Universal Composition—an optimal scheme for structuring (software) systems

Henrik Gedenryd, Tullgatan 8, SE–223 54 Lund, Sweden

henrik.gedenryd@lucs.lu.se

(I am not affiliated with Lund University, I have just kept the e-mail address.)

Draft of 6 June 2001. This text will be updated as needed.
Note that this early draft is likely to be heavily revised .

Informal overview of the paper

I have realized that this is quite a long paper. It contains a theoretical discussion, example applications, and a description of the implementation, and that is perhaps too much for one presentation. Therefore, in response to some comments, I have replaced the abstract (which wasn't very useful anyway) with some informal comments about the contents, to help you get an overview of it all:

This paper proposes a mechanism that is, for example, both a replacement for inheritance and a general solution to aspect-oriented programming—among other things. This is because I framed the problem that I address on a very general level, as program/system composition. The purpose of section 1 is to introduce this unusual way of framing the problem, and I would say that this change in point of view is the key to grasping what I am trying to do in the rest of the paper. Section 2 briefly introduces the key proposal of the paper, Universal Composition (UC). Section 3 develops the proposal and shows how to turn the abstract idea of UC into a concrete mechanism, and how to make it as simple as possible. This goes via the idea of using graphs to describe the structure of compositions. This section is also where the relation to ideas such as perspectives, views, subjects, etc. is discussed.

After that, section 4 draws out the consequences of the UC proposal on a theoretical level, in relation to concepts like program complexity, code growth, decomposability, and separation of concerns. This is to demonstrate that one can make several quite strong, principled claims about how powerful UC is. (These points should really be understood in relation to what has been written about aspects, separation of concerns, software generators, etc.) Section 5 presents an application of UC, as a concrete illustration of using the scheme. To validate some claims from section 4, it describes the scheme at a level of detail that is so great that it may distract one's attention from the central topic of UC, at least for a first reading. Accordingly, *both of these sections can probably be skipped on a first reading*, to get an overall, rather concrete idea of the message that is in there.

Sections 6 and 7 are related to the practical side of the scheme itself. The first informally describes the simple principles of how to generate programs from components and composition rules. It includes a second application of UC to implement BitBlt, to further illustrate how code is composed. The next section briefly describes the prototype implementation of the UC mechanism that I have made, and that was used to develop the examples. Finally, section 8 is meant to show more explicitly how UC is related to aspect-oriented programming.

(If you still have problems, or other comments or suggestions about the presentation or else, let me know at the above e-mail address.)

1. Introduction

1.1 The boundaries of object orientation and the frontiers of programming language research

The history of programming languages has presented a perpetual stream of new approaches to program composition. Object-orientation is now widely recognized as the state of the art in structuring software. Nevertheless the boundaries of the OO approach have been so thoroughly explored that today they are well known and documented, and as our ambitions continue to grow, these boundaries will come to stand ever more in the way of further progress. Although it may not have been overtly stated in this way, some directions in programming language research serve as the clear signs of reaching these limits and trying to overcome them. I will here briefly sketch these efforts as illustrations, and I wish to apologize if I have not done them full justice.

Inheritance

As old as object-oriented programming itself is the quest to improve on inheritance. This is the OO field's own search for a better mousetrap. Whereas inheritance proved to be a revolutionary device for code composition, it was more a serendipitous discovery than the product of a systematic inquiry to solve a well-understood problem (Dahl & Nygaard, 1981). There is even disagreement as to whether it is essential to the concept of object-orientation or not (Wegner, 1987; Kay, 1996).

While whetting the appetite for reuse and code savings, its limitations were soon encountered: single inheritance doesn't scale well beyond a rather small limit; multiple inheritance has proven difficult to use; prototypes/delegation is considered better but perhaps not enough so to prevail. As Kay has stated (Kay, 1996), no one has gone to the bottom of the problem to work out what inheritance really does, and do it right; instead, multiple inheritance (Shan, 1993), mixins (Bracha & Cooke, 1990), delegation (Chambers et al., 1991), etc. are typical ad-hoc elaborations (Kuhn, 1962) of single inheritance—they adopt a fix-it strategy rather than providing a radically new solution. This is just like the circular model of planetary motion from ancient astronomy was long made to account for new, inconsistent observations by adding ever more epicycles to it, instead of switching to a model with elliptical orbits that could account for the data in a simple and natural manner. For example, this kind of ad-hoc extension is illustrated quite well by the odd mixture of concepts in the title "Parents are shared parts" (Chambers et al., 1991).

Design patterns

Perhaps the most prominent sign recently is the design pattern movement, at least if impact and penetration is considered. For even if this is not the conventional view, the original design pattern book (Gamma et al., 1995) may somewhat pointedly be regarded as a catalog of commonly encountered shortcomings of object-oriented languages. It clearly states that current OO languages lack the facilities for capturing these design patterns in a natural manner, so that they have to be dealt with in an informal manner and be administered "manually", much like you can write object-oriented programs in non-OO languages. A few patterns, such as singletons and prototypes, even have the required support in less widespread languages. So what is the pattern book if not a collection of common problems in object-oriented systems, plus outlines for their solution, but where these solutions lack proper language support? Such support would require more advanced means for describing abstract properties than what objects and inheritance allow for.

Aspect-oriented programming

A third, more recent line of development is aspect-oriented programming (Kiczales et al., 1997; Czarnecki, 1998), which has suggested new, interesting ways of specifying underlying traits of programs that existing languages cannot isolate, promising both to increase the clarity of the solution by an increased "separation of concerns", and to obtain great savings in code size at the same time. If only our tools could handle such aspects—and we knew how to apply this technique—this might make programs shorter and clearer by orders of magnitude (Kiczales et al., 1997).

However, so far the "aspect problem" remains unsolved. From initially having pointed toward very complex solutions, such as needing a new language for every aspect domain (Kiczales et al., 1997), the level of ambition appears to have fallen drastically more lately, so that a general solution to the aspect problem no longer seems to be on the agenda . And software generators such as GenVoca and P2 seem to require rather elaborate language extensions, while their value as domain-general tools remains to be demonstrated.

Strictly speaking, aspects are the recent incarnation of a much older line of work with a great number of related contributions (Smith & Ungar, 1996), where inheritance has been replaced by "perspectives", "views", and similar concepts, which as we will see later approach the same issue in various related ways. While this line of work has shown great promise, there has never been any decisive definition of the semantics of these mechanisms, or the scope of their applicabilitity, i.e. of what they can do and how far this will go in terms of what problems they can solve. In other words, it seems never to have been defined with the precision required of a programming language construct.

Software architecture

The final example is another victim of the inflation in oriented programmings, in an increasingly buzzword-oriented world. Architecture-oriented programming, or simply "software architecture" to distinguish it from the other AOP, seems to address the limits of object orientation more explicitly than the others (Garlan & Perry, 1995), while at the same time merely reopening some classical, unresolved topics regarding program structure (Dijkstra, 1968; Parnas, 1972; Parnas, 1974; Parnas, 1979). The canonical example is Dijkstra's use of a "layered" structure for an operating system (Dijkstra, 1968).

However, the idea of an architecture of software systems seems to still be an intriguing analogy that remains to gain a non-metaphorical meaning. We need to gain better notions of what the architecture or structure of a program really is, notions that point to concrete courses of action, and we need to gain the tools to define software on a higher and more abstract level of organization than today. We need the ability to address the architecture of a system as something more concrete and consequential than a diagram on paper.

The present proposal

This paper presents a scheme that enables developers to organize programs according to any structuring principle of choice. It is thereby allows for any program architecture to be specified and followed. The quest for higher-level programming languages is a quest toward allowing programs to be built using the same elements that we use to understand and reason about the problem domain. With the flexibility of the scheme proposed here, an implementation may mirror the highest-level, most understandable representation of the problem domain and solution. As a consequence, the way to build programs with this scheme is to develop the best conceptual model of the solution that one can come up with, and then to apply that structure to the program.

Moreover, as the scheme imposes no limitations on how a program may be decomposed, or on how different program entities may share elements of their implementation (cf. inheritance), it allows for a full separation of concerns, and minimal, redundancy-free implementations. It will also specifically be shown how this solves the aspect problem in general. That is to say, this proposal allows for the full use of aspect-oriented decomposition, while requiring neither special-purpose aspect languages nor special aspect weavers, but only the basic, general-purpose language.

The way into the proposed scheme goes via recognizing that all of the above-mentioned approaches are various forms of composition, a very simple yet most powerful principle for problem solving. This paper is an attempt to take this principle and equip its user with the best possible means for applying it. The result therefore shouldn't be understood too narrowly as a solution to either of the above problems specifically, but as addressing a more general problem. While this may require some change in perspective, in the end it will hopefully be clear how the proposal addresses each of the above problems, once and for all, as it were.

1.2 Composition: a fundamental problem solving technique

If you find a problem hard to solve, try to divide it into several sub-problems and solve each of them separately. By doing so, instead of having to solve the whole problem at once, you can focus on each smaller problem on its own and can disregard everything else. And when you have solved each smaller problem, you put together the partial results to obtain the overall solution.

When you use this technique, you don't even need to fully grasp the overall picture of how or why all the small parts come together to make up the full solution. You only need to work out how the parts of each piece come together to make up the solution of that particular piece. And when you have the solution to that piece you can use it to solve other problems while completely disregarding the details of how it was produced.

This method has been introduced into computer science under various names, ”top-down decomposition”, “stepwise refinement”, “separation of concerns”, etc. However, the technique of breaking a problem into its constituent parts was not invented a few decades ago, as part of the structured programming movement. It was described and taught as “analysis” already by the ancient Greeks (Hintikka & Remes, 1974). The hierarchical form of decomposition was first described by Descartes. These ancient problem solving heuristics were adopted by Polya (Polya, 1945), whose work was undoubtedly known to the pioneers of design methodology, e.g. (Alexander, 1964), the field from which all software methodologies descend (a more thorough history is given in (Gedenryd, 1998)).

There is an additional benefit to composition, besides making the problem easier to solve. When a certain sub-problem occurs in more than one part of a complex problem, you only need to solve it once. Then you can reuse the solution wherever the problem reoccurs. Perhaps this advantage has been particularly evident in software development, but also the ancient Greek geometers reused previous proofs when putting together more complex ones.

For small problems, it may be more or less obvious what sub-problems to divide the whole into, and in any case, the impact of the choice will be small. But as problems grow, how they are broken down into parts becomes increasingly important. As we will see, the chosen structure will affect problem solving in several ways. This meta-problem of choosing how to structure the problem we might want to call “problem architecture”.

Simple though it may seem, it is also very general, and it must arguably be the most important and most fundamental of all problem solving techniques. In what follows it will be explored on several different conceptual levels, and to capture its essence in a sufficiently general and inclusive manner, I have chosen

to refer to it as *composition*: in problem solving you compose larger solutions (or problems) out of smaller ones; in software development you compose more complex systems out of simpler elements. While others may have assigned other, more complex or more specialized meanings to composition, I will use it in a simplest and most general sense, as in "how to build things out of other things". However, also the terms structure, organization, and architecture will be used more or less synonymously, emphasizing different aspects of the general idea.

Composition is embodied in a variety of programming language constructs: Procedures compose statements into behavior, records compose data, modules are one way to compose programs (state and behavior). Objects and classes (or prototypes, etc.) are two other ways of composing state and behavior. Inheritance is a peculiar form of composition that allows a particular form of sharing. The variety of these constructs also illustrates how elementary and generally applicable the principle is.

