The *e* Language: A Fresh Separation of Concerns

Yoav Hollander Matthew Morley Amos Noy
Verisity Ltd., 8 Hamelacha St., Rosh-Ha-Ain 48091, Israel

December 20, 2000

Abstract

The *e* programming language enjoys widespread use in the microchip industry with applications to specification, modeling, testing and verification of hardware systems and their operating environments. Unique features of *e* include a combination of object oriented and constraint oriented mechanisms for the specification of data formats and interdependencies, interesting mechanisms of inheritance, and an efficient combination of interpreted and compiled code. Since the language is also extensible it serves as a living, industrial scale, implementation and application of the aspect oriented programming paradigm. This paper briefly describes the *e* language highlighting its novel features and their particular suitability to the task of hardware verification, and reports on our experience of aspect oriented programming in this intense commercial setting.

1 Introduction

With the advent of the object oriented paradigm there also came the realisation that despite its numerous benefits it is often the case that the unit of modularization reuse is not the class, but rather a slice of behavior affecting several classes. The research on *aspect oriented programming* (AOP) [1], *subject oriented programming* (SOP) [2], and the work on *adaptive programming* (AP) [3], represent almost a decade of attempts to address this problem. The influential AOP group [1] motivate the departure from the traditional OOP approach thus:

> "Objects have been a great success at facilitating the separation of concerns. [...] But objects are limited in their ability to modularize systemic concerns that are not localized to a single module's boundaries. [...] Rather than staying well localized within a class, these concerns tend to crosscut the system's class and module structure. Much of the complexity and brittleness in existing systems appears to stem from the way in which the implementation of these kinds of concerns comes to be intertwined throughout the code."

Mezini and Lieberherr [4] similarly observe that while object oriented techniques have given the programmer excellent data abstraction mechanisms, objects themselves are cumbersome when it comes to expressing aspects of behaviour that affect several data types. Conversely, OOP fails in naturally facilitating non-invasive extension mechanisms for layering new functionality over existing code. Essentially the same issue motivates the SOP community [2], and authors such as Lauesen [5], Wilde [7], Fisler [8] amongst many others.

Fundamentally, the AOP/SOP/AP quest is for mechanisms that will allow one to encapsulate concerns (features, or aspects) in modules even if they happen to cross class boundaries; the intent is to be able to apply different combinations of these modules to existing code rather than manually modifying it. This paper describes a solution to the separation of concerns problem which is represented in the *e* [6] programming and modeling language, which blends the OO paradigm with its own unique programming approach.

Unusually *e* has emerged from, and is today extensively applied in, a competitive industrial setting. Research directions such as AOP, SOP and AP emphasized the traditional software development model in which a basic product is subject to changes and additions over its lifetime, often by different teams. In our domain, that of *functional verification* of hardware, the software lifecycle is greatly accelerated, while the software is needed to support a product line of systems as well as a huge suite of tests.

A few words are needed here to explain the term *functional verification* as used by the design automation community. The target of this verification activity are hardware devices including CPUs and other microchips, network equipment such as routers, and systems combining hardware with embedded software. We shall refer to all these simply as "hardware" in the sequel. Functional verification is the process by which the correctness of the hardware design is established, the term *testing* having long been associated in this community with the process by which *manufacturing defects* are detected.

In functional verification, the hardware is not tested directly. Instead a description of the hardware written in a hardware description language (HDL) such as VHDL [9] or Verilog [10], is compared with a different, high level description of the hardware. Software written in ***e*** may therefore play a double role. On the one hand it is used for expressing the expected behavior of the *device under test* (DUT). On the other hand, it is used for exercising a certain functionality of the DUT.

Thus, there are at least two directions in which code proliferation occurs in ***e***: a collection of test environments describing the different DUTs in a product line, as well as a rich suite of test cases required for thorough coverage. This diversity exists even in a first version of a functional verification testbench. Thus, ***e*** evolved in a setting in which the need for mechanization of separation of concerns was even more important than in a traditional sortware lifecycle model.

A further aggravation of the hardware verification domain is that the testbench software merely plays a supporting role in the development of the hardware. The production of a testbench can commence when there is a design specification for the hardware, but it can only be used once a simulation model of the hardware has been developed. As a result, time constraints on the development of testbenches are even more acute than those on the hardware which in its turn has a very short time to market, and relatively short lifetime.

