual language, additional levels of abstraction, and thus reuse, can be provided via *composition*. We propose to layer a *light-weight dedicated composition system* (LWDCS)[2] on top of a targeted *core language* and its tools (see Figure 1) to provide richer abstractions and allow programmers to think about their programs in new and interesting ways. The composition system is dedicated because it addresses issues for a single targeted language, and light-weight since once developed and deployed it is assumed to be operable without its users directly being aware of it. The LWDCS injects a core language with additional constructs, giving users the possibility to define reusable components and to compose them in desired ways, all tailored for the need at hand. The LWDCS is responsible for interpreting the newly introduced constructs and for composing specified components into programs or descriptions of the core language. Thus, the existing and already developed tools are reused and the semantics of the core language is appropriately retained (as mentioned, we deem the core languages already capable of expressing what they should). Furthermore, a LWDCS is type-safe, ensuring that resulting descriptions (programs) are (syntactically) valid with respect to the underlying core language. Ensuring semantically correct results is also possible, but not further discussed here.

[2]Pronounced *low-deeze*; pl. LWDCSs (*low-deezes*).

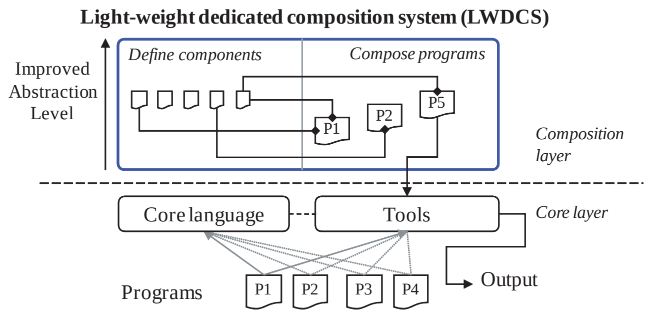

Fig. 1. A light-weight composition system can be layered on top of a language and its associated tools to improve and add the level of abstraction provided by the underlying language itself.

A composition system can be seen as a triple consisting of: a *component model*, a *composition language*, and a *composition technique* [1]. We argue that a LWDCS should be constructed as a refinement of a generic composition system, where the triple comprising the system is specialized for the task at hand—indeed tailored (see Figure 2). As can be seen from Figure 2, the dedicated composition language is adapted for the specialized task and refined from a more general-purpose language. Furthermore, the dedicated component model references an *upper-level component model* where general composition system concepts are modeled. Finally, instead of including a general composition technique and generic operators in the dedicated composition system, it is shipped with a set of predefined, specialized, composition operators.

The most important detail to notice in Figure 2 in order to answer the question about the desired darkness of components is the relationship between the set of dedicated operators and the dedicated component model. The detail to notice is that the component model, which effectively determines the darkness of the components used in a composition system, heavily depends on the specific composition operators included in the LWDCS. We shall return to this issue with a more detailed discussion in Section IV. First, before describing our notion of refined composition interfaces, we will briefly describe how a dedicated composition system may be semi-automatically generated. In particular, how a component model may be derived from a core language.

III. Generated component models

We intend to build upon the language-independent composition technique introduced in ISC. This means that components may contain hooks that can be replaced by other components. We refer to these positions in components by the more general term *variation points*. Thus, the variation points declared in components define the components' interfaces. From these

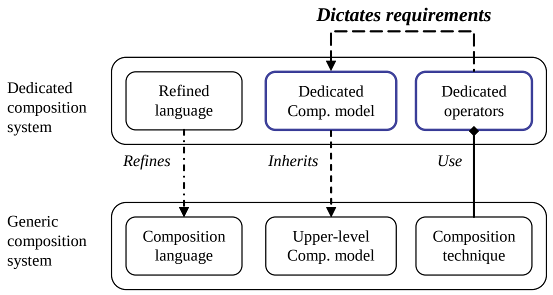

Fig. 2. A dedicated composition system is a specialization of a generic composition system where the tailored composition operators addressing particular issues in a declarative language dictate the form and detail of the dedicated component model and thus, the components' interfaces.

