Virtual Machines, Managed Code and Component Technology

John Gough
Queensland University of Technology
Brisbane, Australia

Abstract

Abstract machines have been used as an implementation mechanism for programming languages for more than thirty years. In their latest incarnation execution engines based on virtual machines offer "Managed Execution". The implications of this change go far beyond the superficial advantages of platform portability and go to the heart of software reliability.

In this paper it is argued that managed execution platforms such as the .NET Common Language Runtime and the Java Virtual Machine form the only reasonable basis for trustworthy component software. There is also an overview of current research in this field, including the vexed question of version evolution.

1. Introduction – A Brief History

The definition of abstract machines as a mechanism for *reasoning* about programs goes back to the dawn of computer science as we now understand it. The use of such machines to define program representation for compilers dates back at least to Wirth's "P-machine" in 1973[1]. The P-machine was a stack-based abstract machine intended to define an intermediate form for the "portable" *Pascal* compiler. Porting the compiler to a new execution platform required the creation of a new "back-end" that transformed the instructions of the abstract machine into the binary code of the target machine. However, a standard part of the porting process was to write an *interpreter* to simulate the P-machine on the new target, as a first step in the bootstrap process. This particular strategy had an unforseen consequence when Kenneth Bowles at *UCSD*[2] dispensed with the back-end and wrote tiny interpreters for the "*P-Code*" hosted on the multitude of incompatible microprocessors that were appearing in the early 1970s. The use of an abstract machine in this context was pivotal because of the simplicity of the interpreter that was needed. A further benefit was the extremely high code density that *P-Code* achieved — a very important factor in the days of tiny memories.

The P-machine was designed to support just one programming language, although it was expressive enough to support some other languages also. A later development of the concept was *U-Code*[3] which was used as an intermediate form for a number of mainstream commercial compilers for several languages on many target machines. With *U-Code*, as with *P-Code*, the operational semantics of the instructions are defined by an abstract stack-based automaton. It may be argued that the main role of the abstract machine at this time was one of *compiler factorization*. By using the common intermediate form, the problem of compiling N different languages for M different machines is reduced to producing N "front-ends" and M "back-ends" rather than $N \times M$ monolithic compilers.

A typical expression evaluation in such a system would be fetching the value of a 8-byte floating point field b of some structure a. Typical code for a stack machine of the era would be—

```
ldadr 'a'   // load address of 'a'
ldc.i4 4    // offset of field 'b'
padd        // pointer arithmetic
deref.r8    // load 8-byte real
```

Several things are worth noting at this stage: the front-end apparently needs to know the layout of the structure on the target machine, and the language is *untyped* except for the integer/floating point separation. Although front-ends based on this scheme produced "portable" code, the output was generally parameterized for each target machine. My own *Gardens Point* compilers were typical[4]. They required to know the following target information: alignment rules, argument assembly conventions, stack mark size and which of three possible methods were used for passing structures by value.

Abstract machines have also played an important part in the implementation of specialist languages. The *Warren Abstract Machine*[5] for the *Prolog* language is a typical example. There are at least two factors at work here. Languages that have dynamic aspects that make static compilation unattractive have routinely used interpretation for their implementation. The interpreter emulates an abstract machine that bridges the semantic gap between the source language and the instructions of the host machines. Furthermore, the use of an intermediate language factors the implementation problem when multiple machine architectures are targetted.

So far, according to this brief history, abstract machines have been used in two ways. A stream of instructions for an abstract machine may be used, transiently, as an intermediate representation for program texts inside factored compilers. Alternatively, the instructions for the abstract machine have been the end product of the compiler, to be subsequently executed by an interpreter when the program is run.

In 1989 a further possibility became apparent. In that year the Open Software Foundation called for proposals for an *Architecture Neutral Distribution Form* (*ANDF*) for computer programs[6]. The idea was that programs written in any source language would be compiled into an architecture neutral form, supported by the facilities of a standard environment. Each such program could be run on any machine that possessed an "*Installer*" utility. The installer would complete the compilation process by transforming the *ANDF* into native code for the particular computer.

