Communication Across Levels of Abstraction

Sameer Sundresh
sameer@sundresh.org
Department of Computer Science
University of Illinois at Urbana-Champaign

1 Introduction

Computer systems consist of interacting components operating at multiple levels of abstraction: processes, kernels, microprocessors—even the surrounding environment. Languages and tools for building and analyzing *systems*—as opposed to individual components—must account for the interactions amongst different levels. These interactions are often event-based in nature.

For example, a process may issue a `sleep()` request to the operating system; this is a *downward event:* a higher level of abstraction causes an event to occur at a lower level of abstraction. When the operating system decides that the process has slept long enough, it continues the process's execution from the point where it left off; this is an *upward event.* This idea generalizes to systems with several levels of abstraction, where some events may jump levels.

This article discusses an approach to modeling these downward and upward events within a framework known as *language level virtualization.* It is my hope that this approach will be useful for structuring languages and tools for building and analyzing software and cyberphysical systems.

2 Overview of language-level virtualization

This section introduces the basic concepts of the language-level virtualization framework. The subsequent sections use this framework to explore downward and upward events within systems involving multiple levels of abstraction. (A detailed definition of language-level virtualization is beyond the scope of this extended abstract.)

Language-level virtualization may be viewed as a variant of small-step operational semantics which includes rules that allow us to remove some operations and define other new operations within a context. This is a good match for modeling systems which consist of components operating at multiple levels of abstraction. For example, operating systems and virtual machine monitors usually trap attempts by applications to perform certain unsafe operations (perhaps emulating their behavior using other safer operations), while allowing safe operations (such as addition) to be performed directly by the CPU. While virtualization is a well-established feature at the operating system level, the same pattern of mediation can also be useful at other levels of software design. For example, one may wish to run a library written in C in a sandbox which prevents it from crashing or subverting its host application; or one may wish to run an untrusted script in a modified environment which only allows write access to certain resources; or one may wish to write certain components of an application in terms of operations defined in a trusted kernel, while disallowing direct access to low-level primitives (such as unguarded pointer arithmetic). All of these use cases can be addressed by building virtualization features into programming languages—*i.e., language-level virtualization.*

Basics of language-level virtualization. Consider a program represented as a *term, i.e.,*

$$Term ::= Head() | Head(Term, \ldots)$$

where *Head* represents a set of constants which may be used to tag terms. For example, $+(1(), +(2(), 3()))$ would be a valid term, assuming $\{+, 1, 2, 3\} \subseteq Head$. But what is the meaning or behavior of the term $+(1(), +(2(), 3()))$? We can define this term's behavior by (a) fixing an order of evaluation of subterms; and (b) defining the meanings of different kinds of subterms: $+(\cdots)$, $1()$, $2()$ and $3()$. Let's agree to evaluate terms via a left-to-right postorder traversal, and moreover let's tag *values* (non-executable results

of evaluating program terms) with $\langle\cdot\rangle$. Assuming 1(), 2() and 3() simply represent integer constants, the first three steps of evaluation would be as follows.

$$+(1(), +(2(), 3())) \longrightarrow +(\langle \mathbf{1}()\rangle, +(2(), 3())) \longrightarrow$$

$$+(\langle 1()\rangle, +(\langle \mathbf{2}()\rangle, 3())) \longrightarrow +(\langle 1()\rangle, +(\langle 2()\rangle, \langle \mathbf{3}()\rangle))$$

After the third step, we need to evaluate the term $+(\langle 2()\rangle, \langle 3()\rangle)$. But without a definition of how + works, we don't know what to do. In standard operational semantics, we could simply apply the static rules for + on integers. But with language-level virtualization, we would like to allow *different* definitions of + in different contexts. The solution is to produce a *request message* which is propagated outwards until the definition of + is encountered. This corresponds to sending a message *downwards* to lower levels of abstraction.

