International Journal of Parallel, Emergent and Distributed Systems
Vol. 1, No. 1, 2 2005, 1–17

On the use of metatypes for safe embedded operating system extension

Gilles Grimaud Yann Hodique Isabelle Simplot-Ryl

L.I.F.L. CNRS UMR 8022
Université de Lille I, Cité Scientifique
F-59655 Villeneuve d'Ascq Cedex, France
email:{grimaud,hodique,ryl}@lifl.fr
()

We present in this paper an extensible system for small secure embedded devices. We advocate the use of a typed intermediate language as a transformation of various high level languages. We present an extensible type system that unifies in a unique hierarchy some type systems from various source languages and ensures integrity and confidentility. To increase execution efficiency and use flexibility, we propose a dynamic binding mechanism that allows the programmer to describe the bindings of his code without breaking the type system. We also design the whole type system so that future addition of new kinds of objects has as little impact as possible.

Keywords: embedded system, extensibility, safety, type system.

1 Introduction

We present in this paper a dynamic binding mechanism which, once combined with a structured type system, allows system extensibility with a high level of security. This work is dedicated to embedded systems and, more precisely, we focus on systems for small devices that need to be highly secured, like smart cards for example. These hardwares were historically running a single application burned in ROM. Because the fastest they react the fastest they get (or keep) customers, software providers no longer want to "re-burn" an embedded system (a smart card) when they want to offer a new service. This is particularly the case in cellular phone related smart cards and their applications. To allow software deployment in a previously existing embedded system we must include a mobile code support (code loader, binder, and runtime). Nevertheless, in order to avoid lacks of performance obviously induced when piling up abstraction, it is consequently necessary to raise the operating system up to the highest level by enabling software providers to design the abstraction (components) they need closer to the hardware : in other words, operating system must be extensible. Thus, what we call extensibility is the ability of an embedded system not only to dynamically load new applications but also to load new components of the system itself. We want to achieve extensibility with a high security level: the word “security” is used here in the sense of data integrity and confidentiality. The system brings the applications the guarantee that other applications will not have the possibility to alterate nor access their data. This may be achieved by using strong typing and some simple rules (for example pointer arithmetic has to be forbidden).

Usual component bind consists in a clean calling convention. It is the case for example when you are linking to C modules. It can be more complex when modules (or components) are loaded dynamically, though. In this case binding consists in (i) loading the required component in the runtime environment (in memory) and then (ii) modifying the caller software by injecting the appropriated calling convention towards the loaded component. Some Operating System toolkits like Think [11] provide a ”binding factory” scheme which allows dynamic interception of any software call. Nevertheless, Operating Systems and for example kernel components define dedicated binding methods to achieve building links with the underlying hardware. This is for example accomplished in C by defining macros in place of functions declarations in the prototype file. In this case, the compiler, and more exactly the pre-processor, instantiates dedicated binding in the “caller” component according to the “callee” specification. However this strategy is unsuitable when the software is to be deployed dynamically, since the “caller” may be already in production.

Paper overview. Section 2 argues the use of an intermediate language for embedded systems and presents a type system that allows extensibility and security. Section 3 describes a dynamic binding mechanism used to specialize bindings at loading time. The mechanism, combined with our type system, gives a lot of flexibility without breaking the security properties. Section 4 proposes some experimental results before conclusion.

2 System design

The fiability is here a synonym for integrity and confidentiality, which are the minimal properties we want to provide to the loaded applications. Most of the time, these properties rely on correct typing of applications and elementary rules that are sometimes expressed in informal specifications as “it is not allowed to cast” or “it is not allowed to forge a pointer”.

In order to offer extensibility facilities but still fiability, we advocate the use of an intermediate language as target for various high-level languages. The intermediate language should have the following characteristics:

- it should be compact,
- it should be strongly typed to help at "security" verification,
- it should offer facilities to implement embedded verification mechanisms: the verification has to be performed at bytecode level for all source languages in order to avoid redundance of efforts. Moreover, an embedded system needs to be autonomous in terms of verification. The system fiability does not rely on the application provider nor on a third-party.

