Liberating Composition from Language Dictatorship

Lodewijk Bergmans Christoph Bockisch Mehmet Akşit

{bergmans, c.m.bockisch, aksit} @ ewi.utwente.nl
Software Engineering group, Dept. of Electrical Engineering, Mathematics & Computer Science
University of Twente, The Netherlands
trese.ewi.utwente.nl

ABSTRACT

Historically, programming languages have been—although benevolent—dictators: fixing a lot of semantics into built-in language constructs. Over the years, (some) programming languages have freed the programmers from restrictions to use only built-in libraries, built-in data types, or built-in type checking rules. Even though, arguably, such freedom could lead to anarchy, or people shooting themselves in the foot [12], the contrary tends to be the case: a language that does not allow for extensibility, is depriving software engineers from the ability to construct proper abstractions and to structure software in the most optimal way. Instead, the software becomes less structured and maintainable than would be possible if the software engineer could express the behavior of the program with the most appropriate abstractions. The new idea proposed by this paper is to move composition from built-in language constructs to programmable, first-class abstractions in the language. As an emerging result, we present the *Co-op* concept of a language, which shows that it is possible with a relatively simple model to express a wide range of compositions as first-class concepts.

1. MOTIVATION

The software engineering discipline faces many challenges; one of the important challenges is to cope with the changes that need to be incorporated into software systems during their lifetime. A particular difficulty is that small, local, changes in the requirements of a system, often lead to non-local changes in the software. This is caused by the fact that the structure of the implementation tends to be significantly different from the structure of the problem domain.

A second software engineering challenge is managing the complexity of software [10]. We are building increasingly large and complex software systems, enabled by steady improvements in the software engineering discipline. Such systems encompass a substantial amount of *inherent complexity*; partially inherent in the problem domain, and partially in the solution domain. But the *realization* of these systems also introduces a large amount of *accidental complexity* [3], while going from the conceptual solution to a realization model.

A key argument, as coined e.g. by Brooks [3], adopted in this paper is that the limited ability of realization models to accurately represent the concepts and their interdependencies in a conceptual solution is the main cause of *accidental complexity*. As a result, the complexity of our realizations is typically substantially larger than the complexity of the conceptual solution.

In software engineering, a key strategy for dealing with change and managing complexity is "divide and conquer": achieve *separation of concerns* [4] by dividing a solution into building blocks, and delivering working systems by expressing proper compositions of these building blocks. The history of programming and design shows a steady movement towards supporting higher-level abstractions (including building blocks) and more advanced ways of expressing such compositions. For example object-oriented and aspect-oriented programming are largely motivated by the need for improved modularity and separation of concerns. Recent trends in software engineering, such as Model-Driven Engineering (MDE) and Domain-Specific Languages (DSLs), all aim at offering the 'right' abstraction level for expressing particular types of problems: to this extent, they offer dedicated (possibly graphical) syntax, dedicated data types and operators, or dedicated abstractions and corresponding composition techniques, to achieve better modularity and separation of concerns for specific domains.

The message of this paper is that there is substantial benefit in offering languages where abstractions and composition techniques are not hard-wired; rather they should be easy to introduce on demand by software engineers as new types of compositions are identified. In the remaining sections we will first discuss the notion of composition mechanisms, their role in existing languages, and which problems this causes. Then we define the scope and assumptions of our proposed solution, also contrasting our choices to other areas of related work. We then present our proposed approach, and show evidence that this is indeed feasible, we end with a brief discussion on implementation strategies and a conclusion. We mention related work interspersed with this paper, especially in sections 4 and 7, more detailed discussions on related work can be found in our previous publication [7].

2. COMPOSITION MECHANISMS

Programming languages[1] offer explicit and implicit means

[1]Whenever we talk about languages in this paper, this

to express composition of abstractions, which means that the characteristics of a new abstraction are expressed in terms of the characteristics of one or more existing abstractions (and possibly additional specifications). For example, *function composition* in functional programming, such as $f(g())$, expresses that—the behavior of—f and g are composed by using the result of g as an argument of f. Similarly, *function invocation*, like the call to q in $p()\{...q()...\}$ is used to define the behavior of a procedure p—partially—in terms of the behavior of q. In this sense, the function invocation expresses that the behavior of p is a composition of the behavior q with other behavior specified for p. Another example is *object aggregation* (also referred to as *object composition*): this expresses how a new object structure is defined as e.g. the union of other object structures.

