Evolving Software with Extensible Modules

Matthias Zenger
Programming Methods Laboratory
Swiss Federal Institute of Technology Lausanne
INR Ecublens, 1015 Lausanne, Switzerland
matthias.zenger@epfl.ch

November 6, 2002

Abstract

We present the programming language KERIS, an extension of JAVA with explicit support for software evolution. KERIS introduces extensible modules as the basic building blocks for software. Modules are composed hierarchically revealing explicitly the architecture of systems. A distinct feature of the module design is that modules do not get linked manually. Instead, the wiring of modules gets inferred. The module assembly and refinement mechanism of KERIS is not restricted to the unanticipated extensibility of atomic modules. It also allows to extend fully linked systems by replacing selected submodules with compatible versions without needing to re-link the full system. Extensibility is type-safe and non-invasive; i.e. the extension of a module preserves the original version and does not require access to source code.

1 Introduction

This paper presents KERIS, a pragmatic, backward-compatible extension of the programming language JAVA [27] with explicit support for modular, component-oriented programming [59, 60]. Many modern programming languages provide mechanisms for modular program development. They allow to define modules that depend on functionality imported from other modules. Furthermore there is often support for separate compilation, allowing modules to be compiled in isolation. Separate compilation and the ability to abstract over external functionality make it possible to flexibly deploy modules in different contexts with different cooperating modules.

Opposed to this typically well supported form of reuse, most mainstream programming languages do not address the ability to extend modules without planning extensibility ahead. Since modules, as architectural building blocks, are subject to continuous change, we consider this lacking support for unanticipated extensibility to be a serious shortcoming. In practice one is required to use ad-hoc techniques to introduce changes in modules. In most cases this comes down to hack the changes into the source code of the corresponding modules. This obviously contradicts the idea of deploying compiled module binaries — a process which does not require to publish source code. But even for cases where the source code is available, invasive changes like source code modifications are considered to be error-prone. With modifications on the source code level one risks to invalidate the use of modules in contexts they get already successfully deployed.
The design of the programming language Keris includes primitives for creating and linking modules as well as mechanisms for extending modules or even fully linked programs statically. Keris’ type system ensures that the definition, assembly, and evolution of modules is safe. Programs written in Keris are closed in the sense that they can be executed, but they are open for extensions that statically add, refine or replace modules or whole subsystems of interconnected modules. Extensibility does not have to be planned ahead and does not require modifications of existing source code, promoting a smooth software evolution process.

In this paper we introduce Keris as an extension of the programming language Java. In Section 2 we substantiate the need for linguistic abstractions in object-oriented programming languages for programming in the large and for evolving systems in the large safely. In Section 3 we present the design of the programming language Keris by a step-wise introduction of Keris’ extensible module abstractions. The examples presented in Section 4 explain how extensible modules support family polymorphism. Section 5 gives an informal overview over the type-system. Our prototypical implementation of the Keris compiler gets briefly reviewed in Section 6, followed by a discussion of related work in Section 7. Section 8 concludes.

2 Motivation

Like many popular object-oriented languages, Java does not provide suitable abstractions for programming in the large [17]. In this section we argue that Java’s package system is generally too weak to be useful as an abstraction for reusable software components. We do this by looking at three important properties: modularity, genericity, and extensibility.

2.1 Modularity

Modularity is about the separation of components from other components both logically and physically. Therefore, modularity is essential to allow software components to be developed and compiled independently. This is typically achieved by means of encapsulation and by the explicit specification of contracts between components. These contracts define explicitly what services a component provides and what other components are needed to render the services.

Java’s package system offers relatively good support for modular program development. It requires that context dependencies are specified explicitly and it has support for separate compilation. On the other hand, Java’s package abstraction is often too coarse-grained, so that structuring software systems consisting of many smaller subsystems can become very difficult on the package level. For instance, large libraries often require means for internal structuring. It is possible to nest packages, but this also limits access to non-public members. Therefore all classes that need to access library internal data, which does not get exposed to the outside world, have to reside in the same package.

Java’s package mechanism was designed mainly for structuring the name space and for grouping classes. A package does not even allow to fully encapsulate a set of classes since the Java programming language does not offer a way to close packages.1 Thus, like

1In Java class loaders can be used at runtime to ensure that only a fixed set of classes is loaded from a package. The concept of sealed packages exploits this mechanism to restrict class loading for classes of such a
in most popular object-oriented languages, classes are predominantly used to structure software systems.

Classes on the other hand do not fully support modular programming [58, 14]. In general, classes cannot be compiled separately; mutually dependent classes have to be compiled simultaneously. Since classes do not define context dependencies explicitly, it is difficult to find out on what other classes a class depends. This can only be found out by inspecting code.

Even though classes are the basic building blocks for object-oriented programming, most classes do not mean anything in isolation. They have a role in a specific program structure, but there is only limited support to formulate this role or to make this role explicit. A priori, class interactions are implicit, if not using a special design pattern that emphasizes cooperating classes. Due to the lack of expressing dependencies between classes explicitly, formulating design patterns, software components, the architecture of a system, and even expressing the notion of a library on the level of the programming language turns out to be extremely difficult in general.

A good example for this problem is the way how industrial component models represent software components in class-based object-oriented languages. In these models, the implementation of a software component is typically guided by a relatively weekly specified programming protocol (e.g. JAVABEANS [33]). The composition of software components is mostly even performed outside of the programming language, using metaprogramming technologies. Thus, neither the process of manufacturing a component nor the component composition mechanism are type-safe.