2.1 *Intermediate language*

Since the apparition of Java, there has been a lot of research works related to the use of intermediate languages, in particular in the area of embedded systems. Intermediate languages are interesting because they can be high-level enough to express properties such as types, that are lost in assembly code, and low-level enough to be almost executable as is. In general purpose computing, intermediate languages are now frequently used (*e.g.* the Java Bytecode, CLI [2] for .net) but modern embedded devices support also mobile code with Javacard [4] and Multos [7] for example.

It is of a great help in embedded systems to have access to high-level properties, since most security policies rely on these ones, and embedded devices are wanted to be highly secure. The intermediate language can be closely bound by the type system it uses. For example the object system of Java is expressed deeply in the Java bytecode, which has a semantic similar to what can be found in the Java language. It is necessary to be able to map directly properties such as types, so that they can be used in embedded verification after loading the code into the device. Anyway, as we want to support many high-level languages, we cannot just specialize the type system to a particular one.

Another advantage is that the intermediate code is more compact than its native equivalent, so that it is easier and less expensive to transfer applications to typically slow devices. As we want our intermediate language to be efficient, we prefer to avoid the use of a virtual machine as much as possible. That is, we want to have the possibility to compile our intermediate language into native code. This is particularly relevant for the "kernel" part of the system: those primitives are used a lot, and should be as efficient as possible. As for applications, we let the possibility to have very high-level function calls, that will slow down the execution (in less critical parts), but will make the intermediate code more concise. We make here the choice of using a memory-consuming method where speed is absolutely needed, and save high loads of memory where it can be afforded.

2.2 *Extensibility and efficiency*

In this part, we will have a closer look at what we expect from a type system. Since our intermediate language must be able to support object concepts, its type system has to be extensible by the user (creating a new type by inheriting from an existing one is allowed). But it should also not be bound to a particular source language, so that the definition of the type system should be extensible itself: we want to be able to express complex mechanisms, such as multiple inheritance, without applying heavy transformations over what is expressed in the source code. Here we exhibit another advantage of manipulating an intermediate language: any verification can occur on the bytecode, independently of the source language. This helps having a common infrastructure for code verification.

Given that, it seems natural to use an object type system, but we want to avoid the usual drawbacks of this approach. Some fully object-oriented languages, as Smalltalk, pay the price of the object layer even when it is not really needed (from a low-level point of view). For example, when performing an addition between two bytes, it's a pity not to generate a single `ADD` instruction, that would be executed directly by the microprocessor. The object concepts are highly time-consuming, mostly in the most basic part of computation. Since the embedded systems are usually low-resources devices, we need to avoid that situation.

In conclusion, we wish a system that can support and manage any object concept that can be found in high-level source languages. We also want all of them to be unified in a single hierarchy that can be extended as much as needed to accept new elements.

For managing the different aspects of our system (highly extensible, focused on efficiency), we chose to introduce a notion of metatype in our type system. The metatypes help us to classify our type system by grouping similar behaviours. They are in charge of inserting an abstraction layer over the object mechanisms to be installed, and also to make the compilation of the intermediate language instructions more flexible, in order to allow the generation of highly optimized machine code when needed.

2.3 *Metatypes as types*

Metatypes can be built naturally enough via metaclasses (this mechanism is often used to build reflexive systems, like Lisp for example [3,5]), that is, classes whose instances are classes themselves. This fits well with the existing type system and does not introduce a separate system (which would break our unified representation). We want to separate different concerns about type behaviours in different metatypes that are in charge of defining them (for example the distinction between *value* objects and *reference* ones would be expressed there).

The primary purpose of the metaclasses is to provide an abstraction over the type-related mechanisms (mostly object-oriented ones, but not only) that are really implemented at the classes level. For example, the existence of the concept of instance, or the existence of dynamic bindings should be defined in a metaclass. A class that would be an instance of such a metaclass would naturally provide the requested mechanisms.