$$+(\langle 1()\rangle, +(\langle 2()\rangle, \langle 3()\rangle))$$
$$\longrightarrow +(\langle 1()\rangle, \mathsf{Request}(+(\langle 2()\rangle, \langle 3()\rangle), \Box))$$
$$\longrightarrow \mathsf{Request}(+(\langle 2()\rangle, \langle 3()\rangle), +(\langle 1()\rangle, \Box))$$

The two arguments to Request are (a) the term which we wish to evaluate—in this case, $+(\langle 2()\rangle, \langle 3()\rangle)$; and (b) the context into which to place the result—here we are requesting that the $\Box$ in $+(\langle 1()\rangle, \Box)$ should be replaced by the the result of evaluating $+(\langle 2()\rangle, \langle 3()\rangle)$. Notice that both arguments to + are *values:* this is because evaluation operates as a postorder traversal. Also note that the term $+(\langle 1()\rangle, \Box)$ is a (delimited) continuation, which specifies how to evaluate the rest of the program.

In order to complete this system, we need a way to *catch* and *service* request messages. There are several ways in which this may be done; the simplest is the Handler term, as below.

$$\mathsf{Handler}(term, [r()], [c()], [body])$$

Here *term* is a term which is to be evaluated; $[r()]$, $[c()]$ and $[body]$ are *quoted terms* which are not evaluated before evaluating the Handler term as a whole (this is indicated by the $[\cdot]$). If *term* evaluates to a request message, then the first and second (request and context) arguments are bound to $r()$ and $c()$, and *body* is evaluated within this augmented context. Assuming we have access to other standard programming language features, this would allow us to define a behavior for + within *term*, even if + is undefined or has a different definition outside of the Handler term.

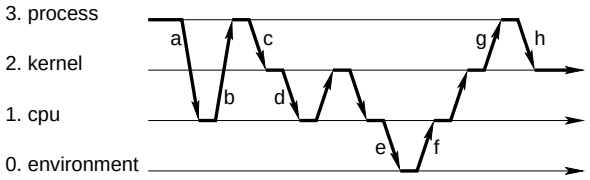

Figure 1: Example of three interacting level of abstraction.

A couple of related handler constructs are useful: (a) ConditionalHandler, which explicitly specifies a term head, and only catches request messages matching that term head; and (b) Allow, which specifies a set of term heads for which request messages are allowed to escape its context, along with a default handler for any other request messages. The ConditionalHandler construct allows us to introduce a new operation without interfering with existing operations; while the Allow construct allows us to restrict which operations may be used within a particular context.

Downward and upward events. Handler and ConditionalHandler terms perform two functions: (a) execution of a subterm at a higher level of abstraction; and (b) processing messages received from higher levels. A message receive event is a *downward event,* because the message traveled *downwards* from a higher to a lower level of abstraction. Conversely, continuing execution of a subterm at a higher level of abstraction is an *upward event.* The following sections explore these abstraction level transitions schematically.

3 Downward events

In order to discuss downward and upward events, it is useful to consider each level of abstraction as a thread, and turn the traditional message sequence chart on its side, so that the lowest level of abstraction appears at the bottom of the diagram, as in Figure 1. This figure depicts three levels of computational abstraction: 1. the CPU, which executes machine instructions; 2. the operating system kernel, which mediates access to resources; and 3. a user process. Additionally, we can view the computer as interacting with additional lower levels of abstraction, defined by the laws of physics. This particular choice of levels is simply one of many possible examples, cho-

sen here for reader familiarity. The arrows between levels indicate events.

For example, downward event (a) is induced by the process in the CPU. This corresponds to a *request* by the process to execute an instruction. The actual behavior of the instruction is defined by the CPU: the process can't just run on its own, as it is an abstraction built on top of the instructions provided by the CPU and system calls provided by the operating system kernel. Similarly, the downward event (c) induced by the process in the kernel corresponds to a *system call*—again, a request to perform some behavior, in which the lower level of abstraction (here the kernel) has complete control over the outcome. Notice that servicing event (c) involves additional CPU events like (d) triggered by the kernel; but these events are invisible to the user process. This reflects the intention that system calls are *abstractions.*

In the limit, everything that happens on a computer is simply an abstraction built on top of the laws of physics. We normally deal with higher-level abstractions, since the particular physical implementation details are not as important as the information processing properties of software. However, it is sometimes useful to explicitly model certain aspects of the physical nature of a computational system. For example, Figure 1 includes an environment with which the CPU can interact. In this case, a downward event might be an actuation or sensor sampling request.