2.2 Genericity

Modularity is essential for the independent development of software components. But modularity alone does not allow to deploy components independently of each other. Support for independent deployment requires that modules are generic with respect to their context dependencies; i.e. they have to abstract over depending modules. Furthermore, a mechanism is needed to instantiate a component and resolve its context dependencies by linking it with concrete instances of depending components. Thus, genericity is required whenever one wants to reuse a single component in different contexts with different cooperating components.

JAVA packages are not generic. Packages hardwire their context dependencies (imports) by referring to other concrete packages. Thus, there is no explicit support for the reuse of packages in different contexts or with different compatible dependent packages. Even though references to other packages are specific, the JAVA runtime environment offers possibilities to adjust the “linking context” so that a different implementation of a cooperating package is chosen at load time; for instance by modifying the class path or by using special class loaders [50]. Such hacks are statically unsafe and therefore do not provide acceptable alternatives for genericity.

package only to a particular Jar file. Regarding the open nature of packages, it is surprising to see that adding classes to a JAVA package is not type-safe. This can break programs that import all classes of a package via the star-import command.
2.3 Extensibility

Besides modularity and genericity, extensibility is another important property. It is important because in general, independently developed components do not fit off-the-shelf into arbitrary deployment contexts. They first have to be adapted to make them compliant with a particular deployment scenario. Apart from this, extensibility is also an essential requirement for enabling software evolution. Software evolution includes the maintenance and extension of component features and interfaces. Support for software evolution is relevant, because components are the basic architectural building blocks of software, and as such, subject to continuous change. A typical software evolution process yields different versions of a single component being deployed in different contexts [44]. Extensibility is also required when developing families of software applications [48, 6]. For instance, software product-lines [34, 65] rely heavily on a mechanism for creating variants of a system which share a common structure but which are configured with possibly different components.

JAVA supports the development of extensible software only on a very low level by means of class inheritance and subtype polymorphism. Extensibility has to be planned ahead through the use of design patterns typically derived from the AbstractFactory pattern [24]. Furthermore, extensibility can often only be achieved by using type tests and type casts due to the lack of appropriate means to refine abstractions covariantly. In general, such techniques circumvent the static type system and are therefore dangerous to apply in practice.

With JAVA’s late binding mechanism developing open software that can be extended with plug-ins is relatively easy. Again, this has to be planned ahead and allows only to extend an application in a restricted framework [44]. For writing applications that are open for unanticipated extensions, often complicated programming protocols have to be strictly observed. An example for such a protocol is the Context/Component design pattern described in [69].

Another pragmatic approach to extensible software, is to use static meta-programming or at runtime, reflective capabilities. Again, such approaches are typically unsafe, since they sidestep the type system.

The main contribution of this paper is a JAVA-based mechanism that allows type-safe unanticipated extensions of systems. This approach is based on a system of extensible modules, which will get introduced in the next section.

3 Extensible Modules for Java

The design of the programming language KERIS was driven by the observation that extensibility on the module level can help to develop highly extensible applications [30]. KERIS tries to facilitate the development of extensible software by providing an additional layer for structuring software components. This layer introduces modules as the basic building blocks for software. With KERIS’ modules, it is possible to give concrete implementations for concepts like design patterns, libraries, applications or subsystems. All this is done in a completely extensible fashion, allowing to refine existing software or to derive new extended software from existing pieces. To keep software extensible, KERIS promotes programming without hard links which are frequently found in JAVA programs in form of
class instantiations or accesses to static methods or fields. Of course, being a conservative extension of JAVA, it is possible to introduce hard links in KERIS whenever desired. The module system of KERIS was designed to fit smoothly between JAVA’s class and package level. With support for true modules, the package system is now mainly used to structure the module name space. Of course, it would be easily possible to add name space management facilities to our module abstractions if backward compatibility to JAVA would be irrelevant.

3.1 Defining Modules

In KERIS, modules are the basic top-level building blocks for software supporting separate compilation as well as function and type abstraction in an extensible fashion. KERIS’ modules specify context dependencies explicitly. They can only be deployed in contexts that meet these requirements. We now present a small example that defines a module SORTER which provides functions for reading a list of words, for sorting, and for printing out lists.

```java
module SORTER requires INOUT {
    String[] read() { ... INOUT.read() ... }
    void write(String[] list) { ... INOUT.write(list[i]) ... }
    String[] sort(String[] list) { ... }
}
```

The header of the module declaration states that module SORTER depends on functionality provided by another module INOUT. Within the body of a module it is possible to access the members of the own module as well as all the members of modules that are declared to be required. Members of modules are generally accessed by qualifying their names with the corresponding module. This distinguishes requirements from imports of traditional module systems which typically make members of other modules accessible so that they can be used in unqualified form.

It remains to show a specification of module INOUT. We do this by defining a module interface that specifies the signature of this module. Such a module interface does not contain any code, it only specifies the types of members provided by a concrete implementation of this module.

```java
module interface INOUT {
    String read();
    void write(String str);
}
```
We will now define a module `CONSOLE` that implements this interface and thus is a possible candidate for being used together with module `SORTER`.

```java
module CONSOLE implements INOUT {
    String read() { ... System.in.read() ... }
    void write(String str) { System.out.println(str); }
}
```

This module implements the functions `read` and `write` by forwarding the calls to appropriate methods of the standard JAVA API for text in- and output on a terminal. Here is an alternative implementation for `INOUT` based on functionality provided by a third module `LOG`.

```java
module LOGIO implements INOUT requires LOG {
    String read() { ... System.in.read() ... }
    void write(String str) { LOG.log("log: " + str); }
}
```

Note that module `SORTER` does not explicitly implement a module interface. This is not strictly necessary since every module declaration implicitly defines a module interface of the same name. Nevertheless, the separation of module implementations from interfaces is an important mechanism that is essential to enable separate compilation of recursively dependent modules. Some module systems, e.g. Oberon's module system, provide means to support separate compilation without separating module interface definitions from module implementations, but only for modules without recursive dependencies. Besides separate compilation, explicit module interfaces are also important as a facility for hiding concrete representations of module members. Furthermore, they can be used as a vehicle for presenting different views of a single module implementation.