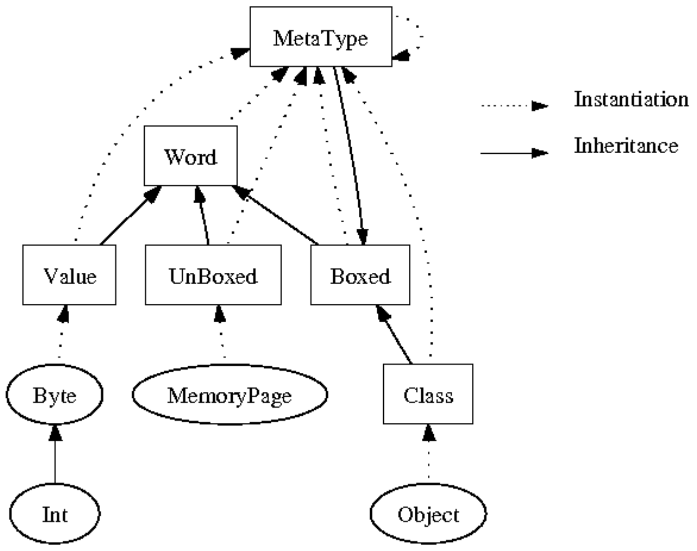

Figure 1. Metatypes hierarchy

Figure 1 presents an example of metatype subsystem (square boxes) in which immediate values and objects coexist. In addition, we introduced another metatype `UnBoxed`, that defines some types whose instances are accessed by references but are statically typed. `Word` is the local top for the metatypes branch (and will be located under `Object` in the full graph). `Value` is the kind of classes whose objects are not references: that is, they are not pointers to more complex structures. `Boxed` and `UnBoxed` are the kind of classes whose objects are references and that keep track (`Boxed`) or not of their dynamic type. `Class` is roughly the same as `Boxed` but does include a notion of single inheritance. Thus the lookup behaviour is fixed, and `MetaType` is the class of all the metatypes. In addition, the diagram presents examples of metatype instances in round boxes (the instantiation relation is represented by dotted arrows, whereas inheritance is represented by plain ones).

Doing this, we obtain a clean and well-ordered type system that supports different subsystems without interference.

We rely on a single-inheritance mechanism for metatypes that will be modeled from `Object`. The reasons for that are:

- we need to insert our metaclasses into the inheritance graph, since they are classes themselves,
- we need to express different behaviours for common properties (which can be mapped on dynamic bindings),
- as most of nowadays high-level languages support that kind of object behaviour, we will need the `Object` branch anyway,
- single inheritance seems sufficient for what we want to put into metatypes.

The concepts expressed in metatypes are the following:

- how the instances of the classes are structured (for example immediate values or references, composite objects or not),
- what it means to inherit from a parent (for example being able to overload a parent's method),
- what it means to receive a method invocation on an instance (static calls *vs* dynamic calls).

For example the concept of dynamic bindings (also called *virtual methods*) will be defined only for the instances of `Boxed` (until such a feature is needed somewhere else). On the other hand, for instances of `UnBoxed`, we have only statically resolved method calls (since there is no notion of dynamic types).

2.4 *Design details*

Our common definition of inheritance (shared by any instance of `Word`) is really minimalist: a derived class can be used as a parent class. This can be done through the use of a method `isAKindOf()`, whose behaviour is specified in `MetaType`, the common metatype for all metatypes. This gives a unique upcast system for the whole hierarchy, where traditional languages like Java behave differently for "real" objects and *primitive types*

It is interesting to note that we define inside the metatypes concepts that are usually part of the typing system itself.

A key point in the use of metatypes is also that there is a close relation between the notion of inheritance (at the class level) and the one of instantiation (at the metaclass level). It has to be strictly forbidden for a class `B` to inherit from a class `A` if the metaclass of `A` is not a superclass for the metaclass of `B`. Given that, we have a clean separation between incompatible types. One could imagine adding one day a new kind of classes that would support a notion of multiple inheritance, and the previous property would prevent its instances from inserting themselves into the "single-inheritance" branch. By keeping things isolated, we ensure that the properties we proved on a type system will remain valid once it is extended by adding a new kind of types. The separation of type subsystems has to be ensured in each metatype, since the latter defines its own notion of inheritance. To keep things safe, it is forbidden for a non-privileged programmer to introduce a new metatype: that operation has to be proved safe and consistent.