The need for high level, adaptable software for testbenches is also highlighted by the emerging market for reusable testbench components [17]. This new market follows the trend towards "systems on a chip" built of ready-made hardware components. Testbenches must match the high level of integrability of these components. It is not normally desirable for a test engineer to write code for adapting a library interface as would be done in more traditional kinds of software; what is required is a software component that is easily configured from without via constraints coming from the requirements of the environment in which the component is embedded. These constraints can be as simple as setting configuration parameters (architectural constraints) of the device being incorporated, or they may shape the data input space.

Similar conditions prevail in the domain of testing of non-embedded general purpose software systems: extreme time constraints, high adaptability requirements and a multitude of similar, but not identical, concurrently maintained testbenches. A major difference though is that the cost of post-release defects is so much higher in hardware than it is in software. Hardware verification is therefore required to be much more thorough, and hardware testbenches tend to be very large in comparison with software testbenches. It is illustrative to note that many hardware developers apply a rule of thumb that there must be one verification engineer for each design engineer.

The ***e*** programming language dominates the market of testbench automation. In our industrial experience we have observed testbenches that constitute some 10–500 KLOC. These numbers are dwarfed when compared to testbenches written in languages such as C, C++, or Perl, which can more than triple the code volume in order to achieve similar levels of design assurance. The common practice of extending the use of hardware description languages (which have few high level constructs, or modeling capabilities) for the purpose of testbench definition is even less immune to the maladies of large software systems.

The ***e*** language, while it is a general purpose programming language, has thus grown up specifically to address the flexible software demands of functional verification. This is explained in Section 2 which sets the context of ***e*** applications, its design rationale, and gives a global overview of the language. Then in Section 3 we describe those features of ***e*** that specifically address the separation of concerns problem—the ability to *extend* class and simple type definitions, as well as method extension. Section 4 discusses other unique features of the ***e*** language which have been shaped by the particular concerns of functional verification. These features add considerable facilities to the language over and above the extension mechanism discussed in Section 3, and include the important *when* (subtyping) inheritance mechanism, constraints, and simple data modeling capabilities. Section 5 concludes.

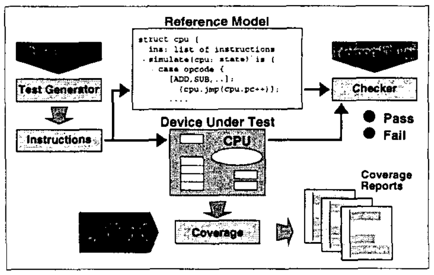
2 Functional verification and *e*

2.1 A typical verification problem

A functional verification program consists of a more or less detailed description of the functionality of a device, its operating environment, and the data transformations it performs. In general terms functional verification is predicated on the assumption that a detailed simulation model of a device has been implemented in a suitable hardware description language. Such descriptions are simulated in software or emulated in configurable hardware for the purpose of determining the precise timing properties of the design, as well as to judge its functional correctness. Given such an implementation of the device under test (DUT) a suitable testbench needs to be erected around the DUT in order to subject it to a large number of tests. The components of a typical testbench, or *verification environment*, for a siple CPU are displayed in Figure 1.

Instructions A key element in any verification environment is an adequate description of the data being manipulated—CPU instructions, in this case. Such descriptions typically do form natural classes of structured data—thus CPU instructions will be defined by some common elements such as opcode and addressing mode, but differences emerge (say) in the operands present causing a classification into *immediate* (e.g., the second operand, op2, is a two byte integer constant), *memory* (op2 is a two byte memory address), and *register* instructions (op2 is a four bit register index).