Figure 1 illustrates some of the modules we defined so far. Module illustrations consist of two parts: one part shows both the module to define (the topmost box) and the required modules (the boxes on the left side). The other part lists all the module members. We use the convention that boxes refer to modules and rounded boxes refer to module interfaces.

### 3.2 Linking Modules

Before discussing the module composition mechanism, we have to stress the distinction between modules and module instances. A module can be seen as a “template” for multiple module instances of the same structure and type. We have to differentiate between the two, since we want to be able to deploy a module more than once within a software system. For instance, we could have two different instances of the `SORTER` module that are linked together with different `INOUT` module instances.

In Keris, modules are composed by aggregation. More concretely, a module does not only define functions and variables. It may also define module instances as its members. These nested module instances, we also call them *submodules*, can depend on other modules visible in the same context. The following definition for module `APP` links module

---

2We use a terminology here which is not fully consistent with the one on the class level. *Submodules* denote *nested modules* and have nothing to do with *subclassing*. The motivation for naming nested modules *submodules* comes from nested modules modeling *subsystems*.
SORTER with module CONSOLE by declaring both to be submodules of the enclosing module APP.

```java
module APP {
    module SORTER;
    module CONSOLE;
    void main(String[] args) {
        String[] list = SORTER.read();
        list = SORTER.sort(list);
        SORTER.write(list);
    }
}
```

Submodule definitions start with the keyword `module` followed by the name of the module implementation. The enclosing module aggregates for every submodule definition an instance of the specified module. Thus, module `APP` aggregates two module instances `SORTER` and `CONSOLE`. A submodule can only be defined if its deployment context, given by the enclosing module, satisfies all the requirements of the submodule. The requirements of a submodule are satisfied only if all modules required from the submodule are either provided as other submodules, or they are explicitly required from the enclosing module.

The program above defines two submodules `SORTER` and `CONSOLE`. Module `SORTER` requires a module instance `INOUT` from the deployment context, `CONSOLE` does not have any context dependencies. The module definition of `APP` is well-formed since it defines a `CONSOLE` submodule that implements `INOUT`, and therefore provides the module that is required by the `SORTER` submodule. Note that module `CONSOLE` was only introduced in module `APP` for that reason. Module `APP` does not refer to members of `CONSOLE` directly.

Figure 2(a) illustrates module `APP`. The submodules of `APP` are displayed as nested modules. The wiring of the submodules, which is implicit in KERIS programs, is made explicit with an arrow from the implementing module to the requirement.

Modules without any context dependencies like `APP` can be executed if they define a
main method. For executing a module, an instance gets created and the main method is called. The main method of the previous code shows that submodules get accessed simply via the module name.

Similarly to the previous code, we could try to link module SORTER with module LOGIO.

```plaintext
module BuggyApp {
  module SORTER;
  module LOGIO;  //
}
```

A verification of the context dependencies reveals that this module declaration is not well-formed. LOGIO requires a module instance LOG which does not get declared within BuggyApp. Since we want BuggyApp to be parametric in the cooperating module LOG, we have to abstract over the LOG instance by requiring it from the context. This has the effect that inside of the module body we are now able to refer to a module instance LOG without actually giving a concrete definition. Therefore the following definition of module LOGSORTER is well-formed. Figure 2(b) gives a schematic illustration which shows that for LOGSORTER, all requirements of submodules are resolved.

```plaintext
module LOGSORTER requires LOG {
  module SORTER;
  module LOGIO;
}
```

The previous examples show that modules get composed by hierarchically aggregating submodules. A module that hosts a set of submodules is only well-formed if it satisfies the context requirements of all of its submodules. A module satisfies the requirements of a submodule if modules required from that submodule are either present in form of other submodules, are explicitly required by the host module, or are subsumed by the host module itself.

This hierarchical composition mechanism has the advantage that the static architecture of a system gets explicit. Furthermore, module composition does not require to link modules explicitly by specifying how context dependencies are satisfied at deployment time. Instead, the module interconnection gets inferred. With this approach we avoid linking modules by hand which can be a tedious task that raises scalability issues [67]. On the other hand, our inference technique only succeeds if we avoid ambiguities; i.e. our type system has to ensure that references to module instances identify modules uniquely in every context. One implication of this is that a module can never define or require two nested module instances (submodules) that implement the same module. If this would be the case, a simple module name could not identify a module implementation unambiguously anymore.

A system like this is reminiscent of classical module systems for imperative programming languages like Modula-2 or Ada. Such module systems allow only one implementation for each module globally, whereas Keris has this restriction only locally for every module context. Globally, there are no restrictions, allowing systems to include as many instances of a single module as required.

Nevertheless, some of the limitations may still appear as rather big restrictions. As we will see later, many of these limitations can be overcome with module specializations.
introduced in Section 3.4. Furthermore, it is always possible to introduce nested modules for instantiating multiple instances of a single module. The type system, briefly explained in Section 5, ensures that such multiple instances are used in a consistent, non-conflicting manner.

3.3 Refining Modules

We now come to the problem of extending modules. Since we do not want to break code that makes use of existing modules, we are not allowed to touch existing modules. In short, extensibility has to be additive instead of being invasive.