We introduced a notion of extensibility at the model level by deporting notions regarding the type system into the metatypes. Thus, adding a metatype is both powerful and dangerous, and the programmer can't be allowed to add a metatype the same way she adds a utility class. But there is no exception to the extension rule for the metatypes: we install that security policy simply by writing the class `MetaType` (which is the class of every metatype) in such a way that it can't be instantiated once the system is running.

3 Embedded compilation process

We have defined a powerful extensible type system for our intermediate language. Since the type system is dedicated to object programming styles, it is rather natural to make the intermediate language itself be object-oriented. But we don't want to deal with the drawbacks of an object-only system, so we desire to be able to install things in such a way that the performances at runtime will be nearly optimal. We want to be able to resolve at load-time some of the calls that are so heavy when done completely dynamically.

To fulfill our need for speed, we can't rely on a virtual machine to interpret bytecodes. Instead, we want to introduce an embedded compilation process that transforms the high-level intermediate language into its executable form. It is vital to fill the gap between the intermediate form and the machine code, at least for critical operations (such as arithmetic ones), that are used permanently during the execution of any program. Though, we don't want to restrain ourselves to native code generation: depending on the case, some policy may be more well-suited.

3.1 *Mechanism flexibility*

We use a load-time binding mechanism of the code that allows the system conceptor to customize to her taste the way things are going to be concretized. The different possibilities are:

- direct compilation to native code by a call to the system component "code generator",

- call to one or several other methods,
- virtual code.

This binding is realized by a dedicated method per class: the `bind()` method is called by the loading process to realize bindings. This method is just a normal method, that is written in intermediate language, like the other ones. It receives under a symbolic form the method of the class that is to be bound, so that only one `bind()` per class is needed.

The `bind()` method is not intended to be invoked directly, since the normal type system would not be able to verify the symbolic data it manipulates. For the programmer, there is a `callBind()` available that wraps around `bind()` and performs needed checks (for example, it checks that the method to be bound is valid for the object `this`, or even that the class performing the `callBind` is allowed to request a bind for the target method). An important point is that a `bind()` of a class `A` is called each time some other class tries to install some code that manipulates `A` (through an instance for example). The bind does not specify an absolute way of compiling a method, but a compilation that depends on the calling context.

In addition, it is possible to let a `bind()` implementation not do anything about a method, in which case, the implementation of the superclass is used instead.

Thus, the compilation of high-level code to intermediate code allows us to unify the type systems of the sources languages whereas the loading mechanism allows us to specialize the bindings as illustrated by Figure 2.

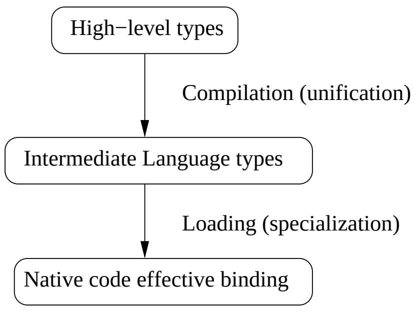
High−level types
Compilation (unification)
Intermediate Language types
Loading (specialization)
Native code effective binding

Figure 2. Type systems transformation.

This binding mechanism is interesting from an optimization point of view, since it empowers the program to compile itself into the most efficient binary form (with a fine control over what “efficiency” means, depending on the context). In opposition with traditional languages like `Java` that make use of many `native` methods, this binding method acts as a late compiler that can be aware of the call context of a given method. Furthermore, such a mechanism makes it possible to fully exploit the capabilities of a given platform, by generating specific machine instructions for driving a particular device for example.

It is worth noting that, if we have access to really low level compilation processes, we are not restricted to it: the binding method can also be used to combine existing bindings, and from there to generate optimized composite operations without rewriting anything but the glue between the components.