In typical object-oriented languages the following composition mechanisms are available: behavior composition through message passing (cf. function invocation), object aggregation, and inheritance [14]. Compositions can be binary (such as function invocation and single inheritance), or n-ary, such as multiple inheritance, or pointcut-advice composition in AOP. Note that in many cases, as in the function invocation example above, the composition specification is integrated with the specification of one of the two composed entities. For example, when a class P specifies that it inherits from class Q, this implies that the specifications of P and Q are composed into a new whole, with the name P.

Composition is not necessarily implemented directly by language constructs; for example, many design patterns, such as those in [5], describe how a set of objects interact to create a coherent new behavior: a composition of the participating objects. This composition is typically realized by a set of code snippets distributed over the participating objects; this must be re-implemented for every design pattern instantiation, it affects the traceability of the patterns in the code, and is hard to maintain.

3. PROBLEM ANALYSIS

New composition mechanism are introduced all the time. For example, in [14], a taxonomy for inheritance mechanisms is described, from which—in theory—hundreds of variants for inheritance can be derived. More recently, many different proposals have been made for aspect-oriented languages and models: a survey report [2] contains 45 different proposals, where in most cases the composition techniques are unique. Similarly, there is an indefinite amount of design patterns that essentially express a composition of several objects.

In general, it can be safely assumed that for each of these proposed composition mechanisms, there is a sound argumentation why that mechanism works better—at least for a certain class of applications, or within a certain context. The fundamental reason is that the application of each composition mechanism involves a certain trade-off, which makes it particularly suitable in certain contexts, but less so in others. Hence, a language that dictates a fixed set of composition techniques, with no opportunity to extend that set, will inherently restrain the software engineer: He or she is not able to choose the most appropriate composition mechanism, with the best possible trade-offs, and the most natural mapping from a conceptual solution to the implementation.

should be interpreted broadly to any means of specifying machine-executable behavior.

Among the negative results caused by such dictatorship[2] are:

lack of traceability since the intended composition has to be replaced with an alternative, typically involving additional 'glue code'.

lack of maintainability because the glue code is usually specific to the context, and has to be added in multiple locations, where it is also tangled with the functionality.

increased accidental complexity because straightforward compositions at the conceptual level have to be realized by more complicated code, often introducing additional dependencies.

4. SCOPE AND ASSUMPTIONS

By now the strategy that we propose in this paper may be obvious; we want to free composition from languages where it is limited to a few, fixed composition mechanisms, and instead propose language facilities where the appropriate mechanisms can be defined, applied, and reused. Before explaining this approach in more detail, we first discuss the scope of our solution approach and some of the assumptions we make. This discussion should also help to distinguish our work from various areas of related work.

Although not essential to the general idea, in the remainder of this paper we focus on object-based languages, which support encapsulation and message-invocation. In this context, composition refers to *composition of the behavior of objects*.

Our approach is *not* based on full reflection: although that is a very powerful technique, reflection based solutions tend to be very hard to organize and manage, hence are not suitable from a scalability point-of-view. In particular, building fully reflective solutions that are also composable and extensible, requires a substantial amount of effort and discipline for the involved software engineers. However, we do propose to adopt limited reflection (where only specific elements of a language are exposed for reflection). We would like to point out that our approach can well be implemented, *using* reflection, as a specific Meta-Object protocol (MOP) [9].

We also aim for solutions that are not transformation-based, such as most Model-Driven Engineering (MDE) tools and external Domain-Specific Languages (DSLs). Although, implementing composition operators as transformations is in principle a possibility, this approach suffers from several issues, in particular if multiple composition operators are to be integrated within a larger solution: (1) it is hard to exchange data between the model (or DSL) world and the rest of the system, (2) it is hard to add additional DSLs without running into all types of integration issues, and (3) require extra tool support to develop at the model level.

5. GENERAL APPROACH

In the general approach of *first-class composition operators* (Co-op) composition operators are objects which *operate* on compositions; we assume that the language provides built-in primitive composition mechanisms, for instance sending messages between objects. Composition operators can (cooperatively) influence the behavior of a composition. As

[2]There are in fact also positive sides; e.g. less choice makes both decisions and comprehension easier—albeit at substantial costs.

we will discuss, this approach is sufficiently powerful to express common composition mechanisms like inheritance, delegation or design patterns. Key characteristics of a language adopting the Co-op approach are the following.

- It does not fix its composition operators: although it provides one or more composition operators that are always available.
- Composition operators can be defined and added to a system, as is one of the key motivations for this approach.
- Composition operators are first-class entities in the language: we believe this is an essential feature for a scalable approach (see also composability 3 below). This also means that the application of composition operators can be as simple as object instantiation, and composition operators can be managed as regular libraries.
- Composability (1): it must be possible to freely apply multiple composition operators within the same application.
- Composability (2): it must be possible to combine multiple composition operators in the same context, i.e. apply them to the same objects. However, the semantics of those operators may impose inherent limitations on the meaningfulness of such combinations.
- Composability (3): composition operators can be used in the definition of other composition operators (save infinite recursion).
- Inherent dependencies or constraints among composition operators should be expressible.