Test Generator This software ultimately creates a sequence of test vectors (of bits) to stimulate the DUT whether on-the-fly, or as a prelude to running a test. Setting aside the question of how to (randomly) generate instances of the data classes involved, the test generator needs to determine what are legal inputs, and what are not. To some extent a strong type system helps define legal ranges—it is easy then to generate a random four bit value for op2 in the *register* class. However via types it is difficult to stipulate, for example, that since register zero never holds a branch address an indexed branch instruction cannot have op2 equal to zero. Constraints, in the form of Boolean relationships over the fields of class definitions, contribute the necessary flexibility, relieving the programmer (or test writer) of much unnecessary programming.



```
struct cpu {
  ins: list of instructions
 simulate(cpu: state) is {
   case opcode {
     [ADD,SUB,..]:
       (cpu.jmp(cpu.pc++));
     ....
```

Figure 1: CPU Design Verification Environment

Reference model Commonly, but not necessarily, a reference model will be used to predict correct responses from the DUT for each datum input during a test. Typically functional verification works at the level of whole transactions rather than clock cycles of the DUT—in this case a *transaction* is initiated by injecting an instruction into the running simulation, and terminated some time later by observing a result on one of the device's output channels. Reference models thus do not need to be *cycle accurate* specifications of the hardware, just functionally accurate.

Checker The testbench must obviously check the expected results of the test against the actual computation. In CPU verifications there are typically two types of checker: a data checker that ensures that all instructions computed the correct results, and a temporal checker that monitors how each instruction is executed by the DUT. This latter activity calls for the definition of behavioural rules (e.g., via executable temporal logic [16, 12], or finite automata) that are run concurrently with the DUT, monitoring its state and progress.

Coverage Metrics that help the verification engineer decide how well the verification is progressing have to be carefully designed with reference to a *test plan*. For instance it may be required to test that the CPU responded correctly to an interrupt when a branch instruction was being decoded. The 'responds correctly' may be a temporal rule invoked under such circumstances, but the fact that this scenario occurred during testing would be entered as a functional coverage point. In a simple case one might be content to count how many times this combination of circumstances occurred.

Given a functional verification environment such as that envisaged above, tests will be devised to exercise the design. Sometimes these need to be very deterministic (e.g., in the early phases of the verification effort when one is testing basic functionality), but better coverage of the state space is achieved through random testing, especially when the 'randomness' can be directed towards particular goals. Often such goals are expressed as corner cases, particularly where functions of the device interact with one another. Principally it is for this purpose that the ***e*** language has been developed: random, directed test generation.

2.2 Factors influencing *e*'s design

Since its initial conception in the early nineties the ***e*** language has evolved to meet the needs of functional verification engineers. ***e*** is used to describe the DUT, its operating environment, its legal inputs, and its behavior over time. Specman, Verisity's flagship product implementing the language and runtime system, takes such a description and uses it to generate test inputs and drive them into the DUT, carry out temporal and data checking by monitoring the device, create coverage reports, and assist in debugging.

Even though ***e*** is a general-purpose programming language (in fact most of Specman is written in ***e***) its design has been geared towards the task of modeling and verifying hardware systems. This specific task imposed a number of important characteristics on the language.

Specialized Lingual Constructs These include *constraints*, for example, which provide an effective declarative mechanism for the specification of configurations and for guiding test generation, and *temporal properties* (also declarative) which are used to describe time based phenomena. Inevitably there are many hardware oriented primitive types and operators on them such as bit-access and bit-slicing (common HDL functions), as well as mechanisms for specifying parallel execution.

Simplified textual syntax The rich toolset that ***e*** provides to its user must be served in an easy to use, non-cryptic syntax. The design of the syntax and the semantics were also influenced by the reality that the principal users of the language are not software specialists but mainly hardware engineers who, in particular, may not be schooled in object oriented languages.

Performance The verification of hardware systems by means of simulation is, almost by definition, a slow process. Every hardware cycle in which many operations may take place in parallel is translated to a sequence of slow software steps. In addition, the quality of a verification process is highly dependent on its coverage level. Even a non-exhaustive verification process may execute for months on dedicated powerful servers. This is the reason why ***e*** has a very efficient implementation; typically, an instruction (such as field access or function call) in ***e*** is implemented in a similar manner to the equivalent instruction in C.

Compiled and interpreted code For reasons that are discussed in Section 3 below, there is a need when building testbenches to be able to load files which add new features, constructs, and especially constraints, on top of an extant code base. There is also a need for mixing those independently constructed additions in an unrestricted way.

On the face of it ***e*** is a lexically scoped, statically type checked object-oriented language with single inheritance. A *struct* in ***e***, just like a class in other programming languages, may declare fields and methods. Structs may also contain several unique declarative components, including constraints (affecting initial values assigned to fields), event definitions (for monitoring DUT behaviour), and temporal properties (checking protocols, etc.). The temporal and concurrent features of *e* are not discussed further here, but see [12].

