Facets of Multi-Stage Computation in Software Architecture

Walid Taha **Tim Sheard**

Oregon Graduate Institute
20,000 NW Walker Rd,
Portland, Oregon
92779, USA
+1 503 690 1121 x.{7047,1439}
{walidt, sheard}@cse.ogi.edu

ABSTRACT
The goal of this paper is to demonstrate that an important, naturally-occurring, and disciplined form of reflection in software systems is readily expressible at the architectural level by using a new architectural operator (connector) called Engage. We begin by extending a simple data-flow architectural description language with this operator. We illustrate the expressive power of Engage by using it to describe both familiar and experimental software architectures. In doing so, we also demonstrate the prevalence of a reflective organizational pattern called *multi-stage computation*.

We are not aware of any other architectural description language that captures the essential functionality of the Engage operator.

The paper's treatment is rigorous, but not formal.

Keywords
Architectural design languages, program generation, multi-stage computation, reflection.

1 INTRODUCTION

The success of any large system is greatly affected by its architectural design [6]. Software architecture is the medium for communication between the technical and managerial parties involved in the development of a software system [13, 18]. Occasionally, the performance of a software system can become an issue that demands a high-level decision, for example: "The user interface is too slow". In such cases, it is important that the architectural description of the system provide some indication of where high performance penalties are being paid, and that there be clear tradeoffs that can be made at the architectural level.

This paper has been published as an OGI Technical Report. It is also available from http://www.cse.ogi.edu/~walidt

The key idea of this paper is that reflection, a particular kind of dynamic behavior, can be expressed at the architectural level in a simple and effective manner. Reflection can have a profound effect on the performance of a system.

1.1 The Need for Reflection

Almost every scripting language provides a form of reflection, and almost every general-purpose computing system provides a scripting language. Typically, such languages are used to initiate compilation, especially as part of a configuration process. After compilation the generated file is usually "run". At this level, the action of running a generated program is often not recognized as computational reflection. The main argument is that such a system is not an "introspective, self-modifying" one. However, program execution as we have just described it constitutes a rather limited form of computational reflection, in that the meta-system changes its state: a new program has now become part of the existing architecture. In any architecture, dynamically executing a program that has just been generated is inherently unsafe, as there are typically no guarantees to the form of this program. At the same time, this architectural pattern is desirable for performance reasons. In practice, a certain level of safety is ensured by using only simple scripts when running dynamically generated programs. However, the real key to the safety issue with respect to this kind of reflection is the restriction that the over-all architecture of the system being considered must be specified before hand, even if not all of the architecture's components are available at the very beginning of this architecture's operation.

1.2 Organization of this Paper

We begin by presenting GDL, a skeletal graphical notation for describing the architecture of generation systems. We

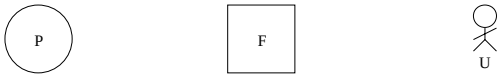
illustrate the basic architectural problem that arises from reflection, discuss possible solutions to this problem, and advocate one of these solutions.

By systematically going through a number of important examples, starting with compilation and going on to high-level program generation, and other architectures where this kind of dynamic behavior arises, we illustrate both the importance of this behavior, and the ease with which it can be expressed at the architectural level.

The paper is concluded after discussing ongoing and future works.

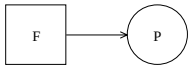
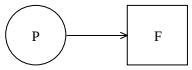
2 A GENERATOR DESCRIPTION LANGUAGE

GDL is a graphical notation for describing the architecture of program generation systems. A GDL description is a graph with the following three kinds of nodes:

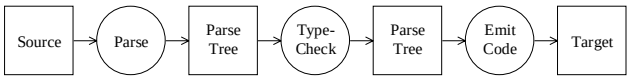


denoting a (*running*) program P, a data entity F, and a user U. Any two useful nodes are connected via an edge. The semantics of more sophisticated data flow languages have been formalized elsewhere (See for example [16]). For the purpose of this paper, an informal description is sufficient. While users play an important role in the description of any real system, they are not of immediate relevance to the essential subject matter of this paper, and hence their treatment will also be minimal.

There are two basic kinds of edges, distinguished by the kinds of nodes they connect:

Edge	Means
	Program P reads entity F.
	Program P writes entity F.