Another possibility brought by the binding mechanism is to apply aspect-oriented programming techniques to the program one develops. By writing an appropriate `bind()` method, one can specialize the meaning of a given method. For example, it is possible to wrap around memory accesses to introduce new checks or events. In fact, the binding provides some kind of basic aspect weaver that can be used in non-trivial cases.

It is rather simple to imagine how we could implement a class invariant using `bind()` as shown in Figure 3 (note that we use a pseudo C syntax for the examples). First, the programmer would have to code a method to verify the invariant, and then the `bind()` method of the class would do the following: for every call to a method of this class (that is, for every compilation of that method, or better, for every invocation of the `bind()`), the real call to the method would be followed by the verification of the class invariant.

```
void check_invariant() { ... }
void bind() {
    switch(METHOD) {
        ...
    }
    invoke(CHECK_INVARIANT);
}
```

Figure 3. Class invariant.

The point here is to note that the assertion will be checked *after* each method invocation, and not *at the end* of each invocation, as it would have been easy to do (and less expressive) with a classical `assert()`. One could also build a full pre/post conditions environment, though it would require more work on interfaces to prevent the programmer from coding every assertion as a separate function.

Thanks to this powerful mechanism, the programmer has access to the definition of a binding, and is allowed to customize it. Most of the possibilities are already investigated in other platforms, for example in Java a reflexive JIT (Just In Time compilier) in [9], method interception in AspectJ in [6], stack introspection for callee monitoring in [13]. All these mechanims are here realized by one mechanism and are not dependent of any source language.

3.2 *Security concerns*

Naturally, the binding methods are fully responsible for the consistency of the generated code, and a lot of checks have to be performed to ensure for example the soundness of the type system after the code has been turned into native. As a consequence, at the end of every compilation there should be a whole bunch of tests to make sure everything went correct, which is impossible without imposing constraints over the programmer.

In such a case, it would be trivial to bypass the type system and manipulate pointers. For example the programmer could write some methods `int cast(Object)` and `bind()` as in Figure 4, that would just compile the first one into a no-op to transform the first argument into an `int` by simply copying it into a variable of type `int`.

```
int cast(Object) {
    return 0;
}
void bind() {
...
    case CAST:
        copy(ARG1, ReturnVariable);
...
}
```

Figure 4. Simple attack.

To control the access to the code generator, we need an intermediate layer that will be in charge of checking the adequation between the signature of the method (its type) and the generated code. For each type we will use its *metatype* to make sure its behaviour is legal regarding this.

3.3 *Binding scheme*

As we said, metatypes are responsible for ensuring the type correctness regarding the target of a method call. For example, a malicious programmer could try to call a *virtual method* on a object whose class is an instance of `Value`, which is obviously meaningless and could lead to a security hole (if the integer has the same binary form as a pointer, pointer arithmetic is then allowed). In such a case, the metatype has to reject the compilation: more precisely it has to control that the origin of the call is a valid type for the operation. For example, in Figure 5, one tries to compile a virtual method `A_METHOD` from class `A_CLASS`. This code has to be rejected because the current instance (`this`, contained in `ctxt`) is an `int` and not an instance of `A_CLASS`.

Let us now expose in details the binding scheme via an example (Figure 6).

```
void cast(int i) {
    return;
}

void bind() {
...
    case CAST:
        callBind(A_CLASS, A_METHOD, ctxt);
...
}
```

Figure 5. Indirect attack.

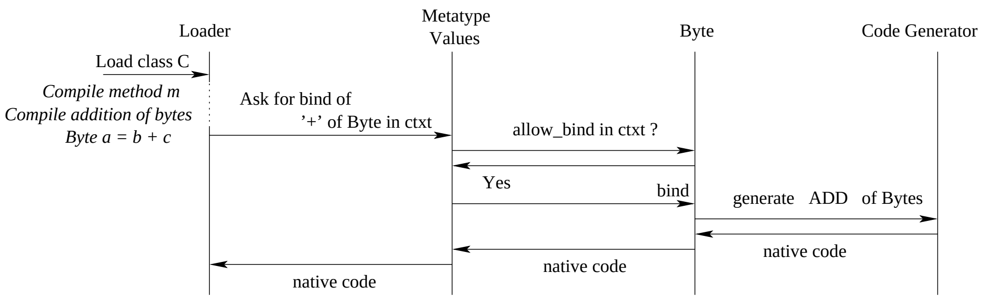

Figure 6. Binding scheme.