There is a very natural way to organize the composition (and sharing) of sub-problems and -solutions: make the structure of the solution follow the structure of the problem. In other words, if you find the problem domain to consist of certain "component problems", then make the component solutions correspond to these. In this way, different problems that share some properties will simply share some sub-solutions by sharing the components that deal with those properties. Note that this principle—giving the solution the same structure as the problem—lies at the heart of object orientation, because this is the idea behind objects. They correspond to entities in the problem domain, and contain (the code and data for) the solution to that part of the problem. This can therefore be said to be the essential principle of object-oriented program organization.

Perhaps this principle seems too obvious, as though there were no other way of doing things. But if so, then note that this is not how inheritance handles sharing—instead it decomposes the solution into a hierarchy of abstraction. The units, i.e. classes organized by inheritance, do not correspond to parts of the problem domain, and so the code and data contained in a class do not correspond to or address any part of the problem, nor any specific function of the program. This is only true if the class is considered together with all its abstract superclasses—but these are several different components (i.e. units of composition) of the program. One may therefore ask: *Why* should inheritance not follow this simple principle? On another level, *goto* statements do not follow this principle, whereas the "structured programming" constructs do, and this is the essence of why they have been considered superior. Suffice it here to recognize that for example inheritance does not follow this essential principle.

2. A universal composition mechanism

Composition is a crucial problem solving technique, and it is especially useful for dealing with complexity, which is a fundamental problem in all software development. It would therefore seem natural to give implementers the ability to use composition to the fullest possible extent. In fact, since a good and thorough system decomposition has long been recognized as the key to slaying the dragon of software complexity (Parnas, 1972; Brooks, 1975), it is perhaps surprising that they haven't been given the best possible means for applying this technique already.

The approach taken here is to place the full power of a universal Turing machine in the service of composition. I will tentatively call this facility *universal composition* (UC).

Such a Turing equivalent composition mechanism has one central, appealing theoretical property: It is the most powerful composition mechanism possible, there are no limitations on what can be composed with it, and so it can be used to build any conceivable structure. This is quite simple to establish: Turing's thesis states that anything that can be computed (constructed, simulated, described, etc.) can be computed by a universal Turing machine, or any equivalent machinery. (This, of course, given the usual reservation

about this thesis not being provable.) From this, it follows that if you can use the full power of Turing equivalence for composition, then any conceivable structure can be composed with it. (Note that this is not so much a proof as a mere statement of fact.)

To keep things simple I will leave the reservation about Turing's thesis implied hereafter. As we'll see shortly, you will not have to use *all* of this power to build everything you need, but you could. For this reason, the restriction is not really relevant. Besides, if the thesis would ever be found false, then the present argument would not be our foremost concern! I will also stay with a minimum of formalization, and require nothing but an informal acquaintance with Turing's thesis and its significance.

Given the theoretical strengths of this proposal, there are two obvious questions: Firstly, given the abstract quality of Turing equivalence and of terms such as "anything", what more specific advantages does this translate into in terms of software development? For example, given the lack of an upper limit in theory, is it also practically possible to remove the current limitations on how well a program can be decomposed?

Secondly, can these principled advantages be realized in practice? If such an unlimited power can be realized, does it not become unmanageable? This may happen in several ways that we may anticipate from previous work. For instance, multiple inheritance is generally recognized as hard enough to use. Does the increased generality and additional power here not make things worse? Also, the task of having to write a program (i.e. specify a Turing machine) for describing the structure of a system at first appears daunting. Does it not introduce an additional level of complexity and problems all on its own? Some previous work has required entirely new languages, such as aspect languages (Kiczales et al., 1997) and languages for software generators . Moreover, some technical implementation problems can be anticipated: How hard is it to translate the program source into running code (cf. aspect weavers)? And does this scheme not incur performance penalties, similar to those for the late binding of messages, or else require sophisticated translation techniques to achieve reasonable efficiency, such as in Self (Chambers et al., 1989)?

These questions will be address as follows: First, I will show how one can turn the abstract proposal into a concrete programming facility. This will provide some necessary grounding for the next point: Drawing out the general consequences of UC for software development; there are a number of far-reaching consequences that follow more or less directly from the theoretical proposal itself. After that I will go on to provide practical examples of using UC, which also illustrate how these theoretical consequences can be realized in practice.

3. The practical expression of UC

The key to making universal composition practical is to remember that there are infinitely many different realizations of Turing equivalence; the goal here is to find a form that is well suited for the domain of composition. To find a suitable format we need to explore this domain: What are compositions like?

3.1 Static composition

The greatest simplification of UC comes from the fact that we need consider only compositions that are *static*, in other words, that are fixed and unchanging during runtime; they are independent of program execution. There is also a dynamic dimension of composition. For example, in an object, the layout of instance variables is typically static, whereas the contents of those variables are dynamic. Thereby static composition changes only when the class declaration changes, whereas the dynamic composition of an object changes whenever the contents of an instance variable are changed. This is usually considered part of the running program than the composition of the object, and is accordingly treated by the code proper rather than the static declarations. (In integrated programming systems such as Smalltalk the distinction

between static and dynamic aspects is blurred. In this case we can say that a static property can be determined at the time of definition vs. at the time of use/execution. This corresponds to early vs. late binding.)

3.2 From trees to universal composition graphs

For the present purposes, we may also here leave the dynamic aspects to the "normal" program, and so consider only static composition. This yields the greatest single constraint on what our composition scheme will need to compute. Being able to entirely rule out the temporal dimension requires us to use much less than the full power of Turing equivalence. It also provides us with a convenient way of thinking about and visualizing compositions: everything that can be statically composed can also be depicted in a non-temporal (stateless) diagram, as a composition graph. It is this domain of all possible composition graphs that the UC mechanism needs to handle.

Traditional language facilities for data composition, e.g. record declarations, allow part/whole compositions of the form

Whole → Part1, Part2, ... PartN

where these may also be nested. This restricts the possible compositions to form arbitrary trees (directed acyclic graphs). The format of the rules also suggests that we may think in terms of a "composition grammar". Trees are straightforward and easy to understand (and are easily depicted in clear diagrams), and accordingly the composition of records or objects out of other elements has generally been considered among the unproblematic aspects of programming.

However, the very reason for the simplicity and appeal of tree-type graphs lies in the restrictions on the forms they may take—the format rules out more complex, "tangled" relations among the nodes. Trees therefore form a quite narrowly restricted subset of all possible graphs. For the same reason, this imposes a rather serious constraint on how the problem solver may break her problem down into manageable parts.

Single inheritance is even further constrained, in that it only allows for each class to be composed with a linear chain of superclasses. This is the reason behind its particularly limited scaling ability: it will decompose a problem domain nicely up to a certain limit, but beyond that point, the usefulness of single inheritance fades rapidly and can no longer prevent code duplication from growing unrestrictedly. But also the scaling abilities of hierarchical decomposition break down rapidly as the complexity of the problem domain grows. While its scaling limit lies higher, it too is incapable of stopping an exponential complexity increase in beyond that point.

Universal composition instead does not restrict the decomposition to form a tree. By definition, it imposes no limits on the composition graph. However, there are a couple of additional natural constraints on the composition, which save us from having to deal with completely unrestricted graphs. We are still dealing with part/whole relationships, which is to say a directed graph. And per definition components cannot be part of themselves. So, as in a tree every component has a number of parts, but unlike trees, where each component has one owner only, we here allow also *any number of owners* for each component. This last point is the essence of the difference between trees and universal composition graphs.

3.3 Making the graph manageable

Note that we can preserve the ownership/containment dimension in the graph: it can still always be drawn so that all owners of a component are "above" it—but perhaps no longer *directly* above it. Also, it is no longer possible to avoid crossing lines. The purpose of imposing this order on the composition graph is to make it as "tree-like" as possible. The next step is to see that in the present state, the composition graph

can be regarded as a set of *overlapping subtrees*, on different levels, and possibly nested within each other. With this we have at least to a great extent been able to get back to dealing with trees, which are already familiar and are rather easy to deal with.

If this still seems daunting to handle, then perhaps it is reassuring that the static call graphs over the functions of a program can be all this complex and more (recursion *is* allowed, for example). It is a testament to the power of composition, abstraction and encapsulation that we are able to implement such complex programs without being bothered by the complexity of the total call graph. Typically, we do not even consider the program from that perspective at all. As I will show below, these techniques are equally useful for the present purposes, and in practice the fully detailed composition graph is not very interesting either, but only those specific parts of it that one is dealing with at any given time.

But can you meaningfully deal with overlapping trees in the same way as non-overlapping ones, or is this just fancy diagram manipulation? To add some meaning to the overlapping trees, let us introduce some terminology: As before, components are the elements of composition (the nodes in the composition graph). In a parts/whole relation, *aspects* of the same whole component are parts that overlap (or strictly, that may or may not have some overlap between them). With this terminology, we may consider overlapping subtrees with the same root as different aspects on the same entity (the root). This also makes sense in terms of the common-sense meaning of aspect: different aspects of something have certain elements in common (or they wouldn't be aspects of one thing—they would be unrelated) while they have other elements which they don't share (or they would be the same thing). The choice of the term "aspect" is not incidental with respect to the aspect problem; terms such as "view" or "subject" may also be applied fruitfully.

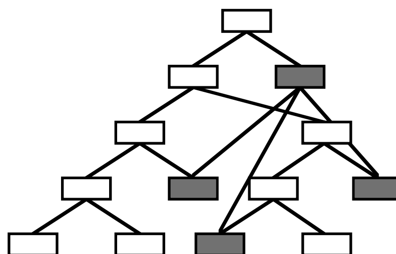

Figure 1. An overlapping subtree that is tangled with the rest of the composition graph and cross-cuts it.

To define these potentially overlapping subtrees, rules of the form *1 whole → N parts* are not sufficient. You need to allow for sharing parts between multiple wholes. One way to do this is to extend the basic rules by adding parameters:

Whole(param1, ... paramN) → Part1(…), ... , PartN(…)

You also need to allow parameters to appear as parts on the right side of a rule. In this way you can pass parts as parameters to rules, i.e. from one whole to another.

Rules of this form are sufficient for specifying universal composition graphs, and this makes the choice of an expression of Turing-equivalence simple. They are particularly similar in form to Prolog rules, but also ordinary procedural languages can handle them in a natural way. (Note: These rules could equally well be described as declarations, and are e.g. unrelated to subject-oriented composition rules (Ossher et al., 1995). As the below sections on code and data composition will show, the semantics of composition used here are very simple and unproblematic.)

3.4 Simple compositions are still simple

It would seem that these parameterized rules are much more complex than current composition constructs, and that this would make UC into a scheme that is much more difficult to use than e.g. inheritance. However, this is not the case, because UC requires simple rules to declare simpler relations, while the more complex rules are used to define more complex composition which are simply not possible using the simpler schemes. The equivalents of single and multiple inheritance would have declarations of the following forms:

Subclass $\rightarrow$ *Superclass*

Subclass $\rightarrow$ *Superclass1, Superclass2, ...*

These forms differ from class declarations in current languages only in their surface syntax, and the second form is also equivalent to e.g. that of record declarations. (However, if UC is used as it ought to be used, programs will not be organized with classes and inheritance.) Accordingly, the extra complexity is required by the more complex definitions, not by the UC scheme as such. So here, simple things are still simple, whereas also more complex things are possible.

3.5 The units of composition

While UC is general enough to allow any conceivable composition, it only describes the abstract structure of the composition, it says nothing about its contents, that is, about what is composed. This is handled by the units of composition, the components. What we want to compose here is computer programs. Because the composition mechanism is so universal, the units of composition need only be able to contain the basic elements, the primitives, of computer programs. In the kinds we are used to, these amount to state and operations (data and code). The universal composition mechanism then ensures that every possible combination of these can be constructed. It is therefore sufficient to define a component as being able to contain state and/or operations. This single unit plus UC are sufficient for replacing all the usual composition mechanisms, those for composing state (e.g. data types), operations (e.g. blocks), and combinations of the two: procedures (functions, methods, etc.), objects (classes, etc.), modules, and so on. Compare this with Kay's (Kay, 1996) notion of making each Smalltalk object "a recursion of the entire possibilities of the computer": the units of composition are very similar to ordinary objects, in that they may contain state (instance variables) as well as operations (methods). The present components are only slightly more general, e.g. in that they are parameterized.