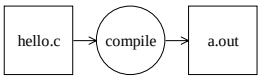
The bearing of the edges (left, right, up, down) is not relevant. With these elements, a familiar multi-pass compiler can be described by the following diagram:



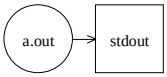
More detailed descriptions at the architectural level are also possible using GDL, but more so with hybrid description languages (See [13], [16] and [8] for example variants of the compiler example). For economies of space, and to avoid possibly distracting detail, we will generally work at an even higher level of abstraction; it is at this level that the multi-stage pattern should manifest itself.

We can follow some straight-forward abstraction steps to hide some of the details of the GDL description above. Such abstractions steps will be discussed in more detail in Section 5.

Now we can illustrate one of the problems that arises due to the presence of reflection by means of a minimal example: consider a program hello.c that prints "Hello World" to standard output (stdout). When this program is compiled, the result is an executable file (a.out):



Eventually, the user may wish to execute this program. This can be represented at the architectural level as:



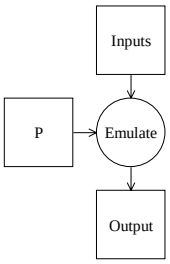
A question this paper attempts to address is: *What is the relationship between these two diagrams?* We know how to describe these two different aspects of the lifetime of this simple program, but how do they fit together and can they be represented in the same framework?

To combine the two diagrams, we are faced with the problem that a.out plays two different roles in each of these two lifetimes: as a data entity, and as a program. The presence of the two distinct manifestations leads to a form of type-mismatch if we attempt to unify the two. The problem arises in any generative architecture.

Our claim in this paper is that a solution to this problem is indeed feasible. In the rest of this section, we will discuss a number of reasonable options, and advocate on of these options as the solution.

2.1 The Relativity of Reflection

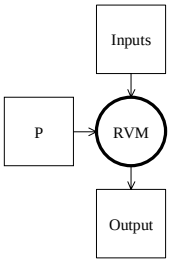
There is more than one solution dealing with the type-mismatch arising between viewing programs both as data and as active processes. In particular, we can employ a conceptually simple emulator for machine code. This construction can be represented as follows:



While such programs do in fact exist, they are significantly more expensive in terms of space and time utilization. If we represent running all programs in this fashion, simply to allow our architectural diagrams to "type-check", then

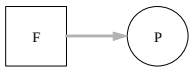
we have failed to capture the essential behavior of the underlying implementation at the level of the architecture: Both the use an emulator and a "realistic" implementation would have identical representations at the architectural level.

What we really need to do here is to hide reflection by using a "Real Virtual Machine (RVM)". This machine represents the hardware on which the machine code will be "interpreted". In other words, we are trying to make the underlying computational machinery of the architectural framework explicit:



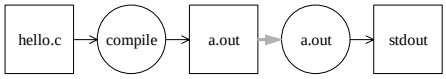
The main disadvantage of this notation appears when we try to describe the execution of more than one program within the same architecture: We either have to make many duplicate copies of RVM, which would be misleading, or we can make all executions explicitly share the same RVM in the GDL description. The second option would result in very cluttered diagrams. In addition, the details of which input (or output) belongs to which program will be lost.

In essence, an optimal solution should be some well-behaved coercion between the data and the program "types". We therefore propose a (hopefully) more intuitively appealing convention, namely that of "engaging" the data-entity D into the underlying computational machinery, which makes this engaged image of the data entity be known to the rest of the architecture as program P.

Edge	Means
	Engage entity F into architecture as a program P.

In the standard sequential setting, Engage can be interpreted as "running D under name P". Having the two names, P and F, is not necessary, but turns out to be useful in GDL diagrams, especially as it gives more insight into the dual role that this same object is playing.

Using Engage, the complete architecture whereby the program hello.c is both compiled and executed can be described as follows:



Note that the kind of reflection that this language can describe is still tightly restricted: The full architecture that will host the dynamically engaged entity is known a priori, despite the fact that not all programs making up this architecture exist initially. In this paper, instead of arguing for the pragmatic benefits of such a restriction, we demonstrate how numerous interesting multi-stage software systems that are used in practice do indeed have an architecture that is fully known a priori.

Naturally, not all software systems have an architecture known a priori: when a user buys a PC, neither she nor anyone else can know what programs and with what architectures will be run on this system. Certain applications such as operating systems must be extremely dynamic. GDL cannot express the dynamic aspect of such systems. At the same time, and as we will see in the following two sections, there is a wide variety of interesting and useful applications where this restrictive condition holds.