KERIS has support for non-invasive extensions through a module refinement mechanism. This mechanism allows to refine an existing module by providing new functionality or by overriding existing functionality. The refined version of a module is backward compatible to the original module in the sense that it can be substituted for it. Thus, KERIS lifts the notion of compatibility between classes expressed by a subtyping relation to the more coarse-grained level of modules.

We now present a refinement of module SORTER that provides a more efficient implementation for the sort function. In the example below, we use a merge-sort technique for sorting. Apart from a new implementation of sort which overrides the existing implementation, we also define various other helper functions. One of them is declared to be private, which hides it from clients of the module.\textsuperscript{3} In other words, such functions do not get exported and can only be used internally. Function merge on the other hand can also be accessed from outside of module QUICKSORTER.

```plaintext
module QUICKSORTER refines SORTER {
    private String[] sub(String[] list, int start, int end) { ... }
    String[] merge(String[] first, String[] second) { ... }
    String[] sort(String[] list) {
        return (list.length < 2) ? list :
               merge(sort(sub(list, 0, list.length/2)),
                      sort(sub(list.length/2, list.length)));
    }
}
```

Module QUICKSORTER is a refinement of module SORTER. It inherits the interface and the member implementations from SORTER and therefore implements the module interface of SORTER as well. Note that it also inherits the context dependencies; i.e. QUICKSORTER actually requires a INOUT module.

Similar to this refinement of module implementations and their implicit interfaces, it is also possible to refine plain module interfaces like INOUT.

So far, we only saw how to refine the functionality of atomic modules. As motivated in in Section 1, these refinements do not affect existing code. So how do we integrate our more efficient sorting module into a system that makes already use of the old SORTER module? Since systems are represented by modules, it is probably not surprising to do this again with a refinement. We explained before that KERIS promotes programming without

\textsuperscript{3}Module refinements and specializations see the modules they refine or specialize as white-boxes; i.e. they can freely access and override private members of the original module.
hard links. Following this idea, we allow to override submodule declarations in module refinements. The following code refines our module APP representing an executable application by covariantly overriding the SORTER submodule.

```plaintext
module XAPP refines APP {
module QUICKSORTER;
}
```

The refined module XAPP replaces the nested module implementation SORTER with one for module QUICKSORTER. Consequently, the inherited main method now refers to the QUICKSORTER submodule. In fact, we can now access the QUICKSORTER submodule via both module names, SORTER and QUICKSORTER. The only difference is that when accessed via QUICKSORTER, we can refer to the new functions. The ability to refine a module interface stepwise to allow different access levels is called incremental revelation [16].

This small example demonstrates that our module assembly and refinement mechanism not only supports the extension of atomic modules. It also allows us to extend fully linked programs (represented by modules with aggregated submodules) by simply replacing selected submodules with compatible version. There is no need to establish module interconnections again; we reuse the fully linked program structure and only specify the submodules and functions to replace or add.

This extensibility mechanism features plug-and-play programming. It does not touch existing code. After having refined our application with module XAPP we can still run the old application APP. We could even assemble a system that makes use of both modules without having to fear unpredictable interferences.

### 3.4 Specializing Modules

Refining a module is the process of extending a module by adding new functionality or by modifying existing functionality through overriding. A module refinement yields a new version of an existing module. This new version subsumes the old one; i.e. it is backward compatible to the old version. As a consequence, it is always possible to replace a module with one of its refinements.

In the following code, module BUGGYMOD aggregates a submodule that subsumes another submodule. In other words, BUGGYMOD defines a context in which two different versions of one module are present. Since references to submodule SORTER are ambiguous within module BUGGYMOD, we declare this case to be illegal.

```plaintext
module BUGGYMOD requires INOUT {
module SORTER;
module QUICKSORTER;
}
```

But as Section 4.2 will motivate, we would sometimes like to be able to evolve a module and use the evolved version side by side with the original one. For this case, we need a different form of reuse: We would like to define a new module on top of an existing one, but we do not want the new module to subsume the old one. We call this process of creating a new distinct module which reuses the definition of an existing module specialization. As an example, we define a specialization of the SORTER module in the following
code. Module SETSORTER implements a set semantics for sorting and is due to this change in semantics not implemented as a refinement of SORTER.

```java
module SETSORTER specializes SORTER {
    String[] filterDuplicates(String[] list) { ... }
    String[] sort(String[] set) {
        return super.sort(filterDuplicates(set));
    }
}
```

Module SETSORTER inherits all members and requirements from SORTER and defines two new functions. Function filterDuplicates can be used to filter out duplicate entries in lists. Function sort overrides the corresponding function in SORTER, but its implementation is still able to refer to the former implementation via the keyword super.

As a specialization of SORTER, module SETSORTER is not required to be backward compatible to module SORTER. In particular, it neither subsumes module SORTER nor implements any of the module interfaces that are implemented by module SORTER. This restriction turns SORTER and SETSORTER into completely different modules. Consequently, it is perfectly legal to define a module with both a SETSORTER and a SORTER submodule.

```java
module SORTING requires INOUT {
    module SORTER;
    module SETSORTER;
}
```

Often, mutual referential modules have to be specialized at the same time consistently. The ability to refer to a specialized version of a module requires that we are able to specialize context dependencies as well. This “rewiring” is expressed in the following code using the as operator. The MYSORTER module specializes module SORTER and instead of requiring the original INOUT module, it now refers to a specialized MYINOUT module.

```java
module MYSORTER specializes SORTER requires MYINOUT as INOUT { ... }
```

A more complete example for module refinements and rewiring of context dependencies can be found in Section 4.2.