While I will use this unit of composition to replace object declarations (e.g. classes) and inheritance in its various forms, the existing means for composing code, i.e. methods and blocks, differ from it merely in surface appearances. Therefore I will not replace these so as to introduce no more novelty than necessary. Methods are also the natural level for performing code composition.

Another simplifying measure is to recognize that objects in a sense already have parts, namely instance variables. These can be recognized as *dynamic* components. This also provides a natural basis for unifying static and dynamic components, as discussed earlier. In this way, I will use "slots" to refer to components of either the static or the dynamic kind. Static components can thereby like dynamic ones be referenced by name, e.g. in methods. (But only dynamic ones need to appear in "instances", as only their binding may change at runtime. More on this below.) One can even allow dynamic as well as static components to appear in composition rules. If so, a component is considered dynamic if its value is not determined at the time of definition (i.e. statically), just as unbound variables in Prolog rules.

Having done these modifications, we may recognize that these components are semantically identical to objects in Self (Chambers et al., 1991), where components correspond to Self's parts. Also, Self's con-

cept of “shared” parts is carried over as a shorthand for “including” the contents of a sub-component in the contents of the whole, in effect making the whole extend the definition of that sub-component. Inclusion is the discretionary counterpart to the way in which a class (always) extends its superclasses by including (but overriding) their behavior. It is however not essential to composition but merely a convenient shorthand for explicitly making references from each included property to the part that defines it. Note however that only the semantics of the composed programs are the same—Self does not support universal composition (but could easily be extended to do so).

3.6 Aspects, views, perspectives, etc.

The principle of encapsulation is supported automatically in that a component does not expose its sub-components and their behaviors to the outside. Combined with the ability to divide an entity into sub-components or aspects, these facilities provide a fully general means for abstraction and selective information hiding. Sub-components provide a natural way of dividing the behavior of an entity into subsets. To expose just a select range of behavior to the outside (i.e. a “protocol”) you put all of an entity’s behavior into such sub-components and “include” only one of them, thereby making only the behavior of that component available from the outside. You can grant access to different protocols on the same entity to different parties, by handing out references to such components instead of to the entity itself. Such references are equivalent to capabilities. Conceptually, you in this manner expose different *perspectives* on that entity. In principle, what is an aspect of an entity on the inside is a perspective on that entity from the outside. Also, compare perspectives to Goguen’s *views* (Goguen, 1986). Note that this conveniently allows one to think of capabilities in terms of perspectives on an entity. Thus, the ability to structure the implementation of an entity internally can also be used to reduce the complexity as it appears from the outside, making an entity easier to use by being able to selectively disregard its full functionality. A good treatment of these issues, and various earlier work along the lines of perspectives, already exists in (Smith & Ungar, 1996). I will therefore omit repeating it here. It is also worth noting that UC provides precise definitions of concepts like perspective, view, aspect, and so on, and also establishes the scope of utility of these notions.

In relation to this, it may be noted that Beta patterns are not quite sufficient for handling UC (Madsen & Møller-Pedersen, 1992). However, by using a technique called template metaprogramming (Veldhuizen, 1995; Czarnecki, 1998) C++ templates may be applied to a similar effect . This is a roundabout and unintended way of using templates, and which depends on the specifics of a particular compiler to work properly. It thereby does not come close to the simplicity of the present approach.

4. UC enables optimal system structure

4.1 The scaling problem

There is a basic problem of all software projects: The design of a program may hold up nicely for a while, allowing the project to progress in an orderly fashion. However, at some point, adding new features, or making modifications, becomes problematic in a number of ways. It causes the organization of the program to break down; a new feature will require several modifications, distributed over a number of places throughout the implementation. In other words, the solution to a certain subproblem cannot be localized to one specific place. This in turn means that a single feature generates several opportunities for bugs to arise for the same feature. The problem aggravates with the number of features and the state of progress. Therefore an initial “high level” design, which is conceptually clean as well as desirable for other reasons (such as various engineering criteria), gradually breaks down. Instead, the system increasingly takes on an orga-

nization that is determined by restrictions on how the implementation can be realized at all. Also, additions and modifications come to require increasingly large amounts of code for doing a similar amount of work.

From this "scaling problem" we may formulate the desired state in terms of a "principle of optimal localization": Ideally, one design feature should be located in just one place in the program, as opposed to having to be spread over several different places. This would correspond to an optimal separation of concerns. For example, consider a choice to support multiple elements instead of just one element of some feature, say multiple telephone numbers for each individual in a database. In a sub-optimal situation this may require, among other things, iteration constructs in several different places in the code, such as for reading in numbers, for printing them and for storing them. Changing the system to support several instead of just one phone number per individual will then require changes in each affected location. Ideally, the implementation of the system should contain only one item that in effect says "multiple" instead of "single", and this item should control the code for each of the affected cases. Changing this one item between multiple and single would then effect all the necessary changes in the system.

The scaling problem can be more succinctly described in the following way: Currently, the amount of effort for implementing a certain feature is a function of the size of the existing code base that it is added to. And precisely because the features cannot be made independent of each other, the function is typically exponential rather than proportional to the number of features, leading to a "lurking code explosion": the acceleration of the exponential curve means that at a certain stage, further development quickly becomes unworkable. In the same terms, optimal localization means that the required effort should be a constant, not a function of the code size. In other words, a feature of a certain size should require the same amount of effort, independent of the number of existing features, and at any point during the lifecycle.

4.2 Eliminate all redundancy

To make this idea more precise, one may think in terms of the "entropy" of a system. By analogy with thermodynamics and information theory, this denotes the amount of order in the system; the level of structuredness and organization within it. Strictly, the entropy is the inverse, the amount of redundancy in the system; a low entropy indicates low redundancy and a high degree of organization. From the theory of information transmission and data compression we recognize that the absence of redundancy means that each piece of information is independent of all others, and that this yields the smallest possible amount of data for a given amount of content. This is the greatest degree of organization that can be attained.

In these terms the scaling problem is that the entropy grows exponentially with the number of features in a system, whereas the ideal is to keep the entropy constant as the system grows, when features are added or changes are made. In other words, to make the amount of redundancy and the amount of effort per feature independent of the overall size of the system.

The point in all this is that with the ability to give a system any conceivable organization, UC has the power to solve the scaling problem for the general case. Consider any composition graph of arbitrary complexity. Given some element that redundantly appears in several of the components in the graph, you can always add a new component and make it a shared part of each of those components (i.e. draw lines from each of them to the newly added box in the graph). The redundant element (however complex) can then be placed in the new component and removed from all other places in the graph. Hence the element now appears in only one place in the design; its own place. By repeating this procedure, all redundancy can be removed from any system, which will thus reach the goal of optimal localization.

4.3 Optimally short programs

It follows by extension that for all programming problems, UC enables the solution to be expressed as the shortest possible program that is theoretically possible.

One may object that the size of a program depends on the language and the power of its built-in expressions. In principle it does, but when formulating a shortest possible program, using powerful basic constructs ought to be considered as cheating—in that way, the whole problem can be defined away, so that the resulting program always becomes one line long. So when minimal program size is concerned, all measures need to start from the same set of basic expressions. In any such comparison UC reaches the theoretical minimum by allowing all redundancy to be eliminated.

However, even in practice, the power of the builtins has a marginal effect that is also independent of program size, for the following reason: If a language requires a long piece of code for what another language provides in a single statement, then by redundancy elimination this piece needs to appear only once, i.e. where it is defined as a component, and elsewhere in the program appear as a single expression referring to that component. Thus the program of the inferior language becomes just as long as in the more powerful one, apart from that single component definition. To put it differently, UC allows any piece of code to become a single statement wherever it appears (beside its definition), no matter how complex that piece of code is. Thus, the advantage of the "better" language is constant, and in practice insignificant, except for very small programs.

It may be noted that each time an element is factored out, the complexity of the basic program is reduced at the slight expense of the size of the composition (one new component is added). Since the composition is also technically a program, it is particularly easy to see that there is a certain point of break-even where such factoring does not make the total specification smaller. This point is when the size of the component is no smaller than the factored-out feature itself. However, this may still pay off for other reasons. Either because other features will be added (or *may* be added) to the created component in the future, or because this scheme makes more sense conceptually. The bottom line is that as the organization approaches its optimum, deciding where to stop elaborating the model is in the end always a matter of judgement, not mathematics.

4.4 Complete implementation independence, an infinitely high level of expression

The next class of advantages is of a different nature. These advantages are not strictly quantifiable, but experience from software engineering suggests that they may be even more important. They derive from the fact that UC gives the designer complete freedom from implementation constraints, so that a solution can be given at an infinitely high level of expression. To design a successful system architecture, there are a number of constraints to consider and to prioritize: understandability, maintainability, scalability, reusability, implementation efficiency, etc. However, all current languages restrict the range available design options because there is one design constraint that is not negotiable: implementability. You are forced to express your design using the forms allowed by the language—or you cannot realize a running implementation. All other design priorities will be limited by what the implementation allows, and the less freedom for composition there is, the more implementation-oriented the solution will be.

The best illustration of this is to consider a recurring feature of computer science texts describing, say, an algorithm or data structure, or a more complex conceptual design; design patterns (Gamma et al., 1995) or software architectures (Garlan & Perry, 1995) are two recent examples. On the one hand, the presentation employs a high-level model, either with concepts meant to be as understandable as possible, or using some formalism, perhaps with the opposite purpose. Often there are illustrating diagrams whose elements have a natural relation to the entities of the conceptual model. However, on the other hand, the implementation is

structured in a widely different manner, because it is restricted by what the language allows for. (For design patterns, an implementation is often not given. Possibly only an outline is provided of how an implementation may be done.)

Take ordinary stacks as a simple example. Typical OO languages allow a stack to be done as a class with messages for the abstract operations. In procedural languages a stack is implemented as one or more record types that define the storage format, plus a number of procedures to call. With Fortran you would have to use explicit arrays or some such; assembly language requires you to address the machine even more directly, and so forth. Here, as the language level decreases, the program is described in terms that become more distant from the problem-oriented conceptual description of how a stack operates, and increasingly take on the format of the implementation. At each point, the concepts used are those provided by the language—class, data structure, function, array, memory block, etc. Hence, how "high level" the implementation is depends on how "high level" the constructs of the language are. It should be clear that as the means for composition diminish (in the lower level languages), the structure of the solution is increasingly determined by the implementation constraints.

In contrast, UC gives you complete freedom to choose a composition for the solution that is entirely unconstrained by the implementation. That is, it gives you complete freedom to use the building blocks that makes the program most easy to understand, build, maintain, use, or whatever principles you decide to structure the system by. Thereby it allows an in principle infinitely high level of expression, independent of how "high level" the basic elements of a language is. There are no further, higher levels that UC cannot reach, and that any other scheme could. There needs to be no conceptual gap between the entities used to explain the system and those used to implement it.

4.5 Why should optimal structure, understandability, and shortest program converge?

Hence, with UC you can also write the most understandable program that is theoretically possible. But it is not obvious that the most understandable and the shortest solutions to a programming problem are one and the same (or more strictly, that the former converges on the latter but stops a negligibly short distance from it). On the face of it, it is reasonable to suspect the opposite: optimizing a program for size would not seem to make it easier to understand! However, from another point of view, more structured programs have always been advocated as being more understandable.