2.2 Reflection without Tears

Engage is a special kind of reflection, in that it does not involve altering the state of the meta-system, which we believe is a generally unsafe practice for developing programs in the large. An example of such undesirable reflection is changing the semantics of the programming language, by dynamically changing the interpreter or the compiler at runtime. Another such example, is self-modifying programs. The fact that Engage cannot express either of these kinds of behavior provides a certain level of safety and conceptual simplicity.

What Engage allows us to express is the fundamental difference between compiling systems and interpretive systems, which will be illustrated in this paper. The significance and relevance of this distinction is no longer confined to theoreticians and programming linguists: Today, high-level program generation research and applications are picking up speed, as is reflected by the interest in Domain-Specific Languages (DSLs). We [20] and others [19] have identified high-level generation as a powerful technique for effectively compiling DSLs, and thereby avoiding most (if not all) performance overhead that may be associated with using them in performance-critical systems. This use of multi-stage computation to compile DSLs is further discussed in Section 4.5.1.

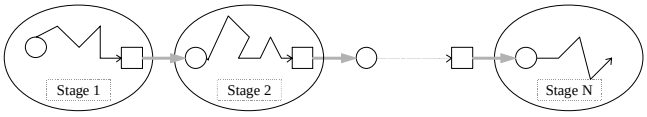
3 MULTI-STAGE ARCHITECTURES

At this point, we have introduced all the machinery needed to be able to define multi-stage computation. Multi-stage architectures are best described as a pattern arising from

the involvement of the Engage operator, either implicitly or explicitly, in an architecture.

Any architecture can be partitioned into stages. Stages create equivalence classes of sets of components. A stage T is said to come after a stage S if there is an Engage connection going from a data entity in S to a program in T.

The following diagram portrays an N-stage system:



Here the stages are represented by the larger ovals. For this partitioning to be interesting, there must be some dependency between the out-going data entity and the in-coming program of each stage. This is typically reflected by the presence of a path in the GDL graph from the in-coming point to the out-going point. In addition, this internal path must contain at least one program. However, this condition is vacuously true for all stages but the first.

A multi-stage architecture is one consisting of more than one stage.

Even in a highly-concurrent implementation, each stage probably needs to remain completely blocked until the previous stage is completed: Part of the architecture for this stage simply does not exist until the previous stage is completed.

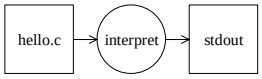
4 APPLICATIONS

To illustrate both the expressiveness of GDL and the prevalence of multi-stage architectures, we demonstrate how a variety of both common-place and research architectures can be described using GDL.

4.1 Program Execution

The simplest way to execute a program is through the use of an interpreter. An interpreter inspects the input program, and performs the actions that this program represents.

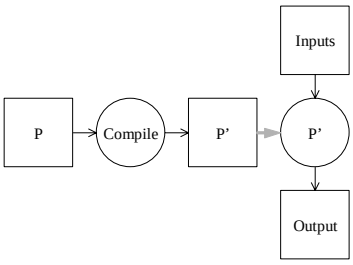
It is possible to imagine using an off-line interpreter to execute a C program:



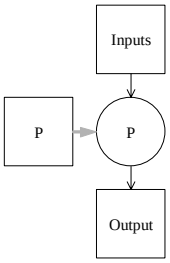
This architecture is simpler than that of compilation-based execution. In particular, the Engage operator at the architectural level disappears. But at the same time, interpreters are typically orders of magnitude slower than compilers. This observation is fundamental: We have noticed that in many existing architectures, Engage-like reflection is frequently used for the sole purpose of improving the performance of the overall system. At the same time, compilers are also typically orders of magnitude more expensive to develop than are interpreters.

4.1.1 The Engage Notation Revisited

It is often convenient to use the Engage notation where programs are really being compiled and then executed. That is, the following architectural fragment:



Can also be represented as:



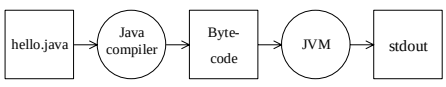
Without being considered inaccurate, because this "reduction" is an abstraction step. Abstraction steps will be discussed in Section 5. The Engage operator still stands for effective reflection.

The Engage operator could also be used even when it is (temporarily) being implemented by an interpreter. This is possible for a different reason, namely, to express a semantically-motivated intuition. However, strictly speaking, an interpretive implementation is not really conforming to the architectural specification as presented so far.