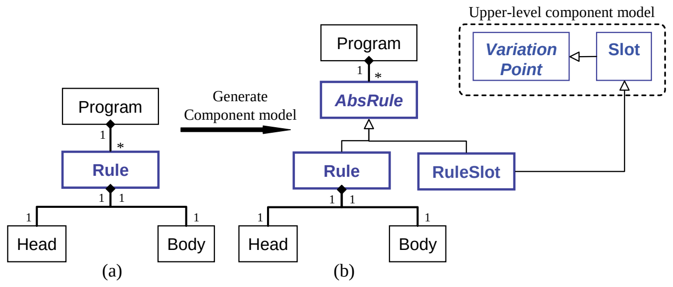

Fig. 3. The abstract syntax description of a simple rule language on the left-hand-side can be extended into the abstract syntax description on the right-hand-side to allow programs to be underspecified with unknown rules, to be composed into the program at a later point.

requirements, one can automatically generate a component model from a *core language* description (grammar or model based) [3]. Figure 3 (a) shows a simple (partial) model of some rule language. The model states that programs of the rule language consist of one or more rules, which in turn are composed from a head and a body (what they are in detail has been left out here).

Assume the simple case that we want to be able to write rule programs in our rule language where certain rules are not explicitly given, but left *unspecified* (the program as a whole is *underspecified*). The underspecified program is a component with an interface, given by the variation points programmed into the component. In order to be able to declare variation points we inject the core language with constructs for this purpose. This modification can be seen in Figure 3 (b). The concept of the rule has here been made *variable*. Rule programs may now consist of normal rules (concept `Rule` in Figure 3 (b)) and rule *slots* (concept `RuleSlot` in Figure 3 (b)). The abstract super-concept `AbsRule` is introduced to represent this choice. A `Slot`, as can be seen in the upper-level component model is a kind of variation point. Here it is assumed that a slot has some concrete syntax such that variation points can explicitly be declared in components. The model in Figure 3 (b) properly describes what our simple rule components look like and defines how the composition technique is allowed to modify the components (by replacing variation points with other suitable components). The only access points to the components are the declared variation points (expressed using slots), everything else is properly encapsulated. As such, the derived language model in Figure 3 (b), along with references to the upper-level component model, is the component model for our simple components. It is possible to automate such transformations.

IV. Refined and Controlled Composition Interfaces

It is useful to be able to reuse common composition techniques across different dedicated composition systems targeting different languages (see relationship between the generic composition technique and the dedicated operators in Figure 2). This is beneficial since the basic technology does not have to be reimplemented for each composition system and targeted language. However, in order to support and realize the appropriate kinds of reuse abstractions, different languages require special-purpose composition operators. Thus, it is desirable that the dedicated composition operators are defined in terms of, that is, reuse, the primitive composition operators implementing the general composition technique (see Figure 4). Furthermore, if a specific reuse abstraction concept is desired for different languages, using the same basic and underlying composition technique is again advantageous for practical reasons. Examples of reuse concepts not limited to a specific language are modules and aspects (disregarding the exact detail of their purpose and how they look in the specific languages).

As our work extends that of ISC, which provides a very general composition technique, we aim at reusing this technique and its primitive composition operators for creating LWDCSs. While a single operation of a primitive operator only can describe a *low-level* composition step, a properly defined sequence of such primitive composition operators can achieve a more advanced desired effect on a set of fragments—a *high-level* composition step. If such a high-level sequence is found useful for different fragments, one would like to be able to encapsulate the sequence as a single reusable atomic composition operator. We call such an operator a *complex composition operator*. Thus, a complex composition operator is able to encapsulate and realize a *non-obvious reuse abstraction*. This notion gives us the possibility to develop language-tailored composition operators to be included in LWDCSs.

One thing to notice about complex composition operators is that they may not only encapsulate a sequence of primitive operators, but also components. That is, some composition operators may require internal components, needed for the realization of the (abstraction) construct they are implementing. Such components are not visible, or indeed known, to programs using the operators; they are completely encapsulated within the operator definitions.

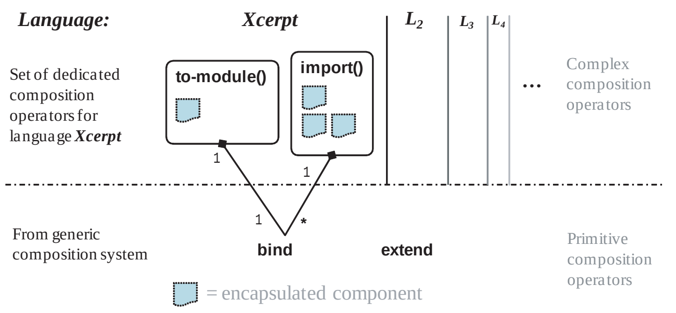
Language:
Xcerpt
L_2
L_3
L_4
Set of dedicated composition operators for language Xcerpt
to-module()
import()
Complex composition operators
From generic composition system
bind
extend
Primitive composition operators
= encapsulated component

Fig. 4. Complex and dedicated composition operators for a dedicated composition system are defined in terms of general composition operators and techniques from invasive software composition. The operators to-module() and import() address specific issues in the core language Xcerpt.