It is relevant to note that the demands of *ANDF* necessitate a higher level of abstraction than is the case for *U-Code* and similar abstract machine forms. In this case there can be no "parameterization of output for the target machine" since the target is unknown at compilation time. Such matters as the layout of structures needs to be deferrred for the installer to decide. The code of our previous code snippet needs to contain *symbolic information*. Something along the following lines is needed, where we have arbitrarily assumed that the type of a is *AType* from module *Mod*—

```
ldadr Mod.AType::'a'          // load adr of 'a'
ldfld real64 Mod.AType::'b'// load field value
```

Consider the second instruction. Field names need to be qualified with the type of the field *and* the type of the structure to which the field belongs. Furthermore, the distribution form must contain type declaration *metadata* sufficient to allow the installer to lay out the data.

ANDF was not a success in the market, although most compiler people agreed that it was a stunningly good idea. At least part of the problem was the very high levels of symbolic information that *ANDF* contained. Major software vendors tended to be spooked by the fact that anyone with an *ANDF* distribution could reconstruct fully typed, readable source code with minimal effort.

The next significant step in our saga arose from the doctoral research of Michael Franz[7], a student at *ETH*, Zurich. This work was continued later at UCI. Michael gave a final twist to the installer idea by invoking the installer each time the program was loaded, rather than just once when the progam was installed on the machine. Nowadays, we would call the "installer" a just-in-time compiler (*JIT*). The "slim binaries" that were the distribution format were for a single language (*Oberon-2*) but were used unchanged on different architectures. The distribution form was extremely compact, despite the necessary presence of the metadata. The mind-blowing result from the research was that the time saved in reading the smaller binary from the disk more than compensated for the processor time to perform code generation within the installer.

And then in May 1995 language *Java* was announced by *Sun Microsystems*[9]. The distribution format of *Java* is part of the definition of the language, and is based on an abstract machine: the Java virtual machine (*JVM*)[8]. Java was always intended to be executed either by interpretation or by *JIT* compilation, and still is. The output of the *Java* compiler is one or more "*class*" files for every *Java* source file. Each such class file contains the "bytecodes" that are the instructions for the stack-based virtual machine together with the metadata that is necessary to allow true target independence. The new element that *Java* added to the abstract machine story was *verification*. Since all user-declared data is statically typed in every class file it is possible to use a lightweight "theorem prover" to check that the code is type-safe and hence memory-safe. Now, every legal *Java* program is necessarily type-safe so it may appear to be overkill for the *JIT* to re-check what the compiler has already guaranteed. This would certainly be the case if the class-files were only a transient intermediate representation between compiler phases. However, the generation of the class files and the invocation of the *JIT* are separated both temporally and spatially. The browser that downloads a "*Java*" class file as a component of an applet cannot trust that the bytecodes were generated by a correct compiler, nor that the code has not been modified either accidentally or with evil intent.

The *JVM* was designed with the goal of supporting just one language: *Java*. Nevertheless with more or less difficulty the *JVM* can support (type-safe subsets) of an alarmingly long list of languages.

The final event in our brief history took place in mid-2000, when *Microsoft* announced their *.NET* system. This system is supported by the Common Language Runtime (*CLR*), another stack-based abstract machine. It features a more expressive type system than *Java*, and is explicitly designed to support a wide range of languages — including those that are type-safe *and* those that are not. The type-safe ones can be verified by the *JIT* while the the type-unsafe ones skate on the same thin ice as any other binary program representation. The authoritative source on the *Common Language Infrastructure* which includes the *CLR*, is the annotated standard[10].