Again consider the redundancy argument. Whenever there is an element of redundancy in the code, i.e. something that occurs in more than one place and can be factored out into a new component, then that is because these places share some property at a higher level of abstraction. If you factor this out and create a new component (and give it a good name and description) it comes to represent this property explicitly, from having been implicit in the less structured code. The result is a high-level representation of that property rather than a lower level description of how it is implemented.

To put it simply, the reason why non-redundancy and understandability converge is that separating the various aspects of a larger problem helps as much for understanding it as for solving it: Each subproblem is smaller and can be understood by itself, separate from everything else, and you then also see how these pieces fit together into a whole. If you contrast this with having to understand the problem as one undifferentiated whole, then it should be clear why also program size and understandability converge, by being two aspects of a low-redundancy design.

It ought to be obvious that breaking the problem down into parts helps in structuring the code, by giving clues about where each piece of code should go, and thereby enabling you to eliminate redundant elements. But as we just saw, this works both ways—working on the solution (the code) helps to reveal the internal logic of the problem, because when similarities emerge in different parts of a program this indi-

cates a shared property between these parts. You will also get other insights on the internal logic by attempting to build some concrete pieces of code. (It has also been observed in practice that new components are often identified and factored out in this way.)

This is probably one of the most important clues on how to do a minimal-redundancy design: Take a few concrete examples and try to implement them, creating new components as you discover elements that are shared between different pieces of code. Such dialectical progress, between attempting solutions and this feeding back into a better understanding of the problem, is a fundamental aspect of all problem solving and design (Gedenryd, 1998). Thus this is a good idea in general, but it is probably especially useful here, because the more complex the internal structure, the progressively more difficult it becomes to identify it, and the level of program structuredness attainable by UC is much higher than what existing languages allow for.

5. Entropi: An Example Application of UC

To illustrate the use of UC as concretely as possible, I will here discuss one example at some length, and later a second case more selectively to illustrate some specific points. The purpose of these is as example applications of UC and the theoretical properties above, not to document these applications in their own right. Thus I will skip over details and motivations that do not contribute to the present purpose.

5.1 General principles

The first example, Entropi, is a facility for automatically maintaining arbitrary relationships between different entities, or more informally, to keep the states of different objects coordinated as they change. To put it to work, the programmer specifies two entities, the relation that should hold between them, and a coordination strategy for maintaining the relation. Typically, multiple Entropies form networks of relationships that should be handled correctly. A relation is specified as a transformation that is applied to the source entity to yield the target entity. The transformation is a function where the source is the input and the target the output. To make things simple, the entities can be regarded as objects although they strictly are just values, as no assumptions are made about how they are implemented. Therefore, they need not be actual objects in memory.

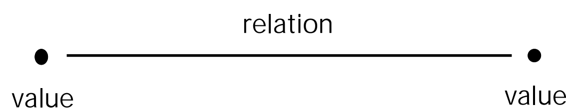

Figure 2. An Entropi relation

While a full description of the purpose of Entropi is beyond the present scope, a significant and representative part of it is as the engine in a scheme to eliminate the need for writing separate user interface code, *á la* model–view–controller, Morphic (Maloney, 1995) or similar schemes. For this purpose, Entropi automatically generates and then maintains an appropriate direct-manipulation representation of every object that needs one. Every object thus has an abstract "data" side and a graphical, direct-manipulation side. It may thereby be regarded from different perspectives, either as a "Computable" entity, or as a "Material" entity, and the Entropi system automatically keeps the two coordinated, so that changes on either side are propagated correctly.

The second major part of the functionality is that the objects are built and modified entirely using direct-manipulation operations. Objects may be composed out of other ones, plus they can be used as prototypes from which others can be derived. They may have slots, and objects may also be "included" into others in

the sense described above. (In other words, the system supports the principles of universal composition.) Here, note that “includes” and “is a prototype for”, etc., are relations between different objects. If e.g. the state of one object changes, then the graphical representations must be updated for itself as well as for all others that have it as a prototype (unless they override the value in question). If the graphical representation is manipulated, then that change must be reflected on the Computable side, which in turn affects derived objects, and so on and so forth.

Moreover, each data object may have multiple graphical formats, “looks”, and may appear in several places (in different formats) at the same time. This is the equivalent of having multiple references to an object, and is necessary for even the simplest direct-manipulation operations, like copying. Some possible formats are a plain inspector-style contents list, a graphical object (i.e. a drawing), or as a row in a multiple-object tabular list. Note that allowing for multiple looks requires the transformation from data object to graphical object to take a parameter—the look—which modifies how each object is converted into a graphical entity.

For the Material side the Morphic framework is used (Maloney, 1995), so that each appearance is a regular (composite) morph. In other words, in the central Appearance transformation (Figure 3) the Computables, which are regular Smalltalk “data” objects, are transformed into Morphic user-interface objects, morphs. The second Instance transformation (Figure 3) maintains identical clones of the source morph, where only the position, the single instance-specific property, may be different from the prototype morph.

5.2 Computational strategies

The Entropi system enabls you to specify the content of computation, the transformation, independent of its implementation, the strategy. In terms of aspect-oriented programming, it separates the what of computation from the how. Whereas the transformation can be thought of as an ordinary function and is rather straightforward, the coordination strategy is the essence of the system and mandates a more careful explanation. As an example, consider the relation between a prototype and its derivatives. If the prototype changes, this change should be reflected in the derivatives if they inherit the affected property. The abstract relation between the two is that the derivative is like the prototype, except for a set of differences. This description says nothing about how the derivative is computed, i.e. how the relation is maintained, and this can be done in many different ways.

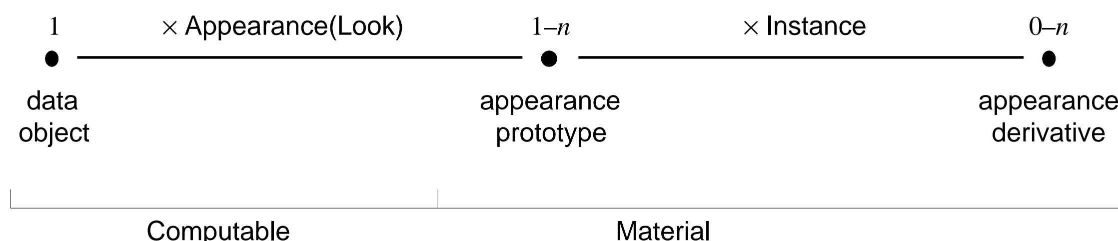

Figure 3. The Entropi relations to automatically maintain a direct-manipulation representation

The technique generally used in prototype-based languages is to use late binding, meaning that the derivative is computed when its value is requested. If a property is asked for, then it is looked up in the derivative, and if not found there then it is looked for in the prototype instead. Thus there is no explicit representation of the derivative, it is always computed as a differential applied to the prototype. Note that the code for this computation looks very different from a function describing the relation between the two. This is partly because the relation is combined with the implementation strategy, but also because the rela-

tion is to some extent unstated: “Except for” is implicit in that the lookup begins in the derivative and then continues in the prototype. Hence the relation is hidden in a lower-level implementation.

Another strategy would be to use early binding: whenever a change is made to the prototype, this change would be propagated to each derivative, accounting for differences. This algorithm may have little in common with the late binding version, at least on the surface.

A computational strategy can be regarded as a generalization of the concept of binding, where binding the value of a function means computing the value of the function to its inputs. The general principle behind different coordination strategies is that computational efficiency is based on a trade-off between space and speed. You can use memory to save processing, but you may also perform extra operations to save memory. Early and late binding essentially represent the extreme ends of this trade-off. Early binding computes the value when one of the inputs changes, and stores the result. Whenever the value is requested it is simply read from memory. In contrast, late binding computes the value each time it is requested.

Here, early binding performs the necessary computation only once, but requires storage for retaining the result until it will actually be requested. Late binding does not need any memory to store the value, but instead may perform the same computation any number of times. Hence, each of the two strategies has its advantages, and which is best depends on the circumstances of each case, such as whether the transformation is expensive, or whether the inputs change more often than the value is requested. There is no simple answer to which is best.

Entropi allows for any computational strategy. Between the two extremes, there are a potentially infinite number of intermediate strategies that trade memory for processing in more subtle ways. Caching is perhaps the most important kind. Note how a cache uses some memory to avoid repeating the same computation (an advantage of early binding), but also ensures that no value is computed unless it is needed (an advantage of late binding). And caching is not just one strategy but has many variations that cover an entire band of the trade-off spectrum, based on e.g. how the retained entries are chosen and the amount of memory used.

5.3 Connection schemes

Beside performing the actual computations, Entropi has to maintain connections between values so that coordination can be triggered and propagated properly. Consider early binding as it is used e.g. for global variable references in standard Smalltalk-80. Whereas message receivers are bound late, the value of a global variable is bound early. When a method referring to a global is defined, the referent of the global is fixed immediately, so that if the value of a global changes, then all previous references to it will point to the old, now obsolete value. Since the system doesn’t keep track of and update all uses of the variable, the way to work around this is to perform a search for all there places where it is used. Keeping the references in storage and computing them (performing a search) are then two possible connection schemes, and also these two exemplify the memory-for-computation trade-off. However, the two kinds cannot be combined entirely at will, as different computation strategies require matching connection schemes: for example, in an early computation strategy the source needs to propagate changes to its targets, whereas in a late strategy it is the target that needs to know the source from which to compute its value. Even so, the Entropi architecture allows connection schemes to be specified separately from strategies for computation. Because the Entropi composition rules are expressed in a full programming language, it would be simple abd natural to augment them with code for validating such combinations of strategy (cf. (Goguen, 1986)).

A more advanced possibility is to have connection schemes that work across more complicated boundaries, in particular network connections, but also between different processes or in a multi-processor system. Given the current Entropi architecture, they would be straightforward to implement. Here, the ad-

vantages of separating this functionality from everything else become especially obvious: For example, robust inter-network coordination can be provided entirely "behind the scenes". Such a strategy can be combined with independently written transformations and computational strategies, where new items of both kinds can be written without having to involve any of the networking code, or even looking at it.

5.4 The Entropi architecture

As stated earlier, UC enables you to structure an implementation by using the conceptual model that most naturally describes the system. This "theory of the problem domain" is typically also the one that arises in the sketches and specifications used to develop the principles of the implementation, and which is eventually used to explain the system, for example in the documentation.

At the highest level, Entropi can be thought of as a directed graph or network, where the nodes are the coordinated values and the arcs are the actual Entropi relations. In the implementation, these correspond to the components EntropiArc and ValueNode. The purpose of Entropi is to maintain relations between values; what the values are used for is complementary to its function. Therefore a ValueNode is simply something that can be asked for its value; the implementation resides almost exclusively in the Arc. The functionality within an EntropiArc is divided straightforwardly between the three already described aspects: the Xform is the "what" of computation (the relation/transformation), the ComputationScheme is the "how", the computational strategy, and the ConnectionScheme keeps track of related values.

EntropiArc(xform, compScheme, connScheme) →
xform, compScheme, connScheme.

5.5 The ComputationScheme architecture

Principles

The key to the architecture of computational strategies is that it must be sufficiently general to allow for any possible strategy. To be understood on this level, one can think of a computational strategy as as a chain of computation that implements the propagation from source to target. There are two basic modes of propagation that need to be considered: push- and pull-style computation. Each of the two modes is triggered and propagated by a separate kind of event. The first is triggered by a source change, the "early" event, and pushes the change from source to target, forward across a relation. The second mode is triggered by a request for the target value, the "late" event, and pulls this value from its sources, triggering computations backwards. This is the first major difference between the two modes.