While module refinements promote the substitutability of modules, module specializations support the notion of conceptual abstraction on the module level [52]. Conceptual abstraction refers to the ability to factor out code and structure shared by several modules into a common “supermodule” which gets specialized independently into different directions. The specializations represent distinct modules that cannot be substituted for the common “supermodule”.

### 3.5 Class Abstractions

Until now we only considered functional modules. With these modules, JAVA’s static variables and static methods get superfluous. Such members could easily be implemented as module members with the benefit of extensibility and improved reusability. Even though functions on the module level can be quite useful to model global behavior, it is more common for object-oriented languages to have modules that contain class definitions. Classes
defined in a module can freely refer to other members of the module as well as to modules required from the enclosing module. The following module defines a class for representing points.

```java
module SPACE {
    class Point {
        Point(int x, int y) { ... }
        int getX() { ... }
        int getY() { ... }
    }
}
```

Module systems for JAVA-like programming languages that allow to abstract over classes are not only difficult in theory, they are also extremely difficult to implement in practice if one wants to stick to JAVA’s compilation model. In such module systems, classes can, for instance, extend classes of required modules for which only the interface might be given. Consequently, at compile time, a compiler has to translate the class without knowing its concrete superclass. This raises implementation issues, but also more fundamental questions, e.g. about not being able to create cycles in the inheritance graph. Ancona and Zucca discuss problems related to this tradeoff between class abstraction and implementation inheritance in greater detail in [4].

Since KERIS is designed to fully support JAVA’s compilation model while being implementable on the standard JAVA platform, we decided not to offer a facility for abstracting over regular classes. Thus, classes on the module level are handled similar to inner classes [27, 31].

### 3.5.1 Virtual class fields

To support reuse and extensibility of types, KERIS introduces the notion of virtual class fields as an alternative type abstraction mechanism. A class field defines a new class by specifying its interface and by possibly giving a concrete implementation, which is typically a reference to a regular class. Here is an example defining an interface, a class, and a virtual class field within one module:

```java
module POINT requires INOUT {
    interface IPoint {
        IPoint(int x, int y);
        int getX();
        int getY();
    }
    class CPoint implements IPoint {
        CPoint(int x, int y) { ... }
        int getX() { ... }
        int getY() { ... }
    }
    class Point implements IPoint = CPoint;
    Point root() { return new Point(0, 0); }
    void print(Point p) { INOUT.write(p.getX() + "/" + p.getY()); }
}
```
Module POINT defines an interface IPoint. The definition of this interface shows that interfaces in Keris can also specify the signature of constructors. In addition to IPoint, module POINT also defines a class CPoint for representing points as well as a class field Point by separately specifying its interface and implementation.

The implementations of the functions print and root show that class fields behave just like regular classes: They define new types, they can be instantiated, and members of corresponding objects can be accessed. The main difference to regular classes is that class fields are virtual and therefore can be covariantly overridden in refined modules. Covariant overriding of class fields includes the extension of the set of implemented interfaces as well as the ability to specify new class field implementations. Here is an example:

```java
module COLORPOINT refines POINT requires COLOR {
  interface IColor {
    COLOR.Color getColor();
  }
  class CColPoint extends CPoint implements IColor {
    ...
  }
  class Point implements IPoint, IColor = CColPoint;
  void print(Point p) {
    super.print(p); INOUT.write(" col = " + color);
  }
}
```

Refinement COLORPOINT specifies that class field Point now also supports the IColor interface and is implemented by the CColPoint class. Furthermore, print is overridden to include the color in the output. At this point, one might wonder what happens to method root of the original module POINT which instantiates class field Point. In fact, for the refined module it now returns a colored point since we were overriding class field Point.

The ability to covariantly refine types (or class fields in our case) is essential for extending object-oriented software. Most object-oriented languages support interface and implementation inheritance. But inheritance alone does not support software refinement or software specialization well. Existing code refers to the former type and often cannot be overridden covariantly in a type-safe way to make use of the extended features. For special cases like binary methods, some languages support the notion of self types [13, 12, 46]. But these are not suitable for mutually referential classes that have to be refined together to ensure consistency [20]. Here, only virtual types are expressive enough [32, 62, 19, 40]. Unfortunately, virtual types rely in general on dynamic type-checking. Therefore recent work concentrated on restricting the mechanism to achieve static type-safety [64, 11]. A formal account of type-safe virtual types is given in [47], which introduces a calculus of classes and objects with abstract type members.

Keris’ class fields are statically type-safe. This is mainly due to the nature of refinements: A refined module subsumes the former module and cannot coexist with the former module within the same context. It rather replaces the former module consistently in explicitly specified contexts. Module specializations do not compromise type-safety either, since they conceptually create completely new modules with class fields that do not necessarily have a (subtype) relationship with the original class fields.
3.5.2 Dependencies Between Class Fields

The previous section explained how to declare and how to evolve virtual class fields. The presented mechanism does not allow to relate different class fields to each other; every virtual class field defines an own isolated class. We need a mechanism similar to subclassing that allows to introduce dependencies between class fields. Since implementation inheritance is difficult to handle modularly, as explained in the beginning of Section 3.5, KERIS only supports the explicit declaration of subtype relationships between virtual class fields. The following code illustrates this.

```java
module GEO requires POINT {
    interface IShape {
        boolean inShape(POINT.Point pt);
    }
    abstract class Shape implements IShape;
    void registerShape(Shape s) { ... }
}
module SHAPES requires GEO, POINT {
    interface IBox {
        IBox(POINT.Point topleft, POINT.Point botright);
    }
    class Box extends GEO.Shape implements GEO.IShape, IBox = {
        Box(POINT.Point topleft, POINT.Point botright) {
            GEO.registerShape(this); ...
        }
    }
    boolean inShape(POINT.Point pt) {
        ...
    }
}
```