To allow for this latter kind of use of the Engage operator, we will say that reflection is primitive only when it involves the execution of machine code directly (or invokes the interpreter of the meta-system, whatever that system may be). The use an interpreter as described above is referred to as non-primitive reflection.

4.2 The Java Virtual Machine

Implementations of the Java programming language employ another interesting architecture for program execution:



In this architecture, the Engage operator is, once again, absent. Instead, it is replaced by a byte-code interpreter. While this design choice can mean some performance overhead, it has the advantage of making the output of the compiler portable: The byte-code program can be execute on any architecture where a Java Virtual Machine (JVM) is available.

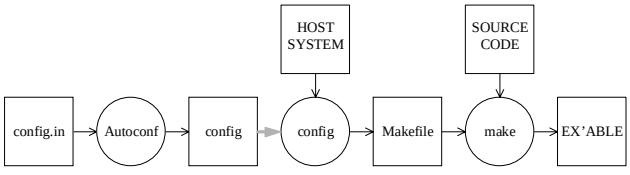
However, the interplay between reflection and portability is limited. In particular, the need for the Java Virtual Machine does not arise because of reflection, but rather because the original output of compilers used to be in machine-code. In fact, we can imagine a refinement of the above architecture, where the Java Virtual Machine is implemented by first generating machine-code, then engaging it. Such systems are commonly known as Just-In-Time (JIT) compilers.

4.3 Application Configuration

Software configuration and installation provides another good example of multi-stage computation: Often, multiple versions of the same software exist in a software development house, each for a different architecture or operating system. Ideally, these different versions are automatically generated from the same source. Usually, this generation step is relatively straight forward. At the installation site, the installation utility queries the particular system for its various parameters. This can be a relatively sophisticated process, involving either the selection of pieces of code to install, or the generation of some parts of system being installed.

The GNU Autoconf facility provides a good example of such a system [15]. Autoconf is designed as a generic software configuration and installation framework, and is employed in the configuration and installation of most GNU applications.

In preparing a software system for distribution, Autoconf takes specifications from a file "configure.in" and generates an appropriate configuration file "configure" which is system-specific. At the installation site, "configure" is executed to generate a make file (Makefile). The make utility is then used to execute this make file to guide and control the compilation of the sources into an executable. This can be represented by the following diagram:



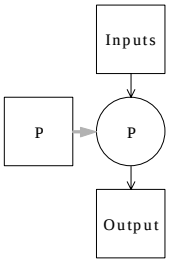
Note that we could have used the Engage operator to connect Makefile and make. This is not entirely accurate: Make interprets the make file, rather than effectively reflecting it. However, using the Engage operator here would capture the intended semantics of the make file more accurately if we wish to avoid an interpretive view.

As the World Wide Web and its applications mature, we imagine that the difference in time between the generation of the distribution configuration file and its execution at the installation site will become shorter and shorter. As such global architectures develop, both customers and systems developers will need to become more and more aware of their structure, and how they fit together.

4.4 Off-line Partial Evaluation

Now we come to a special kind of software system that was devised to *automatically* stage programs: an off-line partial evaluator [10]. Off-line partial evaluation systems also provide us with the first example of three-stage computation.

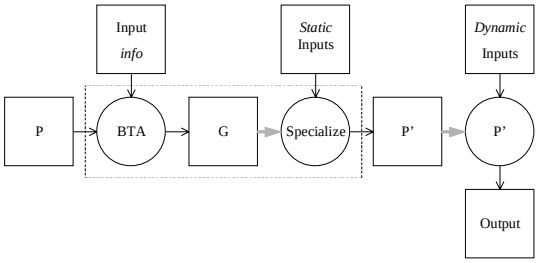
Let us consider the execution of a basic program, with one input and output:



Here, all inputs have been lumped together as one data entity.

The basic idea of off-line-partial evaluation is to take advantage of the fact that some inputs may be available before others. This is accomplished by first performing a Binding-Time Analysis (BTA) based on both the input program, and information about its inputs. At this point, the only information provided to BTA (Input *info*) is whether each of the program's inputs is static or dynamic. An input is static if it is available during partial evaluation, and dynamic if it is not.

The result of BTA is an annotated program. In essence, the annotated program is a program generator G. When executed, this program generator first reads P's static inputs. Then, it performs all P's computations that depend only on the static inputs. Finally, it generates a new program P' (called the residual program). Now, P' only needs to read the dynamic inputs (whenever they become available) and perform the rest of P's computations to produce the desired output. In GDL, this architecture can be accurately described as follows:



The dotted box encloses what is usually considered to be the partial evaluator proper.