A second difference between these two modes is that the push mode always must store its value in memory until it is requested, or it would be lost, as discussed above. This is not the case in the pull mode, which is needed precisely in the cases when the requested value is not stored (i.e. pre-computed). To handle pushing efficiently, one should exploit that there already is a stored value for the target, and therefore not require the whole value to be recomputed every time. (In the pull mode this is not an issue, as there is nothing stored.)

The best strategy is therefore to just change the already stored value. For this reason, the push mode only propagates changes, not values. Accordingly, the transformation is applied to the change message and its arguments, and then that transformed message is sent to the target object. In essence, the pull mode transforms and propagates values, whereas the push mode transforms and propagates (change) messages. This is the last major difference between the two modes.

In supporting the propagation of changes and not merely of values, the resulting system becomes very efficient. It can thereby generate the same computations as hand-coded schemes, and therefore imposes no computational overhead.

Dealing with change messages means that transformations may be more complicated than a simple function. However, by default the transformation of a push message is applied only to the argument of the change message, so the baseline transformation is still a simple function. This works in many cases, but more complex transformations may require e.g. that the selector be changed as well. This typically happens when the source and target are of different kinds (e.g. a plain Computable object and a user-interface morph).

The bottom line is that the implementation of any computational strategy needs to provide just one method each for handling the early and late events; these two methods implement the two modes of propagation. This is sufficient to allow for every possible computational strategy.

The chain of computation

While the early and late strategies could be implemented as two methods each, there is a great deal of redundancy between the two. And in practice some support code is required in addition to the two methods. Therefore it is a good idea to exploit this redundancy, by developing the architecture to an even finer level of detail.

The redundancy lies in that both strategies comprise different mixtures of the two basic computational elements, state and processing, which yield their different storage-vs.-processing characteristics. Therefore each chain can be further specified as binding together its source and target with a simple chain of state and processing nodes. Each node on the chain gets its input from the left and forwards its result to the right. A Xform node simply applies the transformation to its input. The State node will merely place its value in storage, or read the value from storage, depending on whether it is being pushed or pulled.

Each chain is divided into two halves, where the first half of the computation is triggered by a source change, and the second half by a target access. Thus, as the figure shows, the late strategy has an empty early computation, so that the late part takes the source value, applies the transformation to it, and outputs the resulting target value. The early chain uses a storage node to join the two parts of the computation. Thereby the early part takes the source change, applies the transformation to it, and forwards the result. The late part simply takes the value from storage and uses that as its output.

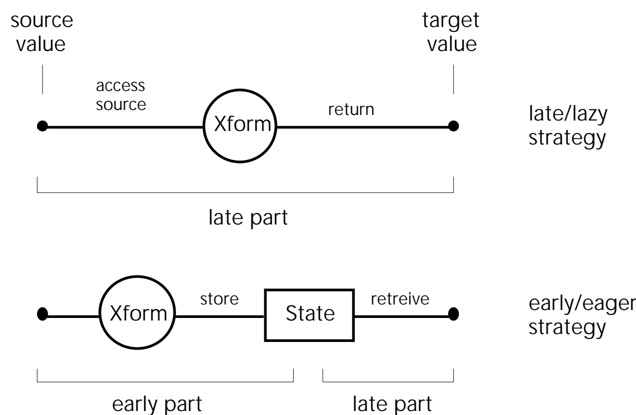

Figure 4. Computational chains for late and early strategy

Support for special architectures

Similar dataflow-like architectures have been reported by others. The original aspect report (Kiczales et al., 1997) used a similar solution in an image-processing example, and the case of the software for Tektronix oscilloscopes is a paradigmatic example in the software architecture literature (Garlan & Perry, 1995). It is probably no coincidence that such a dataflow network architecture reoccurs. It is likely to be a generally useful solution, as it is a natural way to compose mathematical operations. What should be noted here is how UC can be used to leverage specialized schemes such as this one.

Firstly, UC allows you to directly use such chains for the architecture of computational strategies. After defining components that represent the nodes, different strategies can be defined simply by assembling these nodes into different chains. In Entropi, the computational strategies have been defined as chains of XformNodes and StateNodes, where these are two kinds of ComputationNode, and which share its protocol. In addition, ComputationNode holds default implementations of this protocol.

The methods that you obtain by building such a chain are then the actual implementations of the strategy. In this manner, the composed strategies do not need to contain a single line of code, as everything is put together automatically out of code residing in the Nodes. The details of how this works are the topic of the next section. (However, caching strategies could hardly be built out of such simple nodes, but would require new kinds of Node to be added. By choosing these well, a few, properly parameterized kinds would be sufficient for generating a wide range of caching schemes.)

Secondly, when using chains, UC can provide support for defining such architectures in a more direct manner. Composition rules are naturally suited for expressing (overlapping) hierarchical compositions, or what we might call hierarchical or tree architectures—this is arguably the most generally useful single architecture. However, what makes chains into a special architecture is that they don't match the form of trees very well. A chain is not a set of part/whole relations, but a sequence of linearly connected nodes, where the output of one is the input of the next.

This is where UC can provide special help. Instead of using traditional composition rules, which are somewhat awkward for non-tree structures, a small amount of specialized composition code allows a chain to be defined merely as a list of the nodes to be used. The code in question puts the listed nodes together into a proper chain, e.g. by making each node the input of its successor, and by appropriately hooking up the ends of the entire chain. It thereby factors out and hides the irrelevant details of defining a specific chain, and moves it into the architecture. And by doing this once and for all in the specialized code, this task needs also be debugged in just one place.

In this way, the composition of a chain can be defined in a more purpose-oriented way, at a higher conceptual level, than if plain composition rules were used. This is a consequence of having the full power of a Turing-equivalent language to draw on for composition.

The ConnectionScheme architecture

Compared to computational strategies, connection schemes are much simpler. They serve best as examples of basic, uncomplicated compositions. A connection scheme has two basic parts, one forward link in the source end and one backward in the target end. The composition rule for ConnectionScheme then has one slot for each link:

ConnectionScheme(forwardLink, backwardLink) →
forwardLink, backwardLink.

Specific connection types are then defined by providing contents for each slot, for example:

ForwardConnection→
ConnectionScheme(TargetReference, None).

BackwardConnection→
ConnectionScheme(None, SourceReference).

It is useful to have a generic "null" pseudo-component (here called "None") whose contents are empty. This is because to be flexible, generic components tend to accumulate a large number of parameterized properties, where some of these cover special cases and are therefore often left unused. A generic empty component reduces the need to define a dummy component every time a property is left unused.

6. Composing programs

So far composition has been discussed in the abstract, without considering how the actual programs are put together. For instance, the computational chain architecture is worth its while because you can obtain the implementation of a computational strategy directly from building the corresponding chain, not requiring you to write any code for the strategy. To see that this involves neither magic nor any intelligence other than the developer's, let's recapitulate what composition does.

Composition divides a greater problem into a number of subproblems; *universal* composition makes it possible to divide the problem in any conceivable way. In this way a developer can divide an implementation into subproblems, and implement solutions for those.

In the present example, when writing the code for one of the ComputationNodes, the chained architecture has reduced the overall problem of computational strategies into the small subproblem of merely connecting the Input and Output of the node properly. If you are implementing the XformNode, you need to apply the given transformation to the input and pass on the result; in a StateNode, you simply store the input in memory for later; and so on.

So how can the implementation of a strategy just "emerge" from the composition of the chain? Remember that the composition corresponds to how you have divided the overall problem into subproblems. When you provide the solution for the subproblems, the structure of your composition reflects how the sub-solutions should be put together to solve the greater problem. For instance, the layout of an early or late chain reflects how the basic elements should be put together into a corresponding computational strategy.

So the solution structure that remains largely implicit in regular programs has here been made explicit; it is reified in the structure of the composition. The task of composing the solution, the actual program, merely consists in mechanically transferring this structure to the code (and data).

6.1 Composing data

First consider the simpler case of composing data, i.e. objects. The data of the Entropi system is used to represent Entropi networks. It should be divided into different units that are the most convenient elements to build Entropi networks from. The best unit for an object is normally one Entropi relation, although some complex units compose several segments, so that they really represent a small sub-network in one object.

Logically, a connection needs to be divided so that one end of the connection is associated with the source value, and one with the target value. But to keep the values entirely separated from and "unpolluted" by the Entropi system, the segments have been designed so that the value objects do not need to contain any connections, or even be aware of being coordinated. Otherwise, each coordinated object would have to contain components connecting it to its incoming and outgoing EntropiArcs.

To solve this, the connections are kept entirely within the segment objects, and all coordination must be handled from there. In this way, no “changed” messages are required within the code of the values, as in Smalltalk’s dependents mechanism. Therefore a segment always connects the Arc to its target value, and in fact handles all the possible connections for this value, including those going out from it. Thus, because connections must be split, it contains the end half of the incoming connection, and the front halves of 0–*n* outgoing connections. Hence, an EntropiUnit represents a segment of an Entropi network, and contains one incoming EntropiArc (including the end half of its Connection), its target EntropiValue, plus OutConnections, where the latter maintains the 0–*n* front half Connections of outgoing Arcs.

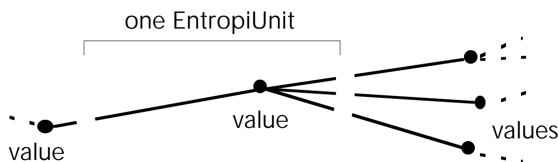

Figure 5. The scope of an EntropiUnit

The Unit component is also responsible for propagating coordination across successive connected segments. To support different kinds of segment, the actual functionality is factored into an EntropiProtocol component, so that it can be put into different kinds of Unit to share the functionality. An Entropi object is always made up of an EntropiUnit. For example:

LateUnit → *EntropiUnit(IdentityXform, LateBindingChain, BackwardConnection, NoTargets)*

EarlyUnit → *EntropiUnit(IdentityXform, EarlyBindingChain, ForwardConnection, MultipleTargets)*

Here, a late-bound strategy need not forward its updates since these always request its value when needed, and therefore doesn’t need to keep track of its targets. However, it needs to be able ask its sources for their values to compute its own. The early-bound strategy, in contrast, needs to update its sources but always gets its updates from its sources. Thus it needs no links to its sources but instead must keep track of its targets.

Now consider how to build an Entropi object. The composition of the object is given as a network of components with sub-components (i.e. a composition graph). The “actual” slots would appear as dynamic sub-components in the appropriate components. For example, the “value” slot would appear as a sub-component in the composition rule for the StateNode component, where the contents of this slot are not statically defined by the rule, like an unbound variable in a Prolog rule.

StateNode(…)→ value …

As a simplest solution, it would be entirely possible to build a composition by simply creating an object for every component, each having slots (instance variables) that refer to its sub-components (Figure 6). This is based on the idea that instance variables can be regarded as components whose contents are dynamic, i.e. may change during the execution of a program. Here, also static components would be explicitly represented. This solution would work perfectly in principle, the only problem with this approach is that there will be a large number of objects for each instance, requiring a large amount of memory.