In this program, we define an abstract class field `Shape` within module `GEO`. Abstract class fields are like abstract classes: They simply define types, and cannot be instantiated. Thus, there is no need to specify an implementing class for abstract class fields. Module `SHAPES` defines a class field `Box` which extends `GEO.Shape`. This extends declaration defines `SHAPES.Box` to denote a subtype of `GEO.Shape` requiring that `SHAPES.Box` implements at least all the interfaces implemented by `GEO.Shape`. The type checker has to make sure that it is not possible to link refinements of `GEO` and `SHAPES` where this invariant is broken. Thus, such subtype dependencies between class fields promote the consistent refinement or specialization of class field hierarchies.

Here is an example which successfully links modules `GEO` and `SHAPES` in the context of module `GEOSHAPES`.

```java
module GEOSHAPES requires POINT {
    module GEO;
    module SHAPES;
}
```

Imagine, we develop a refinement `XGEO` of `GEO` that adds a new method `scale` to `GEO.Shape`:
module XGEO refines GEO {
    interface INewShape { void scale(int factor); }
    abstract class Shape implements IShape, INewShape;
}

We now cannot simply refine GEOSHAPES and introduce the refined XGEO module in place of the previous version GEO. This would break our dependency, since SHAPES.Box now would not cover the newly introduced XGEO.INewShape interface. Thus, we first have to consistently refine SHAPES as well, such that class field SHAPES.Box also implements interface XGEO.INewShape. It is now possible to link both refinements, as the following code fragment shows. The concrete implementation of XSHAPES.Box is irrelevant and therefore left out.

module XSHAPES refines SHAPES requires XGEO {
    class Box extends GEO.Shape implements GEO.IShape, XGEO.INewShape = ...
}
module XGEOSHAPES refines GEO {
    module XGEO;
    module XSHAPES;
}

4 Generic Module Implementations

The code examples of the previous sections were mainly given to illustrate certain features of KERIS’ module abstractions. This section will now present a series of complete examples that show how extensible modules can be used to safely implement generic software components.

4.1 Family Polymorphism

In this section we use modules as means to encapsulate sets of related classes. Since modules are extensible, it is possible to create refinements and specializations of such class families. The term family polymorphism refers to the ability to statically declare and manage relationships between several classes polymorphically; i.e. in a way that a given set of classes may be known to constitute a family, but where it is not known statically exactly what classes they are [20]. We will proceed by explaining how KERIS supports family polymorphism through extensible modules with class fields.

We start with the implementation of a generic module for representing graphs. Listing 1 shows a definition of a suitable module interface. This module interface defines class fields for graphs, nodes, and edges together with the corresponding interfaces. Since members of module interfaces are never concrete, we define class fields only by specifying their implemented interfaces. Such class fields are called opaque, because they do not reveal their implementation. Please note that abstract class fields, as they got introduced in Section 3.5.2, are also opaque, but in addition, it is not possible to instantiate them.\footnote{This terminology follows the one of JAVA which is unfortunately inconsistent regarding the modifier abstract: Abstract classes define an implementation, whereas abstract methods do not provide any concrete implementation.}
module interface GRAPH {
    class Graph implements IGraph;
    class Node implements INode;
    class Edge implements IEdge;
    interface IGraph {
        IGraph();
        Node[] nodes();
        Node addNode();
    }
    interface INode {
        Edge[] edges();
        Edge connectTo(Node node);
    }
    interface IEdge {
        Node from();
        Node to();
    }
}

Listing 1: A module interface for graphs

module DIRECTED_GRAPH implements GRAPH {
    class Graph implements IGraph = {
        Node[] nodes = new Node[0];
        Node[] nodes() { return nodes; }
        Node addNode() { ... new Node() ... }
    }
    class Node implements INode, IDNode = {
        Edge[] edges = new Edge[0];
        Edge[] edges() { return edges; }
        Edge connectTo(Node node) { ... new Edge(this, node); ... }
        boolean reachableFrom(Node node) { ... }
    }
    class Edge implements IEdge, IDEdge = {
        Edge(Node from, Node to) { ... }
        Node from() { return from; }
        Node to() { return to; }
    }
    interface IDNode {
        IDNode();
        boolean reachableFrom(Node node);
    }
    interface IDEdge {
        IDEdge(Node from, Node to);
    }
}

Listing 2: An implementation of module interface Graph
module WEIGHTED_GRAPH specializes DIRECTED_GRAPH {
  class Node implements INode, IDNode, IWNode = super {
    int shortestPathTo(Node node) { ... edges[i].weight() ... }
  }
  class Edge implements IEdge, IDEdge, IWEdge = super {
    int weight;
    void setWeight(int _weight) { weight = _weight; }
    int weight() { return weight; }
  }
  interface IWNode { int shortestPathTo(Node node); }
  interface IWEdge {
    void setWeight(int weight);
    int weight();
  }
}

Listing 3: A specialization of directed graphs

A possible implementation of module interface GRAPH is given by module DIRECTED_GRAPH in Listing 2. We use anonymous class declarations to provide concrete implementations for all class fields of module DIRECTED_GRAPH. An anonymous class declaration consists of a block defining class members which is optionally preceded by a reference to a super class. The use of anonymous classes is sometimes necessary to give the self reference this the right type. In anonymous classes, this is given the type of the corresponding class field. This mechanism is reminiscent of IDEA's way of typing virtual classes [63]. In Listing 2, the implementation of class field Node needs this to be of type Node, otherwise we could not pass it to constructors of Edge.