A simple example, drawn from a verification environment for a packet switching device, demonstrates how constraints are used in *e*.

```
type packet_kind: [empty,short,long];
struct packet {
   i: int;
   j: int;
   kind: packet_kind;
      keep i < j + 1;
      keep j in [1..5];
};
```

The first of these two statements declares an enumerated type, the second declares a structured object with several scalar fields. The *keep*s are constraints that affect initial values assigned to the fields mentioned whenever an instance of this class is created—Specman resolves such constraints during a test run in order to generate a random, directed stream of data for the DUT. Constraints in *e* are linear functions over finite domains.

While the synthesis of constraint solving and object oriented programming in *e* is an interesting subject in itself, it is not explored further in this article which rather focuses on the language constructs that address separation of concerns. Thus, in addition to the simple inheritance mechanism (which is called *like* inheritance in *e*), the language provides a unique and powerful *when* inheritance mechanism, reminiscent of Chambers' predicate classes [11]. Moreover, any *e* struct can be extended in a later module: fields, methods, events, and constraints can be added to it, and method definitions can be modified or overridden. Interpreted files can be loaded on top of a compiled executable, possibly extending already-compiled structs. The extension capabilities are discussed in Section 3 below, the *when* inheritance mechanism is deferred until Section 4.

3 The aspect-oriented features of *e*

3.1 Motivation

One of the basic things people do with *e* is to define structs, and then extend those structs in later files (modules). This process is described through a number of examples below.

Example 3.1 A test specification, in its simplest form, simply adds a few constraints on top of an existing environment. For instance, taking the packet example started in Section 2.2, a test writer might want to specify: (1) For all packets, i should be equal to j, but no packet should be empty. To achieve this, she needs to write the following *e* file (call it `test1.e`):

```
// test1.e
extend packet {
   keep i == j;
   keep kind != empty;
};
```

This needs to be loaded on top of the current verification environment containing all the previously-defined files, and executed – in Specman this is achieved by hitting the "test" button to generate packets randomly, based on the augmented set of constraints. ■

Example 3.2 To take a slightly more complicated example. Assume that the test writer wants to specify a "special packets test": (1) For all packets, i should be equal to j, but no packet should be empty, and (2) the value j should be the same for all packets generated during a given test. For (1), all that is needed is `test1.e` as defined above. To implement (2):

```
// special.e
import test1;
extend sys {
   test_j: int;
   keep test_j in [1..5];
};
extend packet {
   keep j == sys.test_j;
};
```

The first line includes the definitions from earlier files so that they will be loaded (or reloaded) when the `special.e` test file is loaded into the verification environment. The second line extends the definition of another struct, *sys*, by adding a field and a constraint. Finally the packet struct is extended with another constraint that will have the effect of keeping j in every packet the same throughout the test. ■

This is feature-oriented programming: The file `special.e` implemented the "special packets test" feature by layering additional code over existing classes.

Note that this extension mechanism is quite unlike that of C++ or Java, for instance, where class definitions can only be extended via the subtyping mechanism. Thus, in particular, the struct *sys* above is modified by the additional field, but no subtype is created.

Example 3.3 Assume a test writer now wants to arrange that at the end of the test a count of all packets should be printed:

```
// counting.e
extend sys {
   !packet_count: int;
   finalize() is also {
      out("Total number of packets is "
          packet_count);
   };
};
```

This task calls for a new field to the top level struct *sys* (the **!** initialises it to 0, the default value of this type, rather than allowing the constraint solver to pick a random initial value), and an extension to an existing method of this struct which was already provided to execute cleanup code at the end of a test run. The *is also* appends code to the existing method body; *is first* and *is only* are alternative ways to extend (or override) methods.

Next (but in the same module):

```
extend packet {
   post_generate() is also {
      sys.packet_count += 1;
   };
};
```

This modifies another predefined method (one that is actually defined for all structs) which is executed when a new instance of the packet class is created. ■

Note that `counting.e` and `special.e` are independent. One could load either, both or none, and the "right" thing would still happen, even though both extend the same set of classes. Thus, each such file corresponds to a feature (or subject, in SOP parlance).