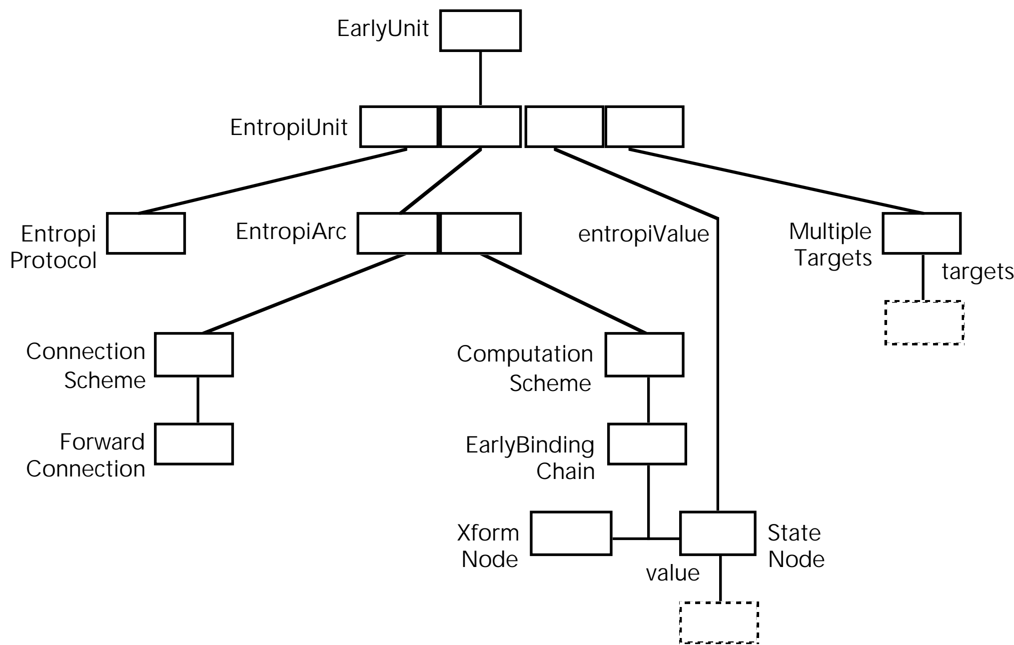

Figure 6. Partial composition graph for an early EntropiUnit. A hierarchy of explicit objects for each component, with a slot holding each sub-component. Dynamically bound sub-components dashed; component names capitalized, slot names in lowercase. What is shown here is only a partial composition graph; a large amount of detail has been omitted.

A better solution is to separate the static from the non-static components, and equip each instance with slots only for the dynamic ones. You compose the static structure once and for all at the time of definition, and at the same time build the instance format by collecting the dynamic components, then keep the static definitions shared by all instances, much like class variables. This strategy is based on the converse of the above idea, namely that static components cannot change during execution. Therefore they need not have a mutable slot in each instance of the composed structure.

Therefore, you obtain an optimal instance format with only the necessary slots. (The value slot holds the value object, the source slot a reference to the upstream EntropiUnit objects, and the targets slot holds a collection of downstream Unit objects; cf. Figure 7.) Since code, including variable references, is composed on a very high level, it is not a problem that the same variable may appear in different positions in different instance layouts.

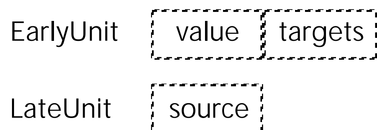

Figure 7. Instance layouts for two sample EntropiUnits

6.2 Composing code

Code is composed by using an extension of the same principles. Just like a crude form of data composition can be done with ordinary objects, code can be composed in a basic but functionally equivalent manner using ordinary methods. Consider objects composed like above, with explicit objects representing each

component. Since components present themselves just like ordinary instance variables, their code can be invoked by sending them ordinary messages.

To add a new EntropiUnit to a network, the Unit is sent a *source:* message with another Unit as the argument. The message is implemented in EntropiProtocol and included into the Unit. Figure 8 traces the messages that would be sent through the explicit component objects of an EarlyUnit instance, where a ForwardConnection is responsible for maintaining the connection between the values (the branch of the ComputationScheme is left out).

Here, the FwdLinkEnd (the end half of the forward link) is responsible for setting up the connection. It does so by sending a message to the front half of the link, which stores a reference to the new Unit instance. (This also illustrates why e.g. a ForwardLink needs to be represented also on the end half of a connection.) The backward link is not used and thus holds a None component, which holds an empty pseudo-implementation of the invoked message (i.e. which just returns *self*).

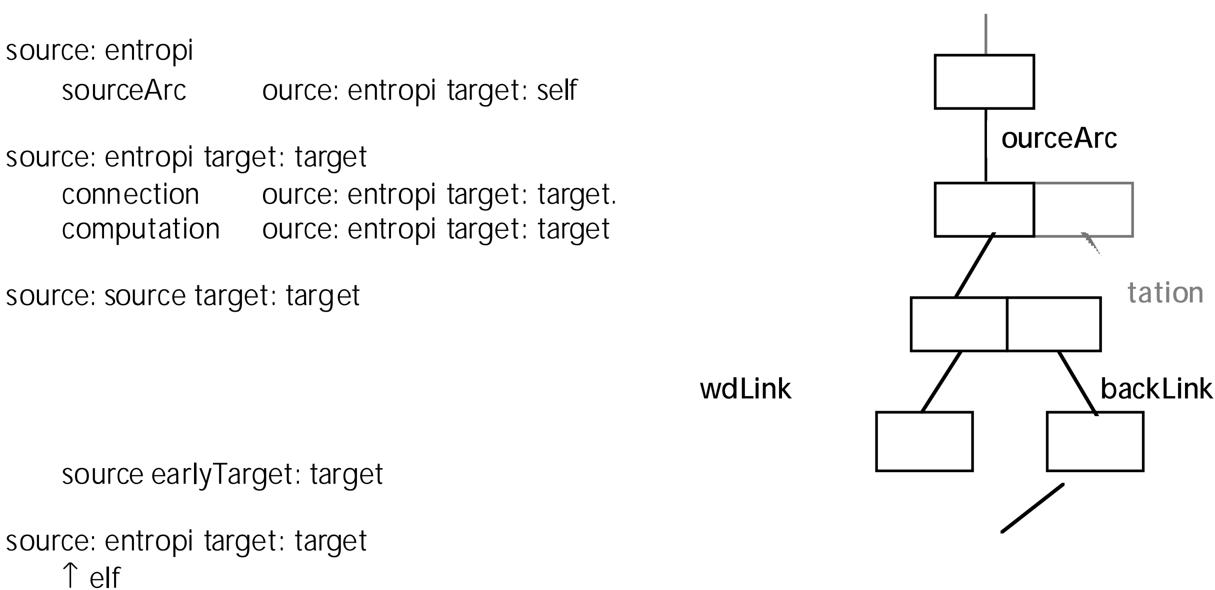

Figure 8. Method composition from static composition graph. A method on the left belongs to the component on the right.

This illustrates the simplest way of composing code, by writing methods that explicitly mirror the structure of the composition. In the example it is easy to see that the call graph for the message follows the composition graph for the static composition of an EarlyEntropiUnit. In this way, the structure of the composition is transferred to the executed code.

In principle, the composition of dynamic behavior is like static composition, except that also the dynamic dimension of the composition must be specified. Even though one tends to read a linear sequence onto a list of sub-components, the part/whole relationship does not really provide any ordering among them. It is therefore not sufficient to compose code for a whole merely by concatenating the respective pieces of code in its parts. You need the ability to specify how the pieces of code should be related to each other also in time, i.e. dynamically.

This is easily solved by writing a method that invokes the methods in the respective parts in the desired order. In the example, this is what the methods in EntropiArc and ForwardConnection do: they merely pass the *source:target:* message on to each of the sub-components, and contain no actual operations of their own. In other words, they only serve to specify a dynamic ordering for their parts.

Meta-methods for dynamic composition

A look at the Entropi implementation reveals that several components only contain messages that repeat the same dynamic pattern for different selectors. This suggests a more advanced way of specifying dynamic composition, which also serves to make the dynamic pattern more explicit: using a meta-protocol for method composition. Instead of requiring a method for each selector, which repeats the same dynamic pattern, you could allow for a generic meta-method that is used by default. For EntropiArc it might look like this:

```
methodFor: aMessage

    connection methodFor: aMessage.
    computation methodFor: aMessage
```

Going further, generic versions might be provided, so that a component could just specify which one to use:

```
combineAllParts: aMessage

    self parts do: [:part |
        part methodFor: aMessage]


firstImplementorOf: aMessage

    (self parts detect: [:part |
        part hasMethodFor: aMessage])
      methodFor: aMessage
```

In this way, the meta-method for EntropiArc would read:

```
methodFor: aMessage

    self combineAllParts: aMessage
```

This resembles the rules for method combination in e.g. CLOS (Kiczales et al., 1991) and those for aspect combination in AspectJ . However, a pertinent question is whether this practice would make an implementation less obscure, rather than more so.

Eager evaluation

As with data composition, the naive version of composing methods by late-bound messages is very inefficient. In the simple example above, six redundant "composition" messages are sent to yield the single real effect, the message sent to the front end of the created link, which resides in FwdLinkEnd. However, by applying a technique similar to the one used for data composition we can turn this into highly efficient runtime behavior.

Note that the purpose of all message sends, even in ordinary languages, is to compose different pieces of code. The message sends have no semantic effect in themselves, besides for polymorphic messages, where different receivers cause different pieces of code to be composed. Dynamic dispatch delays this

choice to runtime—this is late binding—while static dispatch can be performed in advance, i.e. be early bound.

To achieve greater efficiency, we can separate between static and dynamic composition of code just like we did with data above, by seeing whether a message receiver can be determined statically or if the decision must be delayed until runtime. In the present discussion, message receivers are either static or dynamic components. (Also message arguments may be receivers, but they behave as dynamic components.) So it is precisely the same dynamic components, whose values cannot be early bound, that need to be explicitly represented as slots in each instance, and also need to have their message receivers late bound, based on the value of the same slot. (From this, it is also clear that it is the same principle at work in both cases.) These late bound messages are then separated from messages to static components, which can be early bound.

This can be described as "eager evaluation"—evaluating as much as you can as soon as you can—as opposed to lazy evaluation, which puts off all evaluation for as long as possible. The more common term is *partial evaluation*, an established technique for program transformation. It can be regarded as the most general form of code optimization, subsuming many specialized techniques such as inlining and constant folding. Based on a static analysis of the source code, partial evaluation (PE) attempts to perform the operations of a program in advance wherever possible, producing an equivalent version of the program as a result. To take a very simple example, it would transform the expression *3 * 2 + a* to *6 + a*.

It should be clear that an operation can be performed in advance precisely when the result can be determined statically. This makes the combination with Universal Composition into a natural match, with their shared emphasis on separating early and late evaluation. Partial Evaluation should not be mistaken for a general optimization technique which has merely been brought in to improve the performance of UC, but from which any other language may benefit just as much. For instance, because of Smalltalk's highly consistent use of late binding, the same technique would have virtually no effect on ordinary Smalltalk programs, save for the occasional literal expression such as *3 * 2*.

Particularly relevant here is the PE technique of inlining messages. Where sending messages composes code dynamically, inlining is the static form of code composition. This means not merely to early-bind the message receiver, but to actually perform the entire message send, adding the code of the receiving method to the sending method, including parameter substitution and whatever else necessary. The effect is that the message itself vanishes from the resulting code.

Applying eager evaluation to the earlier example results in a method where all static message sends have been removed, so that the artifacts of the static composition are gone:

source: entropi

source earlyTarget: self
… the code for the ComputationScheme …

Just like for data composition, the difference from the naive version is radical, and the code that remains is highly efficient—by any measure. (The use of meta-methods would not affect this result negatively.) The effect is that the expense of a computation doesn't have to be higher than it absolutely needs to be. This principle may be expressed as "when the cost is zero, the price is nothing": if e.g. a certain composition can be computed once and for all, then it shouldn't be computed every time it is needed.

6.3 Using UC to implement BitBlt

To see just how efficient the translation from a high-level design to the resulting code can be, we need a better illustration. This will be the second example of a UC application, and it will be given a much more brief treatment than the first.