With the above characteristics we aim to avoid making any platform and implementation-specific choices. This necessarily keeps the description very abstract; next we discuss a concrete realization of these ideas.

6. EVIDENCE: CO-OP/I LANGUAGE

As a proof-of-concept, we have partially realized these ideas in the programming language *Co-op/I*[3] [7]. We have defined an operational semantics for this language and implemented an interpreter for it in Java. Co-op/I is an object-oriented, dynamically typed language with only *one built-in* composition mechanism, which is a *sending a message*. Other composition mechanisms are not manifested in the language's syntax, for instance, no keyword exists for declaring class inheritance. The semantics of where a message is delivered is also not fixed but determined by the first-class composition operators in Co-op/I: Composition is performed at message sends, by rewriting the message's properties.

Message sends are, per se, undirected, i.e., when the program sends a message, it specifies property values like the message name or initial argument values; since all argument values may be rewritten by composition operators, the first argument value is not necessarily the receiver of the message. Objects in Co-op/I are first-class values with fields and methods just like in conventional OO languages. But in addition, any Co-op/I object can also be a composition operator; this is, objects can be activated and then participate in the rewriting of sent messages. Figure 1 shows the composition infrastructure schematically. In *Co-op*, we have chosen to use the terms *module* and *module instance*

[3]`http://co-op.sf.net`

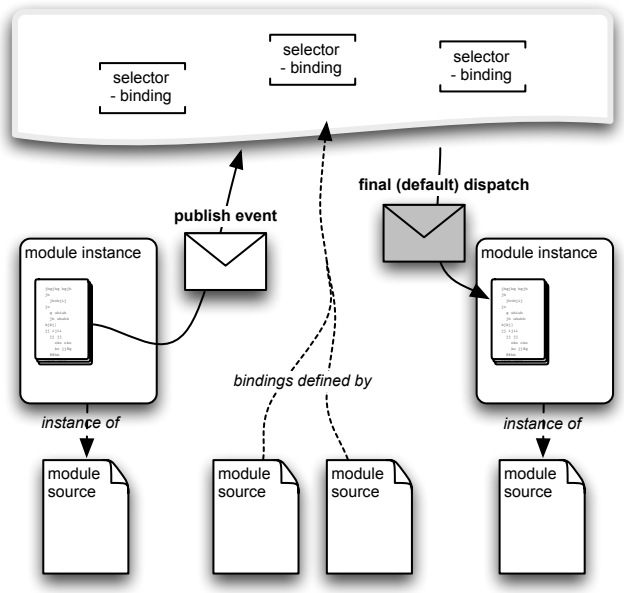

Figure 1: Overview: Composition in *Co-op*

to refer to concepts that are comparable to (object-based) classes and objects, with the distinction that they may also specify composition operators. In the middle left, a *module instance* (object) publishes an *event*. This event is evaluated by the active set of *selector bindings*. These bindings are defined and "activated" by *Co-op modules*. Finally, the evaluation of the event typically leads to the invocation of one (or more) operations, on an instance of the target object. A key foundation of our approach is that (a) all possible behavior involves event dispatch, and (b) hence the manipulation of dispatch can be used to express a truly wide range of behavioral compositions.

To define the semantics of a composition operator, in Co-op/I, classes can define three additional members.

1. *Conditions* define boolean predicates that refer to message properties and to fields or methods defined in the class. These predicates are assumed to be side-effect free; in Co-op/I, they are comparable to regular boolean expressions (closures)[4].
2. *Bindings* specify the condition under which they participate in the rewriting of a message (through 'event selectors') and assign new values to message properties (through 'action selectors') when they are applicable. New values can be defined as expressions referring to message properties, and fields and methods of the declaring class.
3. *Constraints* define relations between composition operators with bindings that are applicable at the same message send. Relations can be an order of evaluation or the prevention of evaluating one operator.

The elements we have described above can be used together to define how events are eventually bound to concrete operations. Figure 2 schematically illustrates this. We briefly list the flow of events in a number of steps (corresponding to the numbers in Figure 2):

1. On the left hand side we see that execution of operations may lead to publishing of an event.

[4]But in a new version of the language, dedicated syntax allows for better analyzability and (hence) optimizations.