A direct consequence of the layered language-level virtualization model is that we could install the upper levels (the software application) in both simulation and deployment environments. This raises an interesting question: *How can we determine whether two different contexts result in equivalent behavior for a particular class of applications?* A useful way to solve this problem may be to consider *equivalence* in terms of the request messages that are emitted by the context-application composite.

Finally, note that most downward events in Figure 1 *may* have corresponding upward event which continue execution at higher levels of abstraction (*e.g.,* a-b and c-g). However, this is *not mandatory,* as lower levels of abstraction do have full control over the execution of higher levels. For example, event (h) may correspond to a call to `exit()` by the process, which, naturally, never resumes.

Form the above discussion, we may conclude the following important properties of *downward events:*

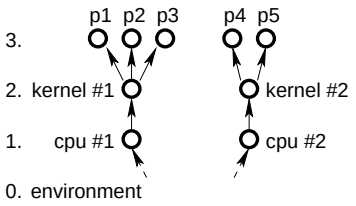

Figure 2: A tree of abstraction contexts. This is Figure 1 rotated 90° out of the page.

1. **Downward events are definitely yielding,** *i.e.,* execution at all higher levels of abstraction pauses when making a request to a lower level of abstraction. This is essential to allow lower levels of abstraction full control over higher levels' behavior. Yielding is an important property that distinguishes interaction between different levels of abstraction from interaction of autonomous processes.

2. **Downward events are possibly resumable.** This is embodied by the *continuation* argument of requests, which allows lower levels of abstraction to resume evaluation of a higher-level subterm. This property distinguishes downward events from nonresumable exceptions (as in Java or C++).

3. **Downward events may result in arbitrary changes to higher levels.** This is exemplified by event (h), the call to `exit()`. This property distinguishes downward events from function calls. Note that `call/cc` is not sufficient as a replacement for downward events, since `call/cc` does not provide a mechanism to allow us to *control* execution of the higher level's continuation (see the next section).

4 Upward events

While interactions amongst autonomous peer processes are generally *symmetric,* interaction between different levels of abstraction is, by definition, *asymmetric.* The arrows in Figure 2 all represent upward events: in each case, the destination of the arrow represents a term at a higher level of abstraction than the source of the arrow. An upward event is, precisely, the introduction of an (executable) term at a higher level of abstraction. Since the execution of such a term is defined and controlled by the lower levels of

abstraction, this results in an inherent asymmetry of upward vs. downward events. The properties of downward events described in the previous section do not hold for upward events; in fact the properties of upward events are essentially complementary to those of downward events:

1. **Upward events need not be yielding.** While the simple formulation of Handler in Section 2 does cause the lower level to yield on an upward event, this is not a necessary constraint. One could imagine a concurrent variant which continues evaluation at the lower level of abstraction along with the higher level. This would be useful for modeling systems consisting of multiple processes and processors, as in Figure 2.

2. **Upward events are definitely resumeable.** Any upward event must eventually lead to a downward event, because a term at a higher level of abstraction can *only* execute by making requests to lower levels of abstraction. For example, upward event (b) must eventually lead to another downward event at the CPU level, because following the execution of any (non-halt) instruction, the program must execute another instruction.

3. **Upward events may not autonomously modify terms at lower levels of abstraction.** The only way in which a term at a higher level of abstraction may affect lower levels of abstraction is by sending a request message which triggers a downward event. For example, the only way in which a process can terminate is to make a *request* which causes the operating system to terminate the process. There is no way that the process in Figure 1 could have terminated itself without making a downward request.