Generically we refer to the *CLR* and the *JVM* are being *managed execution systems*. They are managed in the sense that the final translation to machine code is controlled by the explicit type and accessibility declarations that reside in the metadata of the distribution form. *Data* is also managed in the sense that objects are allocated and later deallocated by a trusted garbage collector within the runtime. The absence of explicit deallocation (and re-allocation) of memory is a precondition of any proof of type- and memory-safety in such systems. An early comparison of the two virtual machines is the paper[11].

2. Why Managed Execution?

Both *.NET* and *Java* have become key technologies in the contemporary software world. This success may appear paradoxical since both systems suffer from the same issues that doomed *ANDF*. Despite the best efforts of the "*code obfuscators*" decompilation of code is still possible. We must conclude that there are other advantages that counter the risks of including symbolic content with the distribution. The things that are different between 1989 and 2005 are the "world-wide web", web services, and the emergence of component technology.

The importance of the web to the success of managed execution platforms seems indisputable. In the case of *Java* the possibility of writing browser applets drove the early uptake of the language, while web services featured in all of the early *.NET* publicity[1]. Despite all of this emphasis on easy access to the web, and the lure of software portability, it is contended here that the real importance of managed execution derives precisely from the fact that it is *managed*.

Here is the main claim of this paper —
Managed execution provides the only reasonable basis on which the promise of component technology may be realized.

2.1. Component Technology

The term *component technology* has acquired somewhat overloaded semantics so it should be clarified that in this paper the term is used in the sense of Szyperski[12]. That is — "*Software components are binary units of independent production, acquisition and deployment that interact to form a functioning system*". Components are thus —

- units of independent deployment
- units of third-party composition
- deployed in binary form

The key issue of component technology, in this context, is the software engineering means by which third parties may compose binary components to create programs that are robust and perhaps even correct. It might be added that a further challenge is to ensure that such programs continue to operate correctly in the face of the evolution of their component parts. This last element is discussed in Section 4.

The traditional means by which software complexity has been tamed is by the use of *abstraction*. That is, parts of the program are replaced by abstract representations, thus limiting the domain of analysis that is required to reason about the behaviour of the whole. Implicit in the validity of this approach is the naïve belief that the abstraction captures *all* of the interactions that propagate across the boundaries of the program parts. Many of the advances in programming languages in the last 30 years have been introduced to progressively increase the accuracy of the abstract representations and hence reduce the naïvity of the belief. Three brief examples will suffice to make the point —

- modular languages guarantee that functions may only be called with correctly typed arguments
- fully type-safe languages guarantee that pointer referents can only be of the declared type
- languages with declarative accessibilty control enforce the need to know principle

Every practising software engineer is familiar with the mayhem that results when these guarantees break down, as a result of memory deallocation faults for example.

Perhaps the most elaborate example of this approach to software design is the "design by contract" methodology incorporated into the programming language *Eiffel*. In this case software parts may be annotated with contracts in the form of preconditions, postconditions and invariants. These contracts are then enforced by a mixture of compile-time and run-time checks. The evidence seems to be that such mechanisms do indeed allow extremely robust and trustworthy software to be constructed.

All of this, so far, has been good news about which software engineers may be justly proud. The bad news is that all of the guarantees and safeguards described above are virtually useless in the context of component software!

Consider the simple example of a program component which depends for its correctness on the fact that a particular field of some object type may only be changed by the code at one program point. Such fields are safeguarded by being declared *private*. Unfortunately, if references to the enclosing object are accessible to other components the field may be mutated either through program error or by malicious intent. Declarative privacy counts for nothing in a binary component environment. Recall, for example, that the buffer overflow exploits that are a commonplace in vandalware seek to mutate function return addresses. In such cases the target location is so "private" that high level languages do not even have a mechanism to refer to the datum.

1 Indeed the last of the code-names used within Microsoft for what became *.NET* was the excruciating *NGWS*, an initialism for "next generation web services".

The issue is that all of the safeguards based on programming language mechanisms depend on the compiler for enforcement. With binary deployment of components, particularly those produced by third parties, neither the compiler nor the integrity of the deployment mechanism are a given. This is precisely the problem that managed execution uses *verification* to solve.