3.2 Orthogonal extensions

Because of the extreme time pressure associated with functional verification, a verification team will often split into several groups (or individuals), each group developing verification extensions to the basic environment. *e* was specifically built so as to allow multiple, independent groups to extend (add features to) some base functionality without bumping into each other and while allowing an integrator to later combine these extensions at will. In that, it is close in spirit to SOP.

One common practice is to divide work according to the chapters of the test plan written for the DUT. Assume, for instance, that one of the features of our packet router DUT is its ability to handle parity errors. One person might go ahead and implement the part of the verification environment which takes care of generation/checking/coverage of that feature, by writing `parity_errors.e` (say).

`parity_errors.e` might contain new fields related to modeling parity errors. (Where do we want the error? On the packet header or on the data? If on the data, on which byte of the data? And so on.) It might well impose new constraints connecting these fields and other fields of the base environment, new coverage points, or new methods. Using this file, a test writer will be able to devise a whole bunch of parity-error-specific tests (making use of the new fields and/or methods).

In parallel, another person might write a module called `timing_issues.e`, containing all that is needed to control the relative timing of packets. For instance, it would add fields to the packet struct determining the delay before injecting it into the DUT, and perhaps other fields for controlling whether or not the verification environment is to send packets into the ports of the device back-to-back, or on all input channels at once, etc..

Test writers might decide to import either file, or none, or both (in which case they would have the flexibility of influencing both parity error behaviour and timing behaviour). These are everyday needs for functional verification engineers which are adequately addressed by *e*'s *extend* mechanism. However, this mechanism does not resolve all orthoganality issues on its own. Programmers need to adopt some conventions to ensure that *any* two extensions will be loadable in *any* order. For instance, namespace conventions are needed to avoid name clashes if two aspects introduce fields of the same name; more insidious collisions may arise when methods are extended. The *is also* method extension used in Example 3.3 is relatively safe to use but *is only* (used to ocerride a function definition) may cause code to break. Such problems are ameliorated by the *when* subtyping mechanism discussed in Section 4.3.

3.3 Other uses of extension

Here are some other cases where the ability to extend a bunch of classes from the outside is very useful for a verification or modeling language:

Ease of debugging/analysis It is very easy to add specialized debug code to be loaded on top of existing functionality. For instance, such code might latch into "interesting" events so as to create a GUI visualizer for the DUT.

Design exploration It is very easy to change the configuration from the outside (typically via constraints), so as to see how the DUT behaves e.g. with a bigger internal buffer, more ports, fewer pipeline stages.

Replacing callback registrations Many programming environments have a facility for registering a callback, saying in effect "when X happens, call my routine Y". ***e*** does not have such a facility, nor does it need one. For each such "interesting" X, there should be a published method (which is often empty initially), which can be extended by any user in any other file. Providing such empty "hook" methods is a convention employed by Specman itself, and verification environment builders are also encouraged to follow it.

3.4 Extending in-place

Note that in all of the above examples, the user needs the extension to be done in-place (i.e., the original class has to be extended, rather than creating a new class). The issue is that if one creates a new subclass, then one needs to go back to all places where instances of the original class are created, and change them to create instances of the new class. This process is tedious, invasive and error prone. While it may be possible to address the need using the Abstract Factory pattern [15], the approach is cumbersome because of the need to manage a profusion of sub-classes, especially when dealing with multiple orthogonal extensions.

The problem of influencing what class gets created was well described by Kiczales [13] who observed that the usefulness of inheritance is severely limited in regular OO languages because they provide no mechanism to change from the outside the *types* of created objects. Kiczales' seminal work was very influential in the early development of ***e***, and is still very relevant today. His solution, *traces*, is different from the one adopted in ***e***, although traces could be implemented using constraints to influence *when* subtypes, and methods under them.

4 Extending `extend`

In the previous section we discussed ***e***'s *extend* mechanism which allows one to modify classes in place. Now we discuss other features of the language that complement this mechanism which we have found to be essential components of a high level verification language.