Figure 2 also illustrates how peer-to-peer communication can be modeled with language-level virtualization. By definition, peer computations should have a symmetric relationship, unlike the relationship explored above between different levels of abstraction. In effect, peers are all higher levels of computation built upon a common lower-level abstraction—the communication mechanism. Processes running on a kernel, processors in a computer, and autonomous robots in a common environment are all examples of this structure. This is, of course, why upward events should not necessarily be yielding: multiple peer terms may operate concurrently atop a shared communication abstraction.

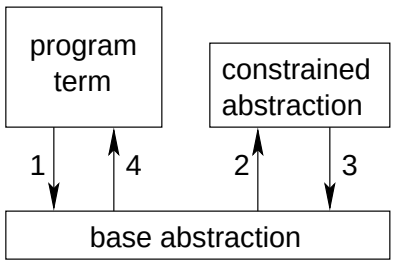

Figure 3: A constrained abstraction. For example, we may want to constrain the ways in which the physical environment can affect a computation.

Up until now, we have assumed that lower levels of abstraction are completely unconstrained in how they handle request messages. Upward events are not even guaranteed in response to downward events. While this stands to reason when looking at lower levels of abstraction strictly as defining the meaning of higher levels, it is too open-ended when considering interactions such as those between a robot and its environment. For example, we would consider an environment which, in response to a simple sensor read, completely changes the program running on a robot to be unreasonable.

This raises the question: *How can we constraint the ways in which an abstraction affects higher levels built on top of it?* One way to go about this would be to introduce new abstraction operations in place of Handler et al, which only allow certain operations to be overridden, and require certain upward events in response to downward events. Architecturally, this may be viewed as in Figure 3: a *base handler* traps request messages from higher levels (1); it then routes these message to a *constrained abstraction* (2) which advises what to do in response to the request (3). Since the constrained abstraction is a higher level built atop the base handler, it continually issues requests which can be trapped by the base handler (3), and translated into an upward event in the program term (4).

5 Related work

The hierarchical definition pattern assumed in this article is widely used throughout computer science. The two most relevant broad areas are (a) how ab-

stractions are presented at run-time in operating systems; and (b) how definitions and effects are expressed and constrained within programming languages. This section reviews several notable instances.

As mentioned in the introduction, operating systems universally use the *system call* model to forward requests from higher-level, unprivileged applications to the lower-level, privileged kernel and drivers. Even simple systems like TinyOS [13] running on highly-constrained hardware are organized as a stack of abstractions. And even the Exokernel architecture [4] follows this pattern—the key difference being that the exokernel exposes much lower-level abstractions to unprivileged code; with higher-level abstractions implemented at user level.

Interestingly, the idea of presenting low-level abstractions to safely multiplex resources was foreshadowed a quarter century earlier by the IBM CP-67 research virtual machine, which lead to the VM/370 hypervisor and its successors [2]. The notable characteristic of *system-level virtualization* is that the *same* code can run either directly on a machine, or on top of a *virtual machine,* which allows resources to safely be shared with other programs; this notion was formalized by Popek and Goldberg in 1972 [16]. Much more recently, system-level virtualization has seen widespread use on commodity architectures, thanks to hypervisors such as VMware [17] and Xen [1] (Xen in particular offers a *paravirtualization* mode similar in spirit to an Exokernel).

All of the above work focuses solely on system-level abstractions, as opposed to the internal architecture of an application process. TinyOS perhaps comes the closest, as its tight pairing with the NesC component-oriented programming language allows the same hierarchical structure to be used in application code or system code. Of course, several other systems pair a language with an operating environment, most notably Smalltalk [7], the Lisp machine [19], and the Java platform [8, 14]. Smalltalk and Java in particular are defined together with a a *virtual machine.* Unlike the VMs discussed in the previous paragraph, the VM is not modeled after an existing architecture; code *must* run under VM assistance. In addition to portability, one of the key features of the Java VM is security: untrusted bytecode can be run with high confidence that it will not have unwanted effects; instead, attempts at unauthorized operations result in security exceptions. Unlike system-level hypervisors, these language virtual machines do not provide an easy facility for creating a nested virtual machine instance which emulates restricted operations via a sequence of safe operations.