2.2. Safety and Security

It is important to distinguish between the separate concerns of *safety* and *security*. Memory safety and its prerequisite type-safety are necessary preconditions for forms of program analysis based on abstraction. We need to be able to reason piecewise about the behaviour of program components secure in the knowledge that components do not invalidate each others declarative invariants. This *safety* guarantee is precisely that which verified, managed execution provides.

Security is quite another matter. Both of the managed execution systems that we consider provide security mechanisms that regulate the security-relevant actions that particular components may perform. It is interesting to note that checking of security permissions requires a costly traversal of the whole chain of activation records, that is, a *stack walk*. This is necessary since it is not the permissions of a particular function that is in doubt, but the permissions of the complete chain of callers on whose behalf the function has been invoked.

Security is a separate concern to memory safety. The fact that the verifier has guaranteed the type-safety of an applet is of little consolation after the applet has reformatted the disk. Nevertheless, the enforcement of memory safety is a necessary foundation for a separate security mechanism. Consider the possible modes of attack against a stack-walking security permission checker. One exploit would be to falsify the permissions that a piece of code possesses. Another might be to falsify the call chain record by overwriting a return address. Both of these attacks are impossible in a verified, managed execution environment.

A more plausible security exploit involves managed code calling out to native (unmanaged) code. Once beyond the oversight of the verifier, anything goes. For this reason permission to invoke unmanaged code must be carefully guarded and sparingly granted in managed execution systems.

Some people find it somewhat artificial that discussions of such matters as type-safety are conducted in the language of conflict. We reason about "attackers" and try to remove "security vulnerabilities". This is neither a sign of paranoia, nor a preoccupation with fantasy games. If the invariants of a component are safe against an attacker with malicious intent, then the same invariants are safe against accidental violation by program errors in other components.

3. Some Research Issues

Managed execution systems, as they currently exist, enforce the constraints of the type declarations of the programs that they execute. This is sufficient to ensure memory safety of programs, and to ensure the absence of certain kinds of interference between components. To achieve even this is an important advance, however the kinds of invariants that can be guaranteed by such mechanisms are *syntactic* and *static*. There is a fascinating spectrum of open research possibilities that might broaden the range of program properties that managed execution might ensure.

There are also interesting research issues that have to do with the implementation of such systems.

3.1. Implementation Issues

Publicly accessible source code for both *Java* and *CLR* implementations exist, facilitating research on language compilers and *JIT* compilers. The question of which optimizations should be performed by each kind of compiler is still the subject of some experimentation, and could very well have a different answer for the *CLR* and the *JVM*.

Reliance on just-in-time compilation also brings with it the possibility, or should that be the challenge, of using the extra information available at runtime to generate faster code than is possible in an "ahead-of-time" compiler. The field of dynamic compilation and optimization is very active one with products starting to move from the laboratory to the mainstream.

A more basic kind of investigation involves the mechanisms of verification. It turns out that the algorithm specified for verification in *Java* can become computationally costly in some pathological cases. Alternative methods of verification based on "proof carrying code" seem promising.

Of course, it is always necessary to ensure correctness of the algorithms (and of their implementation) that managed execution relies on. The issues can be subtle. Here is a favourite example that nicely illustrates the subtle issues involved. One of the features of the Common Type System (*CTS*) of the *CLR* is the possibility to mark instance fields of structured types as *initonly*. The idea is that such fields are initialized at object creation, and are afterward immutable. This is a really useful feature in practice, since programs may be designed to use such fields to hold identity data, permissions and the like. In *C#* we mark such fields as *readonly* —

```
public class C {
    public readonly long serialNm;
    ...
    public C(long sn) { // Constructor
       serialNm = sn;    // assign immutable value
       ...
```

Compiling for the .NET Common Language Runtime[13] correctly warned that in the first release of *.NET* the *C#* compiler enforced this constraint but the verifier did not. As might be hoped, later releases of the *CLR* refuse verification to programs that attempt to mutate an *initonly* field.