4.1 No pre-processor

Most of the experimental AOP languages coming out of AOP/SOP research are implemented as pre-processors: The tool takes program fragments (corresponding to different aspects) from separate files, and "weaves" them into a single program (say in Java). This approach would not suffice in functional verification. Firstly, for performance reasons, most of the ***e*** code is eventually compiled (at least for production runs). However, in each particular run, a different test, adding or modifying constraints, fields, or methods, will be loaded on top of this compiled environment. This process must be relatively quick as it happens very frequently during verification. Thus, a pre-processing style of implementing AOP is not practical. For example we need to extend some methods and add fields without invalidating (and re-compiling) the underlying environment.

Another reason pre-processing is problematic is debugging. When single-stepping through methods, the user would like to see the source of methods she has written, rather than the mangled results of a more complex pre-processing phase. Note, however, that there is a tradeoff here: Aspect-weaving-through-preprocessing may allow more powerful transformations.

4.2 The need for environmental acquisition

One of the main things one does in ***e*** is knowledge representation: trying to model the (concrete and abstract) "things" which influence the DUT. Here is a typical thing one might want to say: By default, all doors are green, except that car doors all have the color of the car. This can be captured as follows:

```
type color: [red, yellow, green];
struct door {
   color: color;
   keep soft color == green;
};
```

The 'soft' constraint declared here has a weaker semantics that the usual 'hard' constraints discussed above. A soft constraint can be ignored (overruled) by a hard constraint, that *must* be satisfied, applied to the same field by another piece of code. Such an overrule is used in the car struct where one captures the requirement that all the car doors have the same colour as the car:

```
struct car {
   color: color;
   num_of_doors: int [2..5];
   doors[num_of_doors]: list of door;
   keep for each in doors {
      it.color == color
   };
};
```

This problem in knowledge representation and programming, that of an object's need to acquire attributes based on its *role* within a composite rather than its parents alone, was described by Gil and Lorenz in [14]. The ***e*** language addresses this environmental acquisition problem to a large extent via constraints.

4.3 Using *when* inheritance

Using *when* inheritance one can easily express things like: All doors should print *bang* when their `shut()` method is called, except doors in hospitals, which should print *hiss*. This unique inheritance mechanism is more flexible than the regular (single or multiple) inheritance scheme, and is similar in spirit to Craig Chambers' predicate classes [11].

First it should be noted that ***e*** has the "regular" OO single inheritance. Thus one can write:

```
struct packet {..};
struct ethernet_packet like packet {..};
```

in order to specialise packets to Ethernet packets, say. However for modeling we recommend writing the same thing using *when* inheritance. This is done by explicitly specifying *determinant* fields—the fields which determine the "dimensions" used for inheritance. Determinants in ***e*** can be fields of Boolean or enumerated type, but not arbitrary predicates as Chambers suggested.

For example to capture various kinds of packet:

```
type packet_protocol: [Ethernet, IEEE];
struct packet {
   protocol: packet_protocol;
   data: list of byte;
   show() is {
      out("Packet length is ",
         data.size() . " bytes.");
   };
};
```

We can use the protocol field in order to specialise packets according to need. The *when* subtyping is implicit in this extension of the packet struct:

```
extend Ethernet packet {
   header: ethernet_header;
   show() is first {
      out("I am an Ethernet packet.");
   };
};
```

There is, in this case, only one field of a type that matches the 'Ethernet' determinant. The *when* subtype thus inherits all fields etc., from the virtual packet class, and provides its own methods and fields. The *when* subtyping can be expressed more explicitly thus:

```
extend packet {
   when Ethernet packet {..}
   when IEEE packet {..}
};
```

There are two main reasons why *when* inheritance is more appropriate than *like* inheritance for the kind of knowledge representation needed in the context of verification: Orthoganality, and constraints.

Orthogonality Using when inheritance, one can have multiple, orthogonal categories. Packets injected to a network switch may be characterised variously. E.g.:

- Ethernet, IEEE (IEEE1, IEEE2, ...) or foreign packets;
- simple or multipart packets (except Ethernet packets which cannot have multiple parts).

The non-orthogonality mentioned in the latter 'axis' can be easily expressed through *when* inheritance

```
extend Ethernet packet {
   keep simple_or_multi == simple;
};
```

(assuming that this is a field defined in the packet class).