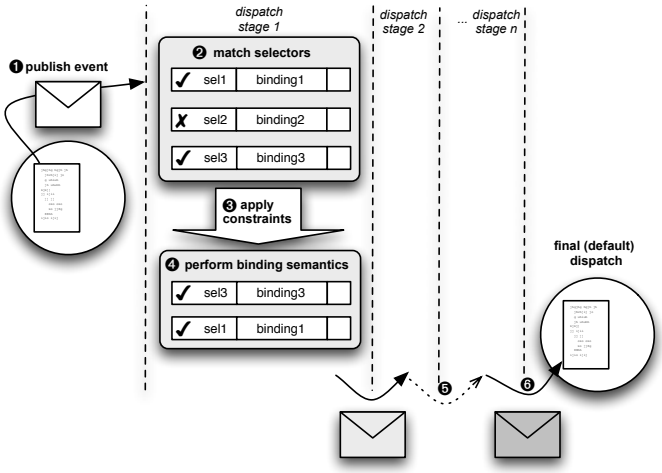

Figure 2: An overview of the event dispatching process

2. It is determined which of all event selectors match with the event. This potentially enables a set of bindings that refer to those selectors.
3. Applying the constraints that have been defined between the various bindings may further reduce the applicable bindings (e.g., because bindings may exclude each other), and determines a (partial) ordering among the bindings.
4. The resulting set of bindings is evaluated; this consists of binding of values from the received event to the resulting event, and the evaluation of the associated action selectors.
5. This process may repeat itself multiple times (i.e. there can be multiple dispatch stages), as long as there are matching bindings.
6. Actual execution of (base-level) operations will occur when a "default binding" is executed.

The default binding is the only built-in composition operator. It does not rewrite the message but delivers the message to a method corresponding to the message properties, i.e., like a simple *function application* composition operator. As it is primitive, an implementation of this composition operator cannot be provided in the Co-op/I language; but an instance of this composition operator exists with which other composition-operators can interact.

In this prototype language we have been able to realize many different composition operators such as aspects, delegation, traits, and different forms of inheritance [7], and several design patterns, including the Memoization, Observer and State patterns [8]. Of the different alternative semantics for inheritance presented in [14], we have implemented all, except for those with ordered multiple inheritance, due to a limitation in the current language implementation, not in the concept. In our examples, we have been able to reuse the implementation of lower-level composition operators by combining them to more complex ones, again by means of Co-op/I composition operators.

While Co-op demonstrates that it is in principle possible to make a language sufficiently powerful to define at least a wide range of composition mechanisms, some research questions remain that must be answered to make this approach ready for practical use. For example: the language must be made more complete to support, e.g., ordered multiple inheritance; ways must be researched to optimize the execution performance of Co-op/I programs; and inter-operation with other programming languages should be supported.

Optimizations are necessary because the dynamic evaluation of composition operators adds an overhead to each message send in the program. To compensate, the current Co-op prototype already aims at a high degree of declarativity in the definition of conditions, bindings and constraints. Thus, their effects on a message can be partially evaluated before runtime. However, it is still an open topic to actually implement and evaluate such optimizations for Co-op/I.

Language interoperability is important to make use of the large body of existing libraries. This is supported for mainstream languages, e.g., the Java Native Interface allows to use C/C++ libraries from Java and vice versa. To support this, ways for communicating between the dynamically typed Co-op world and the statically typed of Java or C++ must be researched. A bridge to another dynamically typed language, e.g. Smalltalk may be simpler, but still not trivial: Smalltalk has a notion of type hierarchies and assignment compatibility while Co-op/I composes the behavior.

7. IMPLEMENTATION STRATEGIES

The previous section has shown a realization of our ideas based on a general-purpose programming language that allows to define message rewriting defined in the same language as the actual program. Facilities for message rewriting can also be offered in other ways, most obviously through a reflective language, as a meta-object protocol (MOP). As already explained in Section 4, the Co-op *model* and a MOP approach are not contradictory, and a MOP approach is one way to implement the Co-op ideas.

However, the advantage of defining a language, such as Co-op/I, is that it can offer the programmer dedicated abstractions for e.g., the conditions, bindings and constraints. This can make the specification more declarative which makes composition operators, themselves, more composable. Because of the declarativeness, the execution environment can better reason about the composition operators and control their application. In addition, it becomes easier to provide tailored IDE support, such as a dedicated debugger. In contrast, in a traditional MOP, the execution environment cannot make specific assumptions about the structure or behavior of metaobjects.

There exists a variety of OO programming languages (e.g. SELF [15], Smalltalk[6]) that aim at offering a very simple core language on which to build complex abstractions. However, to the best of our knowledge, these languages either do not offer the ability to define tailored composition semantics, or do so by opening up the language in a generic way through reflection, as discussed in the previous paragraph.