The essential feature of this architecture is that it factors out the execution step of program P, into two distinct stages, and performs the first one of them. It is the role of BTA to perform this (non-trivial) factorization of the input program P. The complexity of this step depends both on the input information (specifying which inputs will be Static, and which will be Dynamic) and the complexity of the program P itself.

We are not aware of any other architectural description language that can express this essential functionality of an off-line partial evaluation system.

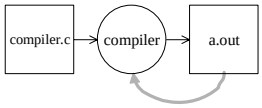
Originally, off-line partial evaluation systems were designed to convert the input program from single-stage to two-stage. Recently, significant progress has been made in the development of partial evaluation systems that can convert a single-stage program into an N-stage program, where N is any arbitrary integer [7].

4.5 Bootstrapping

Bootstrapping is an important technique for the development of realistic compiler systems [1]. In this paper, bootstrapping provides us with a rich domain of examples where we can have architectures of an arbitrarily large number of stages.

4.5.1 A Problem

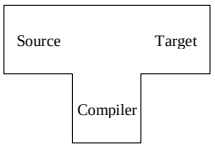
The key idea of bootstrapping can is roughly described by the following architectural description:



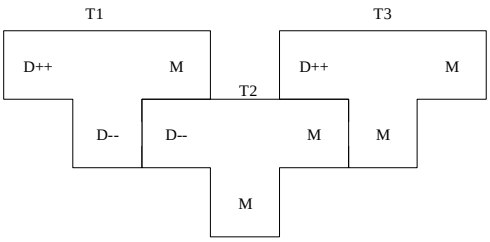
After a new compiler is generated from the source, it can be used to compile itself. Such an architecture raises the need for computing a fixed-point. This is a very demanding requirement that complicates the logic for reasoning about compiler development unnecessarily.

4.5.2 A Solution

Compiler developers have avoided this problem by using the following well-known construction for a "bootstrapping step":



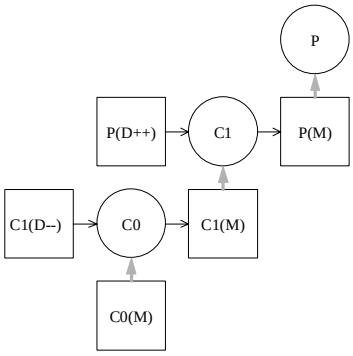
With these "T diagrams", the need for the loop in the architecture is removed, essentially by making each "version" of the compiler unique. For example, let us consider a two-step bootstrapping development process where a compiler for a language called D++ is developed by first developing a compiler for subset called D--, and then developing a compiler for D++ in D--. This process is represented by three elements, T1, T2, and T3 connected as follows:



Note that T2 is acting as an architecture-transformer: It takes the architecture represented by T1, and generates a new program with the architecture reflected by T3.

4.5.3 An Alternative Solution

We contrast the T diagram with the GDL description of the same architecture, starting from the same basic ingredients:



In this diagram, C0(M), C1(D--) and C1(M) are the compilers represented by T2, T1, and T3, respectively.

While the GDL description is somewhat simpler than the T diagram, it hides the architectural-transformation aspect that the T diagram clearly reflects: Just by looking at the C1(D++) box, we cannot tell that this data entity will eventually become a program. While this can be deduced in this case by following the edges of the graph, if C0 had more than one input then it would not be able to say which

data entity did in fact embody the structure that C1 reflects.

On the other hand, the T-notation is somewhat ambiguous, in that two touching Ts do not necessarily mean that they are interacting (See examples in [1]).

4.6 Program Generation

Empirically, several studies, including Keiburtz and McKinney et al [11], have demonstrated that using generation systems can improve both productivity of programmers and the reliability of the product they develop.

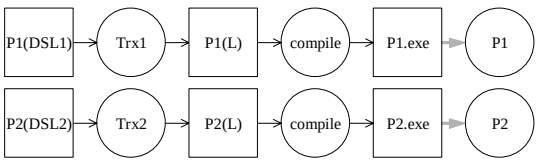
In addition, program generation can be used to significantly improve the performance of software systems. It is in this particular aspect of generation systems that multi-stage computation plays an active role.

The distinguishing characteristic of generative systems is not that they output programs as data entities (usually text), rather, it is the "animation" of these data entities into real, live programs in a later stage that makes these systems interesting. In other words, the combined architecture of both the generation and the "consumer" system is more interesting than each of them separately.