In the description of what *initonly* means, the wording "... and is afterward immutable" seems perfectly clear, but is not the kind of rule that a verifier may check directly. What we need is an *operational* formulation of a test that checks this constraint. Here is a candidate set —

- *initonly* fields may only be set within a constructor for their enclosing type
- constructors may only be called as part of the creation of an object of the type, or as part of the creation of an object of a derived type

The "or as" part of the second rule is necessary, since when an object of a derived type is being constructed the (perhaps private) fields of the base type must be initialized by invoking a base class constructor on the newborn object of the derived type.

It turns out that the candidate rule set is insufficient, since it does not prevent a constructor from being called more than once on the same object. Here is an exploit[2] which mutates an *initonly* field, even in the presence of the candidate rule set —

```
public class D : C {       // D derives from C
  ...
  public D(C victim, long newVal) {
    ldarg.0                // push 'this' ref.
    ldarg.2                // push newVal arg
    call instance void C::'.ctor'(int64)
    ldarg.1                // push victim ref.
    ldarg.2                // push newVal arg
    call instance void C::'.ctor'(int64)
  }
```

The body of the constructor for type D is shown in Common Intermediate Language (*CIL*), where calls to constructors use the invariable name "*.ctor*".

D is a dummy class, we only use objects of this type to do our dirty work. The trick is that we have passed in the victim object of type C as an argument to the dummy constructor. The second rule above does not forbid us from passing this argument to a call of the base class constructor along with the new value for the supposedly immutable field. We cannot express this behaviour in *C#*, so the body of the above code snippet shows how it reads in *CIL* language. As a former, security-guru colleague of mine used to say "To be good at this stuff you need to have the criminal mind". In any case the third rule that is needed is —

- base class constructors may only be invoked on the newborn object within a constructor for a derived class

The first three lines of the constructor body in the code snippet are legal, and indeed are compulsory. These lines invoke the base class constructor on "arg.0", which is the location of the reference to the object under construction. The rather similar looking second call in the code is illegal according to this new rule as "arg.1" is the incoming argument, that is, the intended victim of the exploit[3].

3.2. More Expressive Type Systems

One approach to strengthening the guarantees of managed execution involves extension of the type-system. Another category of research involves the addition of such things as program assertions and protocol checks to the platform-enforced repetoire.

In essence, current managed execution systems enforce the declarative constraints of the type systems of their hosted programs. In principle any declarative aspect of a type system that is capable of being checked by an effective procedure might be added to the execution engine.

Here is a simple example. Languages such as *Ada* and *Pascal* provide for the declaration of *subrange* types, the values of which are restricted ranges of some whole-number type. It is usual for the compiler to ensure that every assignment of a new value to a datum declared to be of such a type respects the constraint. In a single language, known compiler environment consumers of such types do not need to perform range tests on values of the type. In a multi-vendor component environment no such trust could be justified, but a managed execution engine could statically guarantee enforcement of the value constraints. As it turns out neither of our example managed platforms provides for subrange types in their underlying type system, and it is hard to make a strong case for such an introduction, given the low cost of range testing at the point of use.

The more interesting issue of execution engine enforcement of program invariants such as pre- and post-conditions has received some attention. Nam Tran's doctoral research at Monash University has involved implementing *Eiffel*-like contracts with the support of a modified version of the "shared source" version of the *CLR*.

Almost all of the enforcement of managed execution systems have to do with *static* features of the type system. Accessibility constraints, conformance to the rules of sub-type

2 I am happy to share this exploit, since it does not work any more!

3 And of course if you are wondering, overwriting "arg.0" by "arg.1" doesn't get past the verifier either!

polymorphism, and fulfilment of contracts to implement named interfaces, are the kinds of things that are guaranteed. There are a whole range of dynamic issues that relate to correctness of component systems. These dynamic rules may be expressed in terms of *protocols*. Protocols specify such things as rules that certain methods may only be called if other calls have preceeded the call in certain allowed patterns. It is known that in some cases the rules cannot be described in terms of finite state machines. The understanding of such rules is an active area of research. An associated open question is — how can such protocols be enforced within a component framework that allows for third party composition of systems?