We consider a method `m()` of a class `C` that wishes to perform an addition between two bytes (an integral type, represented by the class `Byte`). First, the `bind()` method that is in charge of `m()` builds a container `ctxt` and fills it with the informations related to the call context of the method, then checks that the arguments are compatible with the signature of the method.

Since we perform an addition between two bytes, we are calling the method `Add()` of the class `Byte`.

The second check to perform is to make sure that the class that will realize the binding (`Byte` here) allows it in `ctxt`. If this is the case, then it is called. To avoid the need of reimplementing `bind()` at every level in our type hierarchy, it is legal to allow a bind, and not do anything related to it in the `bind()` method. This means that the superclass has to be considered in place of the class itself for performing the bind: it is the compilation of a purely inherited method.

The addition has no reason to be forbidden somewhere, so we assume this check is completed. Moreover, it is highly likely that the compilation of such an addition will be realized by a single machine operation, so that the `bind()` method of `Byte` will define a behavior for `Add()`: this is the one that is called.

If we look again at our example at Figure 5, the first set of tests (before beginning to bind) are trivially filled, since `cast()` doesn't do anything. The problems arise in the `callBind()` where the attacker tries to call a virtual method on an `int`. The context (`ctxt`) will be tested against the signature of the method, and it will appear that the object `this` is not of the correct type. Thus, the binding will be rejected.

3.4 *Discussion*

Another motivation for using metatypes is to bring together at a single place the code used to check that a compilation is applied in well controlled conditions and provides conform code. Doing this, we both reduce the size of the code and make it safer since we avoid duplication whenever it is possible, and therefore reduce the potential number of bugs and neglects.

Any type checking operation or cast has to be performed in the body of a method that belongs to a metatype. Our security policy ensures that it is illegal for a non-privileged programmer to call a cast (which exists at a really low level in our architecture). To make things safe we need to make sure that any call to this cast is encapsulated in a system call, where we know exactly what operation is being performed: we can ensure that the operation in a whole is well-typed even if its parts involve type-breaking operations. So, instead of performing directly a cast, one has to call the `bind()` of a well-known action that can in turn perform the cast. Those system calls have to be the last step before the code is converted to machine code, and therefore loses any notion of type. As for the explicit type checks, they can be allowed, but will be either false or redundant (so they could as well be forbidden). That policy can be easily implemented using the binding system itself, since it is just a type verification on the caller, which we know during the execution of `bind()`.

Since the last checking of type consistency occurs right before the generation of machine code, we have some kind of portable type verification : provided that the generator (which is usually a not too complicated unit) is correct, we don't need to prove anything else for proving the consistency of the type system on a new platform.

Though it was not a major point at the very beginning, the optimization possibilities that arise when using metatypes are really interesting. First, we came to factorize a lot of code, which reduced the size of the code. And more interesting is the fact that by introducing an embedded compilation process, we are able to perform at load-time checks that cannot be static. For example, if one wants to ensure that a given method is never called from a certain class, this check cannot be performed statically since we are in an open world, and checking it at every invocation is an overkill. At the load-time on the other hand, it is trivial to inject that check in the `bind()` method, so that every subsequent invocation respects this invariant by construction. One can view the load-time as a period where the world is locally closed: there one can perform some kind of semi-static check because the context of the call is available. This technique has two major advantages: first, we avoid the cost of a check at runtime for every invocation of the access-restricted method, and secondly there is no absolute need for complex error-handling since the code that performs illegal calls can be as well rejected.