4.5.1 Generators and Domain Specific Language

As we have mentioned earlier, generation technology gives us a powerful mechanism compiling Domain-Specific Languages (DSLs).

In a system employing more than one DSL, relatively simple generators can be used to achieve almost the same effect as a moderately good compiler developed specifically for that DSL. The basic idea can be outlined as follows: For each DSL, implement a translation from this DSL to one, common language L. There should be a good compiler available for this language. For a pair of DSLs, this architecture can be sketched out as follows:



The key idea to this technique is to piggy-back on the power of the compiler for the original host language, and to use a fairly light-weight translation as the front-end Trx.

The advantage of this approach is combining both efficient execution of these languages with ease of implementation: Programming language technology can now make writing program generators much easier than was previously possible [20,5]. Not only that, having simple translations into one host languages can dramatically reduce the cost maintaining a large number of DSL in the same system, thus making the DSL approach an even more attractive option.

4.7 Other Examples

Runtime-specialization and machine code-generation is another interesting example of multi-stage computation [12]. It has recently been used to improve performance in operating systems software [4]. The key idea in run-time specialization is the dynamic generation of machine-code at runtime. This allows systems to dynamically adapt to changing conditions, and still operate efficiently. This kind of code generation has different requirements than high-level code generation. The major challenge in developing such systems is reducing the overhead involved in code generation, so that it does not out-weigh its benefits.

In this setting, an arbitrary number of stages is again possible, as the generated code can itself, in principle, generate new code too. Reflection is primitive, and is performed in the most natural way: by calling (or jumping to) the generated code.

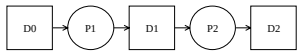
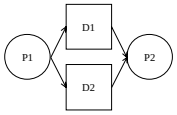
5 TOWARDS A CALCULUS FOR GDL

It is well known that the architecture of a software system can be described at more than one level. In this section, we present the basic machinery needed to relate GDL descriptions at different levels of abstraction.

In general, architectural descriptions can exist at two distinct levels, namely, the abstract and the concrete level [16]. The abstract level is generally used to describe architectural patterns, whereas the concrete level is used to describe particular systems. With GDL, various levels of abstraction are possible. In particular, GDL can be used (with moderate accuracy) to describe architectures both at the abstract level, and at the concrete level. The distinction between the two levels is primarily in the mapping between the elements of the GDL description and the system that it models. A GDL description (or a fragment there of) is concrete when there is a one-to-one mapping between each element of this description, and the atomic elements of the software system. This definition is relative, as it depends on what we consider atomic in our system. It appears to us that this decision, in general, must be made solely on pragmatic considerations.

A variety of abstraction steps (or "contractions") arise naturally in the context of GDL descriptions. We should point out that a branch of mathematics, called category theory, serves as a good guide in identifying these abstraction steps.

We will not make an exhaustive list, but here are some illustrative reduction steps that take us from one GDL to another more abstract one:

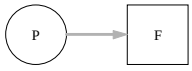
Original	Abstract
	
	

The first rule says that we can hide intermediate forms, and that is essentially due to the fact that we can compose programs. The second rule says that we can combine two (parallel) data items, thus hiding the distinction between them. Loosely speaking, these two rules correspond to the composition axiom and the presence of products in an underlying category.

The main observation we wish to contribute in this section is regarding the following definition: A GDL description H is called a decomposition of G if G is an abstraction of H. So, when we talk of an abstraction step from A to B, then we are also implicitly talking of a decomposition step from B to A. In the light of examples such as the above, it seems easier to think of these steps as abstraction steps when we are defining them formally.

6 A DUAL OPERATOR

The dual of reflection is reification, and as the Engage operator goes from an artifact to a program, a natural question to ask is whether doing the opposite, that is, going from a program to an artifact, is also a meaningful operation at the architectural level. On reasonable interpretation is the following:

Node	Means
	Dis-Engage P from system. The result is a data entity F.

Intuitively, Dis-Engage is a way for viewing and manipulating a program as an artifact. This can be a valuable operation for a variety of reasons, including alleviating the user's need to deal with I/O explicitly: The state of any program can simply be "saved" into a file, together with all its associated runtime datastructures.

If an appropriately high-level representation is used for programs, rather then this Dis-Engage would also provide the programmer with a mechanism for optimization code on based on information that may not be available to the compiler.