Finally, it may be noted that both of our example managed execution systems have announced enhancements to their underlying type systems to support *parametric polymorphism*, or "generics" as it is more commonly called. Sun Microsystems and Microsoft have adopted very different implementation strategies for this very significant enhancement. Sun has chosen to take a less efficient implementation mechanism, but one that leaves the *JVM* unchanged. Microsoft has enhanced the *CLR* with some additional instructions and lots of new metadata so that the *JIT* can specialize code for particular instantiations of generic types.

4. Version Evolution

Software evolution is one of the difficult issues of our time. As many an elderly *COBOL* programmer remarked in the late 1990s "Nobody expected this software to be around for so long". In the past the problems have been less for monolithic software, particularly where programs are statically linked. However for distributed software, and even more so for component software the problems of version evolution have become acute. This is a problem that has received a lot of attention within the Microsoft world, but no component system can ignore these issues.

The rest of this section summarizes some informal discussions. Credit for the key ideas of these proposals belongs to Chris Brumme, Patrick Dussud, Anders Hejlsberg, Jim Miller, Clemens Szyperski, Tony Williams and others within Microsoft. Raising of these topics publicly should not be taken as any kind of endorsement of the proposal by Microsoft, nor as a committment to implement any of these ideas in any future product.

4.1. The Perils of Registry

One of the most difficult problems for component based systems is that of version evolution. The problem is familiar to *Windows* system administrators. Pre-*.NET* component systems share dynamically linked libraries (*DLLs*) the identity and location of which is held in a global registry. A typical problem arises when a newly installed application brings with it a new version of a *DLL* that is used by an existing application. After installation of the new program some apparently unrelated program breaks. Re-installation of the broken program restores that program's functionality, but the previously installed program now does not work. This situation is colloquially known as "*DLL Hell*". It is caused by a failure of backward compatibility in the evolution of the library that is shared by the two applications.

Such a backward compatibility failure does not necessarily indicate incompetence on the part of the software provider. It is an unfortunate fact that programs sometimes depend on library behaviours that are outside that specified in the application programming interface (*API*), that is, they rely on undocumented behaviour. Furthermore it is sometimes necessary to modify even the documented behaviour, for example to eliminate a security vulnerability. In any case the problem is particularly difficult in systems that rely on global registries.

The problems of *DLL-hell* are lessened in the *.NET* framework, which provides for "side-by-side" execution. In this system the identity of loadable assemblies depends on a four-part version number, and a cryptographically strong originator signature. Every application may set a policy that allows it to choose between the latest version of a shared library, or to insist on one exact version. There are several intermediate policy possibilities. All of the various library versions may co-exist in the "global assembly cache" (*GAC*), and simultaneously executing applications may run different versions of the same library "side-by-side" as the name implies. More to the point for component based systems different components of the same application may use different versions of the same library. This possibility effectively uncouples the version dependencies of the different components.

The *.NET* system makes *DLL-hell* a thing of the past, or at least a thing of a rapidly receding present. Sadly however any belief that the version evolution problem is now fully solved is premature. The problem is that not *all* assemblies may be executed side-by-side. For example, if a library controls some unique resource of the machine then only one version may run concurrently. Such a library *must* be shared, and different versions cannot execute side-by-side. Worse still, as more of the operating system software migrates to managed code, more of the system-supplied objects will lock in particular versions of their defining types.

Conflicts between components may be indirect. Suppose two components depend on different versions of the same shared library, A say. Let us further suppose that library A is intended to to permit side-by-side execution. Unfortunately, if different versions of A depend on different versions of some second library, B say, then if B does not support side-by-side execution, then neither can A. It seems that as *DLL-hell* recedes, *GAC-hell* looms on the horizon.