Constraints Using when inheritance, one can specify constraints on the type, usually for directing random test generation.

```
keep protocol != IEEE;

keep soft protocol == select {
   20: IEEE;
   80: foreign;
};
```

The select constraint applied here is used to weight the random generator so that it favours foreign packets over IEEE packets. In fact such weights are often integer valued functions of data sampled from the DUT, thus allowing the generation of packets to be influenced, as need be, by the current state of the device. This is typically how *e* based functional verification works on-the-fly, directed towards corner cases (of the functionality) of the design.

5 Conclusion

We discussed our experience with the *e* language, and placed it in the context of other languages supporting separation of concerns. The main use of *e* is modeling and verification of digital hardware. This subject domain places a very high premium on separation of concerns, and hence this *extend* feature was built into the language from the beginning. We have also explained how the special needs of hardware verification further constrain the language design space. For instance, the disadvantages of pre-processing to combine aspects, and the need to provide programming constructs hardware engineers find familiar.

We have introduced other *e* features such as *when* inheritance and constraints, and have shown how these, in conjunction with extensibility, are important for knowledge representation. These, the reader will have observed, are all *declarative* language features. These work extremely well together in framing testing requirements so that writing tests using the language is, in our experience and that of our customers, many times easier than writing them in a language which does not have constraints and separation of concerns.

The flexibility afforded by separation of concerns is not without costs. For instance, because extensions of a single struct or method can be distributed over many files, users of the feature do report that they need tool support for viewing them all together, in the order loaded into the system.

One open research problem is the extent to which we can make verification abstractions into plug-and-play aspects. For instance a common verification abstraction is to use a *scoreboard* to record transactions injected into a DUT, and checked off once they are completed. Such abstractions vary in complexity from simple mechanisms to generate a clock signal for a DUT, to sophisticated packages for creating streams of CPU instructions. Clearly these are *aspects* of a functional verification environment, but presently they largely exist in the corpus of *e* programs as patterns to guide implementations. The question remains whether and how these can be expressed as adaptive programming components or *executable patterns*.

Acknowledgements Our thanks to Yossi Gil at the Technion, Haifa, for his help and advice in the prepapration of this article.

References

[1] Xerox Parc. The Aspect-Oriented-Programming website. http://www.parc.xerox.com/csl/-projects/aop/

[2] IBM. http://www.research.ibm.com/sop The Subject-Oriented-Programming website.

[3] Demeter. The Demeter / Adaptive Programming website. http://www.ccs.neu.edu/research/-demeter/

[4] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play components for evolutionary software development. In *IOOPSLA'98*. http://www.informatik.uni-siegen.de/ mira/public.html

[5] S. Lauesen. Real-life object-oriented systems. *IEEE software*. 76-83, March/April 1998.

[6] Verisity Ltd. Verisity openly licenses its #1 verification language. http://www.verisity.com/-html/licensee_release.html

[7] N. Wilde, P. Matthews and R. Huitt. Maintaining object-oriented software. *IEEE Software*. 10(1), 75-80, Jan. 1993.

[8] K. Fisler, S. Krishnamurthi and K. E. Gray. Implementing Extensible Theorem Provers. In *Theorem Proving in Higher-Order Logics: Emerging Trends*, INRIA Research Report, Sept. 1999.

[9] P. J. Ashenden. The Designer's Guide to VHDL. Morgan Kaufman Publishers, 1996.

[10] P. R. Moorby, D. E. Thomas. The Verilog Hardware Description Language. Kluwer Academic Publishers, 1996.

[11] C. Chambers. Predicate classes. In *Proceedings ECOOP'93*, Kaiserslautern, Germany, July, 1993.

[12] M.J. Morley. Semantics of Temporal *e*. In *Banff'99* Higher Order Workshop (Formal Methods in Computation), Ullapool, Sept, 1999.

[13] G. Kiczales. Traces (A Cut at the "Make Isn't Generic" Problem). In *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, pages 27–43. JSST, Springer-Verlag, 1993.

[14] J. Gil and D. H. Lorenz. Environmental Acquisition: A New Inheritance-Like Abstraction Mechanism. *ACM SIGPLAN Notices*. 31(10), 214–231, October 1996.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, 1995.

[16] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.

[17] J. Bergeron Languages run verification ecosystem. EE-Times, October 16, 2000.