Dis-Engage can provide a restricted form of persistence. Orthogonal persistence, often seen in Object-Oriented Database systems such as GemStone for example (and more recently also in Java), is a more general mechanism. From a user's point of view, orthogonal persistence is probably the preferable mechanism. On the other hand, Dis-Engage provide finer grain control over the behavior of the architecture, and can, just like Engage, allow the system architect to make significant decisions at a higher level than is commonly done.

6.1 A Correctness Criteria

An important intuition to draw from the reflection/reification analogy is a (partial) correctness criteria for the behavior of implementations of the Engage and Dis-Engage operators: Starting with a program P, Dis-Engaging it as D, and then later Engaging D, should yield a program equivalent to P.

7 RELATED WORK

Architectural descriptions that are much richer than GDL have been studied [18]. However, GDL was intended to be skeletal, and at the same time, still representative of a full fledged, data flow based, architectural description language.

Roughly speaking, the three main elements of GDL, circles, boxes, and arrows, correspond closely to the processing, data, and collecting elements of Perry and Wolf [13]. In Allen and Garlan's terminology, circles and boxes correspond to the components of the architecture, and the arrows are connectors [3].

The discussion in this paper has also been relatively independent of the underlying execution model, that is, we have not restricted our self to a sequential or a concurrent or parallel model. This will probably also change when we try to further formalize this framework (see next two Subsections).

7.1 Dynamic Aspects of Architecture

Reflection is a form of dynamic behavior. Recently, there has been significant efforts towards finding a formal foundation for dynamic behavior in software architectures. While this paper does not address the issue of formalization directly, our goal has been to expose reflection as an important form of dynamic behavior.

Magee and Kramer have studied and formalized the problem of describing dynamic reconfiguration at the architectural level [14]. Their work treats system evolution of distributed systems. In this paper, our concern is also with a kind of dynamic behavior, but on a distinctly different aspect. In particular, we do not address the issue of distribution at all, but rather, the issue of the time dimension in a sequential, non-distributed framework. The difference in the two approaches is also reflected in the underlying programming language theory: while Magee and Kramer use the pi-calculus as the motivating formalism, our research has been primarily motivated by the lambda-calculus. There is no reason why our technique would not extend to a concurrent, distributed

setting. At the same time, this would be a non-trivial extension. However, this is an important step in order to investigate the ideas presented in this paper in the context of the WWW.

In the context of the Rapide system, another form of dynamic has been studied: Rapid can specify systems where the connectivity of the components changes over time [9]. In this study, none of the connectors are dynamic in that sense. Even the presence of the Engage operator is time-invariant. This becomes even more clear when we consider the underlying Real Virtual Machine discussed in the Introduction. We have not addressed this issue in the context of our framework.

Inverardi and Wolf have also studied dynamic behavior in software architecture using the Chemical Abstract Machine [8]. Allen and Garlan used a language based on CSP to formalize communication protocols between components. [2].

7.2 Multi-Stage Programming Languages

As we saw in this paper, reflection is needed in the meta-language to support multi-stage architectures. It's useful to note that this is a very common situation, even though not much attention is paid to it. As we pointed out in the introduction, most scripting languages are reflective even if they don't claim to be, because they implement the Engage operator.

Its useful also to note that requiring a language to be reflective can sometimes be trivial: On a machine where there is no distinction between data and program space, the jump command provides reflection. In fact, it provides most of the functionality of the Engage operator. In such a setting, ensuring that the over-all architecture is known a priori, and guaranteeing that the architecture is not mutated inadvertently become the more challenging requirements.

Recently, there has been significant interest in the programming languages research community in reflection and multi-stage computation [5,17,16,12]. A good introduction to the recent literature can be found in [20]. We are actively pursuing language support for effectively implementing such architectures. Our most recent efforts have been focused on formalizing the semantics and finding appropriate type-systems to allow safe (yet still powerful) multi-stage capabilities. This form of type-safety closely corresponds to knowing the over-all architecture of a multi-stage system a priori. However, one should be noted that a stronger connection is still desirable, as the type system is non-trivial.

Finally, and as an aside, we would like to point out that scripting languages don't need to be interpreted. An example of such a system is Oberon, where JIT compilation is used instead.

7.3 Future Work

Despite its simplicity, GDL can clearly express the essential distinction between interpretation and compilation at the architectural level. We have also found this to also be the case in the context of high-level program generation systems in general. One of our main goals is to be able to use it as a tool for identifying and categorizing distinct patterns of multi-stage computation at a high level of abstraction.