A. *Example – Modules for the Web Query Language Xcerpt*

We have practical experience with targeting and building a LWDCS for the Web query language Xcerpt [2]. Xcerpt is a powerful rule-based language following the logic programming paradigm for querying different kinds of semi-structured data. An Xcerpt program consists of a finite set of rules. What differentiates Xcerpt from some other well-known Web query languages, e.g. XQuery, is that Xcerpt programs (i.e. their rules) have a clear separation between data query parts and data construct parts. As in other logic programming languages, Xcerpt rules consist of a head and a body. The body of a rule can match existing data, resulting in variable bindings. The variable bindings produced by successful matching of the body of a rule can then be applied to the head of the rule in order to derive new data. As such, the rule bodies represent the queries and the rule heads the construct parts.

An identified and desired (but so far lacking) abstraction for Xcerpt was the notion of Xcerpt *modules* (much in the style of other logic programming systems). An Xcerpt module consists of a set of rules, which can be imported and reused in different programs. A good example of a useful module is a set of rules able to perform simple reasoning on ontology documents (e.g. OWL). An example of such reasoning is to derive implicit subclass-of relationships from explicitly declared class-hierarchies.

As a module consists of a set of rules, they should all be included in the importing program at composition-time, such that they are available to the Xcerpt interpreter when the composition result is executed. However, properly realizing the module system is more subtle and complicated than just executing the merger of different rule-sets. Since a module from our point of view is a *component*, certain parts of the module should be able to be encapsulated. From a usage perspective, a module can almost be seen as a black-box with an *input* rule and an *output* rule. The input rule is passed data to process (possibly constructing intermediate results for rules encapsulated in the module) and eventually data to be used by the importing program is constructed by the output rule. At the level of composition, however, we cannot consider modules as black-boxes. In order to allow modules to be encapsulated, one must ensure that inappropriate rule dependencies do not occur when programs and modules are merged before being executed. That is, programs should only have access to certain rules in imported modules, and vice versa. This encapsulation can be realized by transforming the heads and bodies of the rules of the imported module in appropriate ways. The details are left out since it is not relevant exactly how this is realized. What is clear is this: If rules in modules are to be transformed in some way at composition time, the way they are transformed, and thus *accessed by composition operators*, must properly be reflected in the relevant component model.

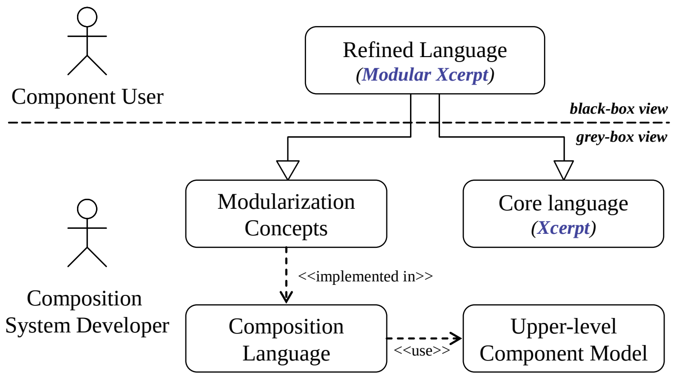

Fig. 5. The *Component User* and the *Composition System Developer* are working with different parts of the composition system and have different views on the system.

To understand the requirements of the component model, it is helpful to distinguish two different roles—or view-points—with respect to a LWDCS (containing the component model). Figure 5 illustrates these different view-points.

1) *Component user role* Users of the above-described Xcerpt module system only want to be able to define (encapsulated) modules and import already existing ones. The constructs for doing so should appear to be first-class constructs of the core language rather than added composition operators. As such, one should not require the module programmers and users to define precisely how and where their modules must be transformed during composition. That is, they should not be required to describe how the underlying encapsulated composition operators realize the module system and, thus, access the modules (components).
2) *Composition system developer role* The composition system seen from the view of the system developer is however much different. The system developer cannot assume the black-box view of the users, but rather a *grey-box view* in line with our arguments of this necessity for declarative languages. The system developer must develop the complex composition operators responsible for realizing the module system and provide an appropriate component model reflecting the intended interfaces of the components. We recall from Section I

the argument for the need to ensure proper component interactions statically. Hence, the components *do* need to be opened up in the deployment of the module system and this responsibility lies on the composition system developer.