4.2. Platform and Library Types

It is likely that the problems of version evolution are fundamentally intractable, but there are some interesting approaches for at least *managing* the problem. In particular, it is important to lessen the *domino effect* caused by chains of dependencies between library components as described above.

One possibility receiving some debate currently involves adding a new declarative attribute to type definitions. At its simplest the idea is to mark every type as being either a *platform* type or a *library* type. Each denotation implies a contractual obligation as to future evolution. Library types are free to evolve between versions, but guarantee that different versions will be able to execute side-by-side. Platform types are bound to a much higher level of compatibility. Applications do not have a choice as to the version of a platform type that they use. As the name implies, the software must use whatever version of the type that the platform supplies, and all components on the machine will use the same version.

Platform types are not capable of side-by-side execution, either because they depend on a non-sharable resource, or because they are locked by a dependency on the underlying operating system or even the *CLR* version. For example the character string type *System.String* must be a platform type, since the type is built in to the execution engine.

Dependencies The constraints on the dependencies between the two categories of types can be easily deduced.

Library types may freely depend on platform types. They may use platform types in the implementation of their own behaviour, and may expose platform types in their visible interface. Any such dependency does not constrain the evolution of the type, nor does it add further dependencies to users of the type. Library types may also depend on other library types. This will make the type dependent on a particular version of the other types. If the “other” library types are not exposed in the visible interface of the type, then the *users* of the type do not become contaminated by spurious version dependencies. In a typical scenario several library types would be defined in a single library. These types will depend on each other, but will evolve together, compatibly, as their containing library evolves.

Platform types, on the other hand, are bound by a stricter regime. A platform type may depend on library types in its implementation, if necessary it can deploy with the library version that it requires. However, a platform type must never expose any library type in its visible interface. Thus for such types every public field, every formal argument of a public method and every return type must be a platform type. A platform type cannot derive from a base class that is a library type, nor may it implement an interface that is a library type. Rather less obviously platform types may not allow library methods to escape in (for example) arrays of *System.Object* or onto a system clipboard.

Using Platform and Library Types The separation between platform and library types only becomes significant at the boundaries of software components. Creators of components will expect their components to have to use whatever version of the platform types that the platform offers. But since every other component of the application will necessarily use the same version, there is no possibility of conflict.

Components may use library types of their own choosing to implement their own behaviour. They will thereby acquire a dependency on a particular library or libraries, but they can deploy with the version of the library that they depend on, and lock down that exact version if necessary. Provided that the component does not expose the dependency to its users, there will be no conflict if another component has locked in a different version of the same library type. In effect, these constraints mean that components must interact and communicate using only platform types. If, contrary to this advice, components communicate by exchanging library types then the components must agree to use the same exact library version. This clearly places a very strong limitation on third-party composition of such components.

Designing Platform Types Platform types appear to be more versatile, so it may appear attractive to make as many types as possible platform types. This is not a good idea. The evolution of platform types is necessarily slow and painstaking. Platform types have contracted to maintain an almost impossibly high level of backward compatibility. When a new version is released every existing application will have to use it ... so it better just work. We may conclude that an extremely high level of quality assurance will be needed to maintain platform types, and this will be expensive.

The challenge is that the required compatibility for platform types is not just at the level of using the same method signatures in the binary form, but at the *behavioural* level. Every aspect of documented behaviour must be maintained in new versions of the type. Conversely, *any* aspect of undocumented behaviour visible to users provides the opportunity for user code to be broken by future version evolution.

Choosing to create a platform type should thus be approached with some caution. Since the users of the code will be very upset if the behaviour ever changes the design has to be right first time. And then, having designed it right, the company needs to expensively maintain the type for as long as it plans to stay in business.