Run-time mediation via a VM is not the only approach taken to controlling computational effects. Type systems can also be used to statically verify that application code cannot perform unwanted effects. Indeed, the Java VM involves a static bytecode verifier as well as run-time checks. Monads provide a mechanism for embedding sequencing and effect information into the type system by representing control flow as data flow, much like continuation-passing style [15, 5]. While originally proposed as a formalism for programming language semantics, monads have since come to be the standard notion for introducing effects in Haskell [9], a statically-typed lazy functional language. Another approach is to extend a type system with *effect* information [6]. Wadler and Thiemann showed the close correspondence between monads and effects [18]. *Type-and-effect systems* have not yet seen widespread adoption in programming languages; nonetheless, compiler optimizations such as region inference implicitly use similar constructs [].

The continuations within our request messages are in fact *delimited continuations* [3]. While a standard continuation is a non-returning procedure which represents the rest of a program, a delimited continuation represents the rest of the computation of a *subterm,* and hence returns (unless the subterm diverges). Kiselyov et al. [12] showed how dynamic binding can be expressed in a delimited control framework. Their notion of *delimited dynamic bindings* coincides with the behavior of request messages (producers of delimited continuations) and handlers (the scope of dynamic bindings and consumers of delimited continuations) in language-level virtualization.

Finally, *Aspect-Oriented Programming* [11] also merits mention. An *aspect* is a piece of functionality which modifies the behavior at certain points throughout a program. Aspects consist of patterns that match at particular *join points* in a program (typically function call sites), and *advice,* which is code that runs before, after, or in place of the matching application code. As such, aspect are a new layer of abstraction interposed between application code and the base language. For example, the AspectJ [10] system allows aspects-oriented programming in Java. A key difference between AOP and system call abstractions is that aspects are not in full control of a program's execution; they are only invoked if the application happens to follow a matching code path.

6 Conclusion

Systems of components operating at multiple levels of abstraction have the interesting property that lower-level components *define* the behavior of higher-level components. This property leads to an asymmetry in events induced by higher levels at lower levels (downward events) and events induced by lower levels at higher levels (upward events). In particular, a downward event always results in higher levels of abstraction giving up control, while an upward event always leads to another downward event, thus keeping lower levels of abstraction in control. Interestingly, this means request messages sent to lower levels of abstraction (CPU instructions, system calls, etc.) have a control behavior that is actually the dual of standard function calls. The next step is to explore how we can use this model to formally analyze and constrain the behaviors of terms and contexts.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Symposium on Operating System Principles*, 2003.

[2] R.J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5), 1981.

[3] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160, New York, NY, USA, 1990. ACM.

[4] D. R. Engler, M. F. Kaashoek, and Jr. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM.

[5] Andrzej Filinski. Representing monads. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 446–457, New York, NY, USA, 1994. ACM.

[6] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 28–38, New York, NY, USA, 1986. ACM.

[7] Adele Goldberg, David Robson, and Michael A. Harrison. *Smalltalk-80: The Language and its Implementation*. Longman Higher Education, May 1983.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall, 3rd edition, June 2005.

[9] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, 2001.

[11] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, 1997.

[12] Oleg Kiselyov, Chung chieh Shan, and Amr Sabry. Delimited dynamic binding. *SIGPLAN Notices*, 41(9):26–37, 2006.

[13] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for wireless sensor networks. In *Ambient Intelligence*. Springer-Verlag, 2004.

[14] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 2nd edition, April 1999.

[15] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[16] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[17] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, 2001.

[18] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, 2003.

[19] Daniel Weinreb and David Moon. The Lisp Machine manual. *SIGART Bull.*, (78):10–10, 1981.