4 Experimental results and possibilities

4.1 *Current implementation*

We have implemented the previously described mechanisms in the Camille operating system, see [1]. Camille is an operating system dedicated to small objects like smart cards. Camille is an exo-kernel providing a minimal number of services that are supposed to be reliable, and an extension mechanism that allows one to:

- load new applications (that is not the historical purpose of smart card operating systems),
- load new system components in order to extend the system itself.

Camille can load bytecode written in an intermediate language called Façade which is a fully object oriented bytecode. We have modified the implementation to introduce our type system based on metatypes. The advantage of the use of Façade has been shown in [10]: the Camille system is able to verify that the loaded code is type correct in linear time using a PCC [8] like mechanism with a lightweight bytecode verification algorithm [12]. So, the embedded system is able to verify that the loaded code is well-typed without any assistance and, using the dynamic binding mechanism describe in the previous section, we can offer specialization facilities without breaking this benefice.

Before the start of this work, there already existed a flexible binding mechanism, but it could not be available to the programmer without potentially breaking the security of the system. One benefit of our work is to make possible for anybody to write powerful (yet controlled) binding policies to suit her needs. Additionally, by centralizing the security-related stuff in a well-controlled place (the metatypes), we reduce the number of paths to the core system, and thus reduce the risk of observing unexpected behaviours.

Our implementation of the metatypes layer on Camille required the introduction of 6 additional types (the metatypes presented in Figure 1). Those classes contain all the checks that are performed to ensure the security of applications when a new class is loaded.

The code of the metatypes represents around 10% of the total code of the Camille platform at the moment, which includes the type system basis, and also all checks and casts to ensure its security. The amount of code dedicated to the metatype level is not too high, provided that the code does not contain any applicative code, but only the exo-kernel part.

4.2 *Multiple inheritance proposal*

What has not yet been exploited is the possibility to extend the type system in good conditions (that is, without changing the existing components). In this section we will have a look at what would be needed to add support for multiple inheritance in our system.

Please keep in mind that this addition is not neutral regarding the security of the whole system, since we access resources where a single mistake can mess the whole process or lead to the violation of a security rule.

The implementation of a sub-hierarchy of multiple inheritance classes would go through several steps. The first thing that is needed is a metatype for all the future classes that will support multiple inheritance. That metatype would probably be a sibling of `Class`, that is a subclass of `Boxed` in our current type system (see Figure 1). The modification is presented at Figure 7, only showing the concerned part.

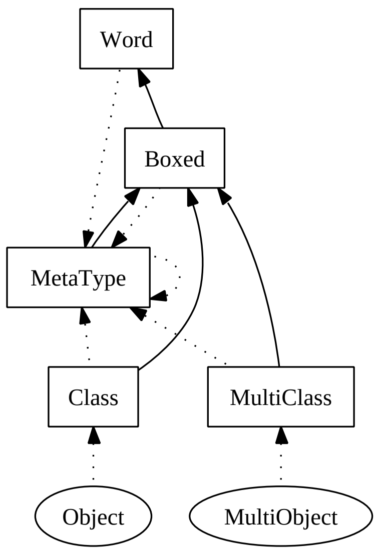

Figure 7. Metatypes hierarchy

A key point in installing a new metatype is to override the methods that define the behaviour of its instances (the classes). For example, the method isAKindOf will obviously not have the same implementation for a single inheritance class than for our new multiple inheritance classes.

One would have to define the semantics of the setParent method, that is charge of binding the class that is being constructed to the type system. In the case of usual single inheritance, that method defines the class ancestor and must be called exactly once. In our case of multiple inheritance, we will allow several invocations, provided that all given ancestors support multiple inheritance. It is also worth reminding that the whole binding mechanism is designed so as to strictly forbid a class to inherit from a class with a different metatype.

One effect of these repeated setParent is to merge the methods and fields tables. One possibility is to provide an indirection table for each parent, so that one can associate methods or fields from the static type of an object to their implementation.

Those tables define the lookup policy to be applied in case of conflicts between multiple candidate methods from an overloading point of view.