Another conceivable approach is to implement composition operators by means of code transformations, which, as we already mentioned in Section 4, has its difficulties. This means, for every composition operator to be added to a language, the language is extended, e.g., with new keywords, and a transformation is implemented. This transforms the code (i.c. the added composition operator) written in the extended language, to behavioral equivalent code in the target language. Implementing composition operators as code transformations has two disadvantages: First, the transformation is usually not written at the same level as the appli-

cation program. This is, the compiler of the used language has to be modified or a preprocessor is implemented. Second, transformations are difficult to compose freely, because each transformation is dependent on the structure and semantics of the target program. However, the effects of other transformations–of other composition operators–has to be taken into account as well, which creates complex interdependencies between the transformations for all individual composition operators. Implementing transformations in terms of a compile-time MOP [11] improves the first shortcoming because transformations and application code can be developed in the same program. It is our conviction that implementing composition operators as individual transformations with the current state-of-the-art target languages is not a feasible approach.

8. CONCLUSION

In this paper, we have argued that fixing the composition mechanisms offered by a language severely limits developers. It forces them to make compromises when choosing a language to implement the requirements which they have decomposed in a way that best fits their business domain. Thus, inevitably programs gain *accidental complexity*. In order to reduce this, we propose the Co-op approach to integrate facilities into programming languages that allow developers to implement their own composition mechanisms according to their domain's abstractions. They should be implemented as composition operators which are first class objects in the programming language such that they can be composed, themselves, by means of custom composition operators.

By discussing a prototype language, Co-op/I, that allows normal application objects to act as composition operators by means of message rewriting, we have shown that realizing these ideas is feasible. Since message passing between objects is able to fully cover the objects' behavior in object-oriented languages, message rewriting is a powerful mechanism to arbitrarily control the composition of behavior. Making the definition of message rewritings declarative as in Co-op/I simplifies the composition of independently developed composition operators as the execution environment can reason about their joint application.

But also other implementations are conceivable. The message of this paper is not how languages with more flexible composition operators should be implemented. The message is that dedicated abstractions that fit the problem domain are needed; since programs in general combine multiple problem domains in unforeseeable ways, programming languages must not hard-wire abstractions and composition mechanisms, but must allow developers to tailor them to their needs. As part of future work, we will investigate more explicit interfaces (e.g. [13]), and type checking (e.g. [1]).

The increasing importance of domain-specific languages and, e.g., support for embedded DSLs, show the desire for expressing different program parts using different, most appropriate, abstractions. While the Co-op approach of allowing programmers to define their own composition operators as first-class objects is quite powerful, it also bears more complexity. This complexity is not necessarily exposed to the application programmer, though, because the implementation of composition operators can be provided by a few, well-trained experts in the form of libraries. The application programmer simply uses such a library, instantiating composition operators as needed, like regular object instances. Certainly its concrete implementation and the implementation of supporting tools like debuggers will also have a great impact on the complexity exposed to or hidden from application programmers. Nevertheless, it is subject to future studies to research the impact of our proposed programming model in practice.

9. REFERENCES

[1] G. Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, 2004.

[2] J. Brichau, M. Haupt, N. Leidenfrost, A. Rashid, L. Bergmans, et al. Report describing survey of aspect languages and models. Technical Report AOSD-Europe Deliverable D12, AOSD-Europe-VUB-01, Vrije Universiteit Brussel, 17 May 2005 2005.

[3] F. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987.

[4] E. W. Dijkstra. *A discipline of programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, 1994.

[6] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.

[7] W. K. Havinga, L. M. J. Bergmans, and M. Akşit. A model for composable composition operators: Expressing object and aspect compositions with first-class operators. In *Proceedings of the 9th international conference on Aspect-Oriented Software Development, Rennes and Saint-Malo, France*, pages 145–156, New York, March 2010. ACM.

[8] W. K. Havinga, C. M. Bockisch, and L. M. J. Bergmans. A case for custom, composable composition operators. In *Proceedings of the 1st International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines, Rennes, France*, volume 564 of *Workshop Proceedings*, pages 45–50. CEUR-WS, March 2010.

[9] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[10] W. Royce. Improving software economics-top 10 principles of achieving agility at scale. White paper, IBM Rational, May 2009.

[11] T. Sheard and S. P. Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37:60–75, December 2002.

[12] B. Stroustrup. Bjarne stroustrup's faq, nov. 2010.

[13] K. J. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with xpis. *TOSEM*, 2009.

[14] A. Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

[15] D. Ungar and R. B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22:227–242, December 1987.