Note that in DIRECTED_GRAPH, both Node and Edge extend the set of implemented interfaces specified in module interface Graph. For both class fields this is essential to enable the construction of concrete objects, since module interface GRAPH does not explicitly specify any constructors for them.

In Listing 3 we specialize the DIRECTED_GRAPH module. The specialized version WEIGHTED_GRAPH adds weights to edges. The new implementation of Edge subclasses the previous (anonymous) implementation by referring to this implementation via super. We can now create a system which deals with both weighted graphs and directed graphs and where the type system guarantees that we cannot mix them. The following module definition does not type check because of exactly this reason: In function main we try to link a node of a directed graph with one of a weighted graph.

module GRAPHAPP {
  module DIRECTED_GRAPH;
  module WEIGHTED_GRAPH;
  void main(String[] args) {
    DIRECTED_GRAPH.Graph dg = new DIRECTED_GRAPH.Graph();
    WEIGHTED_GRAPH.Graph wg = new WEIGHTED_GRAPH.Graph();
    dg.addNode().connectTo(wg.addNode());
  }
}
Programming languages like **GBETA** [19] and **SCALA** [46] that support types as object members allow even stronger couplings between members of a class family. Here one could nest the types `Edge` and `Node` in class `Graph` to even disable mixing nodes from different directed graphs. Such languages are more expressive with respect to family polymorphism, but it is not clear how to combine them with software evolution features similar to the ones offered by **KERIS**.

The next section discusses **KERIS’** ability to express families of recursively dependent classes in a modular fashion; i.e. in a way that does not require to define all related classes as members of a single enclosing class or module.

### 4.2 Design Patterns as Module Aggregates

This section describes the usage of modules to develop generic implementations of design patterns in a modular fashion. We pick the **Subject/Observer** pattern as an example [24]. Listing 4 introduces three modules as the building blocks of this pattern. The observer type is defined in module `OBSERVER` by the class field `Observer`. Module `OBSERVER` has to require the corresponding `SUBJECT` module since the observer type refers to the subject. Similarly, module `SUBJECT` requires module `OBSERVER` for defining a class field `Subject`. We have no concrete implementation for events, so the `EVENT` module gets described by a module interface.

We can now link the mutually dependent modules together yielding a single module `SUBJECT_OBSERVER` that represents the complete Subject/Observer pattern. In addition to the aggregated modules we also define a function `attach`. The composed module `SUBJECT_OBSERVER` is a natural place for defining functions that belong logically to the whole pattern, and not to a specific participant.

```java
module SUBJECT_OBSERVER requires EVENT {
    module SUBJECT;
    module OBSERVER;
    void attach(SUBJECT.Subject s, OBSERVER.Observer o) {
        s.add(o);
    }
}
```

We could create refined versions of that pattern with alternative properties, but here, we are mainly interested in specializing it for a specific application. Following the example in [62], we derive a data structure for modeling a window manager by consistently specializing the mutually referential modules `SUBJECT` and `OBSERVER`. We start with the covariant specialization of the `SUBJECT` module.

```java
module MANAGER specializes SUBJECT
    requires WINDOW as OBSERVER, WINEVENT as EVENT {
    interface IManager { ... }
    class Subject implements ISubject, IManager = ...
}
```

Module `MANAGER` also has to specialize the requirements of the original `SUBJECT` module with the `as` construct. This “rewiring” has the effect that all former references to the `OBSERVER` module now refer to module `WINDOW`. The same holds for `EVENT`. Without this
module OBSERVER requires SUBJECT, EVENT {
    interface IObserver {
        IObserver();
        void notify(SUBJECT.Subject subj, EVENT.Event evt);
    }
    class CObserver implements IObserver {
        void notify(SUBJECT.Subject subj, EVENT.Event evt) {
            ...
        }
    }
    class Observer implements IObserver = CObserver;
}

module interface EVENT {
    class Event;
}

module SUBJECT requires OBSERVER, EVENT {
    interface ISubject {
        ISubject();
        void add(OBSERVER.Observer obs);
        void notify(EVENT.Event evt);
    }
    class Subject implements ISubject = {
        OBSERVER.Observer[] obs;
        void add(OBSERVER.Observer obs) {
            ...
        }
        void notify(EVENT.Event evt) {
            for (int i = 0; i < obs.length; i++)
                observers[i].notify(this, evt);
        }
    }
}

Listing 4: A modular Subject/Observer implementation
specialization we could not link module MANAGER with the corresponding module WINDOW since WINDOW is distinct from OBSERVER and therefore cannot play its role.

    module WINDOW specializes OBSERVER
    requires MANAGER as SUBJECT, WINEVENT as EVENT {
        interface IWindow { ... }
        class Subject implements ISubject, IWindow = ...
    }
    module WINEVENT specializes EVENT {
        ...
    }

Finally, we compose the modules to represent the window manager pattern as a specialization of the Subject/Observer pattern. Here we have to specialize the submodules accordingly. We cannot simply override the original SUBJECT and OBSERVER submodules, since our specialized modules do not subsume them.

    module WIN_SYSTEM specializes SUBJECT_OBSERVER requires WINEVENT as EVENT {
        module MANAGER as SUBJECT;
        module WINDOW as OBSERVER;
    }

The type system also enforces the specialization of the EVENT requirement. Otherwise, we would evolve SUBJECT_OBSERVER inconsistently with respect to SUBJECT and OBSERVER, which both specialize EVENT with WINEVENT.