Engage and Dis-Engage specify putting applications on-line and taking them off-line. They do not, however, specify exactly how this behavior is implemented. It would be useful to be able to objectively say whether the vendor of a given architectural framework can support these operators or not. We intend to make the criteria for this requirement more objective as we develop more experience applying GDL. Employing a formalism, such as that employed by Moriconi and Qian [16], will be a good first step in this direction.

Finally, we have assumed that the issue of reflection is orthogonal to the presence of other forms of connectors at the architectural level. Based only on intuition, this assumption still remains to be validated. To this end, we are trying to better understand how the Engage operator interacts with other architectural connectors [3].

8 CONCLUSIONS

We have demonstrated that an important dynamic aspect of software systems is easily expressible at the architectural level. In doing so, we have also illustrated the variety and diversity of multi-stage architectures, starting from compilers and going to partial evaluation and program generation systems.

ACKNOWLEDGMENTS

Special thanks are due to Paul Hosom, Dino Oliva, Riccardo Pucella and Karen Ward for valuable comments on a draft of this paper.

REFERENCES

1. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1986.

2. Robert Allen and David Garlan, A Formal Approach to Software Architecture, *ACM Trans. on Software Engineering and Methodology*, vol. 4, no. 4, pp. 319-364, 1995.

3. Robert Allen and David Garlan, A Formal Basis for Architectural Connection, *Transactions on Software Engineering and Methodology*, July 1997.

4. Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu and Jonathan Walpole. Fast Concurrent Linking for an Adaptive Operating System, *Proc. International Conference on Configurable Distributed Systems* (ICCDS'96), May 6-8, 1996, Annapolis MD.

5. Rowan Davies and Frank Pfenning, A Modal Analysis of Staged Computation, In *23rd Annual ACM Symposium on Principles of Programming languages (POPL'96)*, St. Petersburg Beach, Florida, January, 1996.

6. David Garlan and Dewayne Perry, Introduction to the Special Issue on Software Architecture, *IEEE Transactions on Software Engineering*, April 1995.

7. Robert Glueck and Jesper Jorgensen, Fast Binding-Time Analysis for Multi-Level Specialization, *PSI-96: Andrei Ershov Second International Memorial Conference, Perspectives of System Informatics*, Lecture Notes in Computer Science, 1996.

8. Paola Inverardi and Alex Wolf, Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model, *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21 (4):373-386, April 1995.

9. David C. Luckham, James Vera, An Event-Based Architecture Definition Language , *IEEE Transactions on Software Engineering*, Vol 21, No 9, pp.717-734. Sep. 1995 , 21 pages.

10. Neil D. Jones and Carsten K Gomard and Peter Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993.

11. Richard B. Kieburtz, Laura McKinney et al, A Software Engineering Experiment in Software Component Generation, Proc. *International Conference on Software Engineering*, 1996.

12. Mark Leone and Peter Lee, Deferred Compilation: The Automation of Run-time Code Generation, *Technical Report CMU-CS-93-225*, Carnegie Melon University, December, 1993.

13. Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture, *ACM SIGSOFT Software Engineering Notes*, 17:4 (October 1992).

14. Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer, Specifying Distributed Software Architectures, In *Proceedings of the Fifth European Software Engineering Conference, ESEC'95*, September 1995.

15. David MacKenzi, *Autoconf: Creating Automatic Configuration Scripts*, Edition 2.8, January 1996. Available from <ftp://prep.ai.mit.edu/pub/gnu/>

16. Mark Moriconi and Xiaolei Qian, Correctness and Composition of Software Architectures, *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*, New Orleans, Louisiana, December,1994, pp. 164-174.

17. Flemming Nielson and Hanne Riis Nielson, A Prescriptive Framework for Designing Multi-Level Lambda-Calculi, In *Proc. Partial Evaluation and Semantics-Based Program Manipulation '97*, Amsterdam, June 1997. ACM Press.

18. Mary Shaw and David Garlan, Software *Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

19. Yannis Smaragdakis and Don Batory, DiSTiL: A Transformation Library for Data Structures, *USENIX Conference on Domain-Specific Languages*, October 1997.

20. Walid Taha and Tim Sheard, Multi-Stage Programming with Explicit Annotations, In *Proc. Partial Evaluation and Semantics-Based Program Manipulation '97*, Amsterdam, June 1997. ACM Press. Also available from <http://www.cse.ogi.edu/~walidt>