Additional problems arise in less obvious parts of the loading mechanism. A particularity of FAÇADE is that it makes use of *temporary* variables, whose type is not provided statically, though it is needed to ensure the type-correctness of the program. In a case like figure 8, where tmp is a temporary variable, for single inheritance objects the type of tmp after the block would simply be the smallest common parent type of A and B. If we introduce multiple inheritance, the case is a bit more complicated since the smallest existing common parent is not unique and therefore would not necessarily export the entire interface that sould be legal for tmp.

For example, if A inherits from I1, I2 and I3, whereas B inherits from I1, I2 and I4, then the type for tmp should be I1 **and** I2.

Once again, we solve this issue by a call to a metatype. When trying to unify two types, we call a method of their metatype (it has to be the same, or else the code is broken) that returns an instance. In the case of single inheritance, this instance is an existing type, whereas for multiple inheritance, it can be a virtual type that exports exactly the needed interface, and that can be destroyed at the end of the loading process. Thus, to support this aspect of multiple inheritance, all we need to do is to overload the unify() method of the metatype.

So the addition of new kind of objects is rather simple, though it is a highly risky task because poorly written metatypes can lead to almost any failure of the system. The benefit of the approach is to provide a valid system from a valid one when adding a correct metatype.

```
if(condition) {
  tmp = f(); // returns a A
} else {
  tmp = g(); // returns a B
}
```

Figure 8. Type unification

5 Conclusion

This paper presented an architecture for an open and flexible operating system. We showed one could take advantage of a dynamic binding mechanism at loading time both for efficiency (by allowing native code generation) and for flexibility (since it allows on-the-fly weaving of the code). We also introduced the use of metatypes as a convenient way of making the binding effective and safe, in the sense that they provide a fine access control over the points in the code where potentially dangerous actions have to be performed Finally, we provided our system a notion of extensibility, not only at the application level (which was one of the primary goals) but also at the type system level, since we designed the loading process as generic as possible, so as to be able to add new "object" notions, without rewriting the whole engine. This secondary extensibility is not dynamic, though. And thus not available to the users. It would be interesting to investigate under which constraints a part of this mechanism could be extracted from the core of the system made available.

References

[1] Camille. http://www.lifl.fr/RD2P/CAMILLE.

[2] Cli. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[3] Bobrow, D. G., and Kiczales, G. The common lisp object system metaobject kernel: a status report. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming* (1988), ACM Press, pp. 309–315.

[4] Chen, Z. *Java Card^{TM} Technology for Smart Cards : Architecture and Programmer's Guide.* The Java^{TM} Series. Addison Wesley, 2000.

[5] Guy L. Steele, J. *Common LISP: the language.* Digital Press, 1984.

[6] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of aspectj. In *ECOOP* (2001), vol. 2072 of *LNCS*, Springer, pp. 327–353.

[7] Maosco. Multos web site. http://www.multos.com.

[8] Necula, G. C. Proof-carrying code. In *Proc. of 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)* (Paris, France, 1997), pp. 106–119.

[9] Ogawa, H., Shimura, K., Matsuoka, S., Maruyama, F., Sohda, Y., and Kimura, Y. Openjit: An open-ended, reflective jit compiler framework for java. In *ECOOP 2000 - Object-Oriented Programming* (2000), LNCS 1850, pp. 362–387.

[10] Requet, A., Casset, L., and Grimaud, G. Application of the B formal method to the proof of a type verification algorithm. In *Proc. of High Assurance Systems Engineering – HASE* (2000), IEEE Press.

[11] Rippert, C., and Stefani, J.-B. Building secure embedded kernels with the think architecture. In *Proc. of the workshop on Engineering Context-aware Object-Oriented Systems and Environments – OOPSLA* (Seattle, USA, 2002), IEEE Press.

[12] Rose, E., and Rose, K. H. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98* (1998).
[13] Wallach, D. S., Balfanz, D., Dean, D., and Felten, E. W. Extensible security architectures for java. In *Proc. of the 16th symposium on Operating Systems Principles* (1997), ACM Press.