To support these different roles—considered attractive for the users—the development of the specific composition operators and the composition system as a whole dictate requirements for the component model. We therefore need to transform the core language model in a slightly different way (Figure 6) as to what was done in Figure 3. As can be seen in Figure 6 (b), in place of the head construct we introduce a head variation point (`HeadVP`), which forms part of the interface of components adhering to the component model. At the variation point, either the original head construct can directly be programmed in its place (as a *default value* for the variation point), or a concrete variation point (slot) can be used. This means that regardless of whether the head of some rule consists of a core language head construct (`Head`) or is left unspecified (using the introduced `HeadSlot` construct), the component model describes it as accessible, as part of the component's interface.

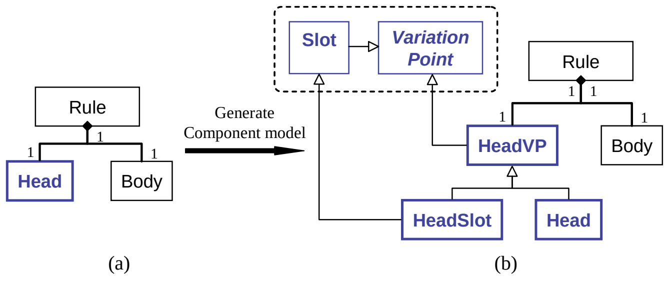

Fig. 6. The original construct (here `Head`) must at composition time be accessible to certain composition operators. Thus, the construct must be part of the interface of the component, which is realized by making the variation point (`HeadVP`) a super-class of Head.

The same kind of transformation can be done for the body of rules. Again, it should be stressed that module programmers working against the component model in Figure 6 (b) do not have to express via some special mark-up that the rule heads are part of the interface. They can write rules as they normally would, but still expect the programmed modules to be usable in the LWDCS realizing the module system.

So, the module system is realized by a set of dedicated composition operators (transforming rules), along with a dedicated component model adjusted to the needs of the operators. Along with a composition language (not discussed here) we can create a LWDCS for Xcerpt realizing *additional abstractions*, in this case the possibility of authoring encapsulated modules and using them in Xcerpt programs.

The critical notion is the following: due to the encapsulation of the complex composition operators, we find it necessary to refine our notion of composition interfaces. This is a consequence from the fact that was remarked upon earlier: the set of dedicated composition operators included in a targeted composition system dictates the form of the associated component model, that is, how components look and interact (see Figure 2).

In a similar fashion one can identify needed abstractions for other declarative languages. Instead of re-designing the language and its tools one can realize the abstraction by implementing the necessary additional constructs as complex composition operators in a LWDCS and generate an appropriate and tailored component model with the *appropriate shade of darkness*.

V. Conclusion

In this paper we have presented the Reuseware approach to Invasive Software Composition in an attempt to answer the question "How dark should a component black-box be?" for components in declarative languages or in situations where composition occurs on the source-code level.

Our short answer has been "Jet-black with plenty of holes, some of which are not visible to everyone." In the long answer we showed that this means that we require encapsulated components where composition can only occur in well-defined places—hence they are "jet-black". At the same time, however, component developers and users should not have to worry about all the details of the composition interface relating to encapsulated composition operators. Rather, this part of the interface should be described in the relevant component model and taken advantage of by the complex composition operators available in the dedicated composition system for which the components have been written. Hence, components "have plenty of holes", but they are "not visible to everyone". More specifically, some parts are visible to developers of dedicated *composition systems* (LWDCS), while *component* developers and users only have to care about the part of the interface relevant to them.

Acknowledgment

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net), as well as through the 6th Framework Programme project Modelplex contract number 034081 (cf. http://www.modelplex.org) and by the German Ministry of Education and Research (BMBF) within the project feasiPLe (cf. http://www.feasiple.de).

References

[1] U. Aßmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
[2] F. Bry and S. Schaffert. The XML query language Xcerpt: Design principles, examples, and semantics. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 295–310. Springer-Verlag, London, UK, 2003.
[3] J. Henriksson, J. Johannes, S. Zschaler, and U. Aßmann. Reuseware – adding modularity to your language of choice. *Proc. of TOOLS EUROPE 2007: Special Issue of the Journal of Object Technology (to appear)*, 2007.
[4] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. Addison-Wesley Publishing Company, second edition, 2002.