4.3. But Will it Fly?

The separation of types into categories according to the style of their evolution may or may not find its way into the type systems of the mainstream managed platforms. Nevertheless, the very idea of such a separation is an important tool in tackling versioning issues in component systems. The concept, and the design rules that flow from it, have a wider applicability to all large scale systems for which it is necessary to upgrade subsystems piecewise. In that case, even if all the sub-systems come from a single vendor, new subsystem versions must interwork with other subsystems that will be replaced at later times. The key lesson is to tightly control the evolution of the types that cross the subsystem boundaries.

Finally it may be noted that within the component world, with third-party composition of independently developed components, the type-safety guarantees of managed execution are critical to controlling the versioning problem. It is also relevant to observe that managed execution systems restrict the visibility of implementation artifacts, and thus make it less likely that the user of a platform type can depend on undocumented features of a particular version of the type.

5. Concluding Remarks

This quick overview has reviewed the historical context in which abstract machines have morphed into the currently popular managed execution systems. The recital of the history goes some way towards explaining how the current systems came to use abstract machines to represent program behaviour. Nevertheless, the notion of managed execution (or alternatively *metadata mediated execution*) and the use of abstract stack machine representations are independent. In fact it may be argued that some slight advantage might be gained by using a program representation other than the instructions for an abstract stack machine. Even the advantage of high program density traditionally claimed for stack machines seems dubious, given the significant volume of metadata that must accompany the bytecodes in these systems.

Irrespective of the choice of instruction set, managed execution systems with their reliance on symbolic metadata, provide representations of program behaviour that are suitably abstracted from the details of any particular machine. In principle they provide for a level of software portability that goes beyond anything previously achieved. This is a one reason for the importance of such systems, but not the most important.

The real importance of managed execution systems, and their critical role in the future of software development depends on the fact that they are *managed*. Managed execution provides for the enforcement of type- and memory-safety in environments where the integrity of neither the originating compiler nor the deployment mechanism may be guaranteed. Memory safety is, in turn, the guarantee that is required to ensure lack of interference between program parts in a component framework. The enforcement of type-system invariants is also an essential factor in managing version evolution.

References

[1] K. V. Nori, U. Ammann, K. Jensen, H. H. Nageli and Ch. Jacobi, "Pascal-P Implementation Notes" Ch. 9 in D. W. Barron (Ed), *Pascal – the Language and its Implementation*. J Wiley, 1981.

[2] K. Bowles, *Beginner's Guide for the UCSD Pascal System*, Byte Books, 1980.

[3] D. R. Perkins and R. L. Sites, "Machine-indpendent Pascal code optimization". Proc. 1979 SIGPLAN symposium on Compiler Construction, ACM, 1979.

[4] K. John Gough, "Multi-language, Multi-target Compiler Development", JMLC, Linz Austria, March 1997. Also in *Modular Programming Languages*, H. Mössenböck (Ed), LNCS No. 1204, Springer Verlag.

[5] Hassan Aït-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[6] S. Macrakis, "From UNCOL to ANDF: Progress in Standard Intermediate Languages." Technical Report, Open Software Foundation Research Institute, 1993.

[7] M. Franz, *Code Generation On-the-Fly: A Key to Portable Software*. Doctoral Dissertation No. 10497, ETH Zurich, March 1994.

[8] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, Reading MA, 1997.

[9] Sun Microsystems, "Java Technology: the Early Years" http://java.sun.com/features/1998/05/birthday.html

[10] J. Miller and S. Ragsdale, *The Common Language Infrastructure Annotated Standard*. Addison-Wesley, New York, NY, 2004.

[11] K. John Gough, "Stacking them up: a Comparison of Virtual Machines". Australian Computer Systems and Architecture Conference (ACSAC-2001), Gold Coast, Australia, February 2001.

[12] C. Szyperski, *Component Software: Beyond Object Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[13] J. Gough, *Compiling for the .NET Common Language Runtime*. Prentice-Hall, Saddle River, NJ, 2002.