BitBlt is the original Bit BLock Transfer operation of Smalltalk-80, used to transfer bitmap graphics e.g. to the high-resolution display. As pixels may be smaller than memory words, this involves not merely simple memory moving operations, but potentially also shifting the contents to handle sub-word positioning. And with the advent of color graphics, the operation has been extended to handle the various formats that pixels may come in, e.g. 2, 4, or 8 bit color map indices and 16 or 24/32-bit direct RGB values, potentially with alpha channels. In its general formulation, the transfer also includes a transfer function, in which case it computes the function from the source and destination pixels, and overwrites the destination with the resulting value.

Squeak's version of BitBlt

In Squeak (Ingalls et al., 1995), the implementation of BitBlt is written in a subset of Smalltalk which may either be run as is, which is too slow for real-time graphics, or be translated into C from which highly efficient machine code may be generated. The need for speed naturally places tight constraints on the implementation. This applies in particular to the most speed-sensitive inner copying loop, which to maximize its speed looks much the opposite of normal Smalltalk code, which typically comes as a number of small, well-factored high-level methods.

In BitBlt, the inner loop is a large monolithic method, over a hundred lines long, and with several highly similar chunks. It could in other words easily be refactored for a much better organized but also slower version. Also, several variants of this inner loop are provided, so as to exploit special circumstances to make certain more important special cases as fast as possible. A mere glance shows that there are great similarities also between these. They differ merely in the optimizations which have typically been applied by hand (other differences could only count as bugs).

In all, it is evident how the concessions for maximum performance come at the expense of most everything we consider good programming practice, and this trade-off would of course turn out no more favorably if coding directly in C, or even assembly language.

A UC version of BitBlt

Of course, standard C provides no mechanism for decomposing BitBlt into manageable units besides function calls, but neither the C++ implementation of classes would probably be fast enough for an efficient version of BitBlt. However, if we use UC, the situation is quite different.

(The version presented here may well suffer from the author's limited experience with the domain at hand. It should therefore be considered as an illustration of the principles of UC more than as an authoritative UC version of BitBlt, which ought to be designed by a domain expert.)

It seems appropriate to have the architecture follow the high-level description of the BitBlt operation, as a general pixel-by-pixel transformation that takes the source and destination pixel maps as its inputs, and the destination as output. From this, we may define the source and destination as PixelMaps. From this, many aspects of the algorithm can be expressed as properties of the pixels: this includes the pixel sizes in bits, and whether they are indexed or direct colors, and if so the RGB(A) format of the color. They may also include endianness for portability.

Also, much of the low-level processing can be encapsulated as higher-level operations on the pixel maps, thereby hiding dozens of parameters like the respective width and height of the pixel maps, and much internal state that is kept track of during the operation, like positions of the current pixel in each map.

A Transformation can be used to organize the remaining aspects: besides the transfer mode various auxiliary functions like the conversion between different pixel formats and the application of halftoning. To handle intra-word pixel alignment, bit rotation and masking can be encapsulated in a Skew aspect, and another aspect can encapsulate the size of machine words.

Moreover, once the implementation has been organized in this manner, only minor changes are required to extend it with new features like endianness conversion, or to support special hardware extensions (like MMX and AltiVec), by adding these in their own isolated components as is appropriate. A more sophisticated Blt architecture might use a computational chain to organize the transformation, an would thus use different chains for e.g. the left and right edges.

The resulting code

By using such a design, the resulting source code for the general (non-inlined) inner loop is straightforward to understand and approaches the level of pseudo-code. It may look like this:

```
copyLoop

    self setupVerticalLoop.
      1 to: sourceMap height do:  [:line |
        self prepareLine: line.
        self resultPixel: self transformedPixel edge: skew leftEdge.
        self nWords > 1 ifTrue: [
         self copyHorizontalLine.
         self resultPixel: self transformedPixel edge: skew rightEdge].
        self  nextLine]
```

In this version, the pixel transformation is expressed directly in its general, most high-level form:

```
transformedPixel

    ^combinationRule
        source: source pixelValue
        destination: destination pixelValue
```

This is how the same operation appears in its most normal (no special case) variation in Squeak's handwritten version:

```
...
thisWord _ self srcLongAt: sourceIndex. "pick up next word"
sourceIndex _ sourceIndex + hInc.
skewWord _
    ((prevWord bitAnd: notSkewMask) bitShift: unskew)
              bitOr:     "32-bit rotate"
       ((thisWord bitAnd: skewMask) bitShift: skew).
prevWord _ thisWord.
mergeWord _ self mergeFn: (skewWord bitAnd:
```

```
    halftoneWord)
              with: (self dstLongAt: destIndex).
    self dstLongAt: destIndex put: mergeWord.
    destIndex _ destIndex + hInc
    ...
```

The UC-generated version of this piece of code is identical to this one, except that stores to intermediate variables are used only when necessary. A first version was written to mimic the original exactly, but the later version captures the intention behind the code much more directly by specifying what to compute without saying how to do it (i.e. whether to use intermediate assignments). It relies on the PE engine's ability to insert local variables when necessary, i.e. when the result of a function is used more than once.

And instead of writing alternative versions by hand to optimize them for certain conditions, different variants can be generated by altering the parameters in the composition rules. Then, whenever possible, optimizations are applied automatically during the partial evaluation. For example, one of the most important special cases is when no source bitmap is used (e.g. when simply filling the destination with some color). To obtain a special version of the inner loop for this case, when you compose the Blt you merely replace the SourceMap with a None-style component, which generates empty operations for every source-related part of the code. The resulting inner loop is instruction-by-instruction equivalent to the hand-optimized version of the same method.

Optimal performance

Table 1 compares the speed of hand-written and UC-generated versions of the BitBlt inner loop, where these have been translated into C. The differences come from the C compiler's varying ability to optimize equivalent versions of the same code but with or without explicitly assigning intermediate values to local variables.

	depth	Original	UC	
paint	1	63	62	98%
	8	190	179	94%
	32	608	611	100%
over	1	10	11	110%
	8	78	84	108%
	32	323	350	108%

Table 1. A comparison of execution times of hand-written and UC-generated versions of BitBlt.

As the table shows, the performance cost of applying UC is practically none, even compared to a hand-optimized version of this highly speed-sensitive algorithm. And whereas the hand-written code trades a great deal of clarity and conciseness for optimal performance, the UC version has been given a high-level architecture that lies very close to the most convenient conceptual description of the operation.

This is not peculiar to the present example, but a general consequence of the earlier principle that a composition may be completely independent of the implementation (and vice versa). Here, this point has been

given a concrete meaning: the implementation can be made highly efficient, without being negatively affected even by a very high-level architecture. The general consequence of this is that even an infinitely high-level composition never precludes optimal performance—a rather unlikely conclusion.

In the present case, the performance comes from only using static components. However, it may be mentioned that UC also allows for using different late-binding schemes for different compositions, so that these may be adapted to the characteristics of each specific case. For instance, the late binding of a choice between boolean true and false values may be done by a simple test—which yields no overhead compared to a non-late-binding implementation. This is another example of the principle that UC needs to incur no cost beyond what is necessary. The prerequisite is of course that the composition is done properly.

7. Implementing UC

7.1 Composition rules and the composition algorithm

The prototype version of UC that was used for implementing the above examples is grafted onto the Squeak meta-object protocol, and uses a rudimentary Smalltalk implementation of Prolog for the composition rules. The format of composition rules suggests Prolog as a natural choice for an implementation, and together with the pattern matching paradigm this language also served as the initial inspiration behind these concepts. Additionally, the unification algorithm in Prolog allows you to express static structures directly without specifying the order of computation—in other words, you can leave out the temporal dimension when specifying the composition. As discussed earlier you only need a small subset of the language; no backtracking is necessary, for example.

In principle, a composition rule looks like a regular Prolog clause:

component0(arg1, arg2) :-
component1(args…), component2(args…), … .

However, there are a couple of complications: firstly, you have to allow "component slots" on the right side; secondly, the gap between Prolog and Smalltalk needs to be bridged.

The basic composition algorithm is very simple; if regular Prolog clauses were used, then a composition could be computed simply by resolving the composition rule ("?- component(arg1, …)."). The working algorithm augments the basic resolution algorithm to handle the above complications; this adds nothing of principled interest.

Once completed, the composition is converted from a Prolog structure into a tree of objects that is stored as part of the class definition of the hybrid class, as a classical "reflection" of the object's structure. This reflection is then used by the meta-object protocol on the Smalltalk side to compute the object format (instance variables), and when "weaving" methods.

The naive version of the resolution algorithm was found inefficient for moderately complex definitions, and this turned out to illustrate the degree of redundancy that can be attained in compositions. The used version of Prolog does not share a single instance between repeat occurrences of the same element in a rule. Components being shared among several parents, a.k.a. "tangling", is characteristic of low-redundancy compositions. Because of the non-sharing algorithm, a tangled component and its entire tree of sub-components would be represented by separate object trees for each parent.

As discussed earlier, for a low-redundancy design, the size of an untangled tree grows exponentially in relation to the linearly growing tangled version. For two examples of single- and multiple-segment Entropi units, respectively, a composition (graph) of about thirty unique nodes yielded roughly a thousand untan-

gled nodes; seventyish nodes about 5000 of them; the latter reflective structure would in all occupy more than 1MB of object space, and the time required to compute it grew accordingly into the range of several seconds. This had not been intuitively anticipated for definitions that would easily fit a single sheet of paper! However, in retrospect this ratio is an accurate reflection of how much redundancy the design eliminates. It serves as a good illustration of the linear vs. exponential characteristics of low- and high-redundancy designs.

The naive implementation was replaced by a standard graph-traversal algorithm that visits each node only once, and thus has linear performance. The result was a negligible use of both time and space even for the most complex case.

There have turned out to be two main disadvantages with a hybrid approach. The first is that you need to bridge the gap between the two languages, which makes the composition code somewhat more complicated. Of course, it makes the UC implementation more complicated as well. The second is that the language support is typically weaker for a hybrid solution. The chief obstacle in the prototype is the lack of good debugging facilities. This is only needed when you exploit UC in earnest, to build complex definitions. However, due to Prolog's backtracking nature, when the need arises, you are truly lost without this help! This may mean that a non-hybrid solution is to prefer, because it benefits you most when your need is the greatest—i.e. for complex definitions—whereas the lacking simplicity is not such an obstacle when the rules are simple anyway. The foremost disadvantage is that you will have to explicitly deal with some lower-level aspects in the composition code since it does not fit the form of composition rules so well.

7.2 Code composition

The reflective nature of Smalltalk's compiler provides a very good foundation for code composition. It is used to convert back and forth between source text, parse trees, and byte codes. The bulk of the implementation consisted in adding a partial evaluation (PE) facility into the existing classes for the syntax nodes that represent parse trees.

This was done as a completely separable and generic PE engine for Smalltalk that has no knowledge of the semantics of UC. Instead it requires that an "Evaluator" object is supplied, which works like an umpire or arbiter that upholds the rules of evaluation. The PE engine will consult the Evaluator about whether the value of a reference can be determined statically, i.e. from the source code and the static context alone. The Evaluator is asked for the value of 'self' (wherever it occurs), the receiving method of a message, and so on, and the answer will either be the value or "Unknown".

To this, the UC prototype provides a class that implements the Evaluator protocol with the appropriate semantics of UC. For example, if the receiver of a message is a static component, then the invoked method can be determined and it is looked up and returned. It will then be inlined by the partial evaluator.

Three different modes of code composition have been implemented: static, dynamic, and just-in-time composition, in this order. The first generates and installs all methods for a component upon a user command. The second composes a single method on the spot, typically when its source code is requested from a browser. It should be noted that this happens quickly enough for real-time use, even though the process has been little optimized (such as by caching intermediate results) and the sources for each inlined method are read from disk to include comments. (Because Smalltalk includes a Decompiler, a much faster way with no additional work is to decompile the bytecodes instead, but this means that comments are lost.) Finally, a just-in-time version composes a method when it is invoked at runtime. This was done later, in part because the dynamic version was found to be so fast. It required little extra work; the main reason for waiting with this version was that the UC implementation ought to be sufficiently stable, or you may have bugs popping up at unpredictable times and under highly inconvenient circumstances.

The relation between these modes can be illustrated by applying Entropi concepts to code composition (and code translation in general). The relation or transformation is between source and machine formats, and the three kinds of translation just employ different strategies of computation: an interpreter uses late binding, a compiler early binding, while a JIT uses a caching strategy. Thus, a compiler for example tangles the "pure" transformation from source to machine code with an early-binding strategy of computation.

7.3 Tool support

The prototype implementation was not developed as a pure experiment, but to simplify the implementation of Entropi, where single inheritance was perceived as a great if not insurmountable limitation. For this reason, it was designed to be as practical for real development as ordinary Smalltalk: It needed to work with the ordinary browser and debugger, and source code management tools such as change sets and the changes file, all in a robust and dependable manner. All of this required a relatively small amount of code.

8. Relation to Aspect-oriented programming

The relation of UC to aspect-oriented programming deserves some closer attention.

> "We present an analysis of why some design decisions have been so difficult to cleanly capture in actual code. We call the issues these decisions address *aspects*, and show that the reason they have been hard to capture is that they *cross-cut* the system's basic functionality. … in the course of this paper [we] will be considering units of system decomposition that are not functional." *(Kiczales et al., 1997)*

It is worth noting that the separation of concerns is simply another name for (de-)composition, the most essential of all problem-solving heuristics. The original contribution of AOP was to identify an important unit of composition, the aspect, which is not supported by traditional languages. However, aspects were defined in terms of "non-functional" decomposition, which now has turned out to have been something of a red herring.

As previously discussed, UC offers a straightforward interpretation of what an aspect is: it was defined above as an overlapping sub-tree in a composition graph. This definition captures the essence of aspects as discussed by AOP; for example, an overlapping sub-tree in a diagram provides a direct illustration of tangling and cross-cutting, two essential AOP concepts (cf. Figure 1).

This also shows how essential the idea of aspects is to UC: You gain the power of universal composition by extending traditional, hierarchical decompositions to also allow for non-hierarchical structures (i.e. overlapping subtrees); in other words, by introducing aspects. Thereby, designing with aspects (or views, perspectives, or whatever name you prefer for the same general idea) is the skill we need to master in order to realize the full power of UC, which enables a complete separation of concerns.

The idea of non-functional decomposition seems to have come from the notion that traditional decomposition is able to handle the conventional program (i.e. which implements its basic function), and that the new form of decomposition thereby would concern "non-functional" parts of the program logic. (Just what is "a unit of system decomposition that is not functional"?)

I would suggest that the fallacy here is an unstated notion that every program has one identifiable, canonical Function (which is addressed by the functional decomposition) and that aspects therefore have to be subordinate to it, and of a different nature. The key is to change perspectives to seeing that there is no single, divinely designated Function, but that one may analyze any system in terms of several, potentially

overlapping functions, and that this is a more powerful form of analysis that leads to a design with multiple aspects or points of view, each corresponding to one function.

Take Entropi as an example: there is nothing to gain by identifying either of its main functionalities (or concerns) as being primary to the others. The power lies in identifying and separating them, and then to treat each of them on its own, and in being able to enforce this separation, even though this may require a more advanced problem decomposition than a simple tree. So AOP did not introduce form of decomposition that is no longer "functional", but which isn't *hierarchical*. Being an aspect is thus a feature that is structural, and not content-related as "non-functional" suggests. And unlike the latter, the structural definition of an aspect also naturally matches the common-sense meaning of aspect, as a point of view on the whole composition.

The "aspect weaver" is the machinery responsible for building a running program out of the code that is distributed over the different aspects (Kiczales et al., 1997; Czarnecki, 1998). The present scheme uses only a single principle of composition with an extremely simple semantics. For this reason, the task of putting together a running program becomes very simple, as described in the sections on composing data and code. Because of the simple semantics, the whole issue of join points seems to disappear, as the corresponding function is performed by ordinary messages. Thereby, code can be composed either dynamically by ordinary run-time message-sends, or statically by partial evaluation. The latter would correspond most closely to what an aspect weaver does.

Like aspect weavers, software generators (Czarnecki, 1998) are simply devices for program composition. By analogy, dynamic languages "generate" multiple behaviors by binding together implementations in various classes and superclasses, but they do so by message sends during runtime. Although the details may vary, the only principled difference is that software generators perform the same function ahead of runtime. This is a difference in the strategy of computation; the function that is performed is identical. Perhaps the generative function of composition is more apparent there because it has been separated from, and thereby is not tangled with, the execution of the program itself.

The point about minimal program size should be seen in the light of the drastic reductions in code size that have been reported by these approaches. An aspect-oriented implementation yielded code that was 35 times shorter in an extreme case (Kiczales et al., 1997). A reduction of roughly twenty times compared to an Ada version was achieved with a software generator, compared to a fivefold reduction with C++ (Singhal, 1996). And even though such a figure needs to be taken with great caution, it has been estimated that the theoretical minimum for the latter case would be 150 times smaller than Ada. The point here is that UC is sufficient to achieve this theoretical minimum for every program.

However, even in the light of the classical result that the cost per line of code is almost independent of the implementation language (Brooks, 1975), the line count is not the important point. Instead, it is to draw out the full consequence of the AOP demonstration: An optimal separation of concerns both leads to the most understandable implementation, and will by itself converge on the minimal program size. This is strictly just a result about good program structure, and the software engineering benefits of this will not be repeated here. However, the important point is that UC allows these benefits to be gained to the fullest extent, whereas current languages do not by far allow this degree of decomposition.

9. Conclusions

Composition (and its complement, decomposition) is the technique of dividing a larger and more complex problem into several smaller and simpler problem parts, and then to assemble the solutions to these parts into a solution to the original problem. It is the ultimate means for dealing with complexity.

The idea behind Universal Composition is to enable programmers to apply this technique to the fullest possible extent. This was achieved by reducing composition to its most elementary form, and then to couple it with the ability to apply it in any conceivable way.

One theme of this paper has been to show that a number of older as well as more recent developments are really special cases of composition—in lighter as well as more elaborate disguises. The point is that a universal scheme for composition can subsume these special forms all at once.

Perhaps more importantly, UC can provide a general solution to aspect-oriented programming, since it is such a special case, but which has previously lacked a general solution.

But to do only this would be to stop short. Existing languages place various constraints on how a program may be composed. The structure of an implementation is therefore based on the mechanisms of the used language (such as inheritance). UC imposes no such constraints. Therefore, to make the most of its capabilities, on its own terms, instead of using it only as a better old thing, this is the question to consider:

If there were no restrictions on how to structure an implementation, what would be the best way to organize it?

The answer that has been given here is to take the same conceptual structure that is used on paper, on whiteboards, and so on, when reasoning about the problem domain, and to apply it also to the implementation. The problem of finding a good structure of course remains with the designer, but poor program structure can no longer be attributed to language limitations.

This paper has aimed to demonstrate that universal composition is a practical scheme, not just a theoretical construct that can subsume all other means of program composition. The examples that were given demonstrate that designs that use UC indeed can be made to directly match the clearest conceptual model of the problem domain. Moreover, the UC version of BitBlt showed that even the highest-level implementation does not need to impose any restrictions on runtime performance.

10. References

Alexander, C. (1964). *Notes on the Synthesis of Form*. Cambridge, MA: Harvard UP.

AspectJ. . *Homepage of AspectJ™*. Xerox PARC, Palo Alto, CA. http://aspectj.org/.

Bracha, G., & Cooke, W. (1990). *Mixin-based inheritance.* Paper presented at the Proceedings of the 1990 ACM Conference on Object-Oriented Programming Systems, Languages and Applications.

Brooks, F. P. (1975). *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison–Wesley.

Chambers, C., et al. (1991). Parents are Shared Parts: Inheritance and Encapsulation in Self. *Lisp and Symbolic Computation, 4*(3).

Chambers, C., et al. (1989). An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. *Proceedings of OOPSLA '89, SIGPLAN Notices, 24*(10).

Czarnecki, K. (1998). *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* Unpublished Ph.D. thesis, Technische Universität Ilmenau, Germany.

Dahl, O.-J., & Nygaard, K. (1981). The Development of the Simula Languages. In R. W. Wexelblat (Ed.), *History of Programming Languages* . New York: Academic Press.

Dijkstra, E. W. (1968). The Structure of the ‘T.H.E.’ Mulitprogramming

System. *Communications of the ACM, 18*(8), 453-457.

Gamma, E., et al. (1995). *Design Patterns, Elements of Object-Oriented Software*. Reading, MA: Addison-Wesley.

Garlan, D., & Perry, D. (1995). Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Eng., 21*(4).

Gedenryd, H. (1998). *How Designers Work*. Lund, Sweden: Ph.D. thesis, Lund University Cognitive Science.

GenVoca. . *Home Page of the Product-Line Architecture Research Group, http://www.cs.utexas.edu/users/schwartz*.

Goguen, J. (1986). Reusing and interconnecting software components. *IEEE Computer, 19*(2), 16-28.

Hintikka, J., & Remes, U. (1974). *The Method of Analysis: Its geometrical origin and its general significance*. Dordrecht: D. Reidel Publishing Co.

Ingalls, D., et al. (1995). *Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself.* Paper presented at the Proceedings of OOPSLA’95.

Kay, A. (1996). The Early History of Smalltalk. In T. J. Bergin & R. G. Gibson (Eds.), *History of Programming Languages–II* (pp. 511-578). New York: ACM Press.

Kiczales, G., et al. (1997). *Aspect-Oriented Programming.* Paper presented at the Proceedings of the European Conference on Object-Proented Programming ECOOP’97.

Kiczales, G., et al. (1991). *The Art of the Metaobject Protocol*. Cambridge, MA: MIT Press.

Kuhn, T. (1962). *Structure of Scientific Revolutions*. Chicago, IL: Chicago UP.

Madsen, O. L., & Møller-Pedersen, B. (1992). *Part Objects and their Location.* Paper presented at the Proceedings of TOOLS 7, Technology of Object-Oriented Languages and Systems, Dortmund.

Maloney, J. (1995). *Morphic: The Self User Interface Framework* : Sun Microsystems, Mountain View, CA.

Ossher, H., et al. (1995). Subject-oriented Composition Rules. *Proceedings of OOPSLA’95, ACM SIGPLAN Notices, 30*(10), 235-250.

Parnas, D. (1972). On the Criteria for Decomposing Systems into Modules. *Communications of the ACM, 15*(12), 1053-1058.

Parnas, D. (1974). *On a ’Buzzword’: Hierarchical Structure.* Paper presented at the Proceedings IFIP Congress 74.

Parnas, D. (1979). Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering SE-5, 2*, 128-137.

Polya, G. (1945). *How To Solve It*. Princeton, NJ: Princeton University Press.

Shan, Y.-P. (1993). *Panel: Is Multiple Inheritance Essential to OOP?* Paper presented at the Proceedings of OOPSLA’93.

Singhal, V. P. (1996). *A Programming Language for Writing Domain-Specific Software System Generators.* , Univ. of Texas, Austin.

Smith, R. B., & Ungar, D. (1996). A Simple and Unifying Approach to Subjective Objects. *Theory and Practive of Object Systems, 2*(3), 161-178.

Veldhuizen, T. (1995). Using C++ template metaprograms. *C++ Report, 7*(4), 36-43.

Wegner, P. (1987). *Dimensions of Object-Based Language Design.* Paper presented at the Proceedings of OOPSLA'87.