5 Type System

In this section we informally review the basic principles of the type system of KERIS. We explain what restrictions have to be made to ensure statically that a system assembled from modules is sound. Furthermore we explain how the type system helps to evolve software consistently.

5.1 Types

Type systems of JAVA-like object-oriented languages are usually nominal; i.e. types are identified by their names, not by their structure. Not considering the package system and class nesting, reference types in JAVA have simply the form \( C \), where \( C \) corresponds to a class name. In KERIS, on the other hand, classes are typically not defined on the top-level, but rather within modules. Since modules can be arbitrarily nested, a reference type in KERIS corresponds to a class name \( C \) which is qualified with a sequence of module names \( M_1 :: M_2 :: ... :: M_n \) where \( M_{i+1} \) is a submodule of \( M_i \) for all \( i \) and \( C \) is a member of module \( M_n \). This module name sequence identifies uniquely a nested instance of module \( M_n \). Thus, types in KERIS have the general form \( M_1 :: M_2 :: ... :: M_n.C \). Two types are equal, if the class name and the module sequence qualification are equivalent. Module equivalence is considered modulo refinements and module implementations (which can be seen as a special case of refinement), so that all the following types would be considered equal for the modules defined in Section 3.5.2: \( XGEOSHAPES::GEO.Shape \),
GEOSHAPES::GEO.Shape, and XGEOSHAPES::XGEO.Shape. Since types depend on module instances, our system distinguishes between the types O::M.C and O::N::M.C in the following code. Both types refer to the same physical class C, but qualified with different module instances. This distinction is necessary since it is possible to refine both instances of M independently, possibly yielding different versions of class field M.C.

```plaintext
module M {
    class C = {}
}
module N {
    module M;
}
module O {
    module M;
    module N;
}
```

Such a dependent type system is characteristic for strongly typed programming languages supporting family polymorphism [20]. A formalization of dependent types for objects and classes is given in [47].

### 5.2 Type Coherence

The previous section discussed when types are considered to be equivalent. This question is in particular relevant for types appearing in the interface of external modules. Consider the following example in which module interface A defines a class field C. We have two modules M and N which both require a module implementing A. Both of these modules are required by a third module O. The question is now, if the type A.C mentioned in M and the type A.C mentioned in N are equivalent in the context of module O. This would turn N.bar(M.foo()) into a well-typed expression.

```plaintext
module interface A {
    interface I { I(); }
    class C implements I;
}
module M requires A {
    A.C foo() { return new A.C(); }
}
module N requires A {
    void bar(A.C a) {}
}
module O requires M, N {
    ... N.bar(M.foo()) ... ??
}
```

In ML-like module systems it is up to the programmer to declare if both types are considered to be equivalent by explicitly introducing sharing constraints. This sharing by specification approach allows the programmer to introduce only a minimal set of equations identifying types. On the other hand, if a large set of interconnected cooperating modules with many type members is used, it becomes quickly unhandy to specify all the required sharing constraints by hand.

For a language without generic context dependencies like JAVA, the coherence problem is trivial. Two fully qualified type references P.C are always considered to be equivalent, no matter what context they appear, since there will be always exactly one implementation available at runtime. We consider this implicit agreement on types (or, in fact, packages) essential for making the development of huge, recursively dependent, object-oriented libraries practical. Thus, KERIS adopts a similar policy and rigorously identifies types with
the same name and equivalent module sequence qualification, even for types appearing in required, external modules, and without giving up genericity. Since type equivalence in Keris is dependent on a compile-time notion of equivalence of module instances, it is essential for the type system of Keris to enforce that modules which get identified statically in the context of a module like O are actually also implemented at runtime by a single module implementation for all possible module compositions involving this module.

Here is an example for a legal composition of modules M, N, and O. Module P is well-typed, because both modules M and N are referring to the same implementation of module A (which gets required by P).

```plaintext
module P requires A {
  module M;
  module N;
  module O;
}
```

Under the assumption that module AI implements module interface A, the following definition of module Q is not well-typed. This is because now, module N gets linked with AI, whereas module M is required from Q and therefore can never refer to implementation Q.AI for satisfying its own context dependency on A.

```plaintext
module Q requires M {
  module AI;
  module N;
  module O;  
}
```

Please note that the ill-formedness of Q is due to the definition of submodule O, because it is O that requires that modules M and N both access the same implementation for A. Omitting O in the example above would turn Q into a well-typed module.

Regarding the mechanism for handling type coherence, Keris trades simplicity for expressiveness. The example above demonstrates that the explicit identification of modules can indeed restrict the number of possible deployment scenarios for a given module artificially, if external modules get identified unnecessarily. But our experience shows that by specifying minimal views (module interfaces) for external modules, one can often avoid unintended module identifications and improve reusability significantly. Furthermore, a simple mechanism for handling type coherence is essential in practice, since in the presence of software evolution features, complex coherence management facilities get even more complex, ultimately yielding unmanageable formalisms.

### 5.3 Module Composition

Following traditional module systems of imperative languages, Keris associates with every module name implicitly a module interface. This is the basis for inferring the wiring of submodules. **Wiring inference** on the other hand requires that references to modules have to be unambiguous; i.e. the type system has to ensure that a module name identifies an implementing module instance uniquely. We impose a set of restrictions on submodule aggregations to enforce this. In general, the aggregation of a submodule M in module N is subject to the following terms:
• $N$ may not define another submodule, or require a module, which subsumes modules captured by $M$ (*uniqueness*).

• $M$ may not subsume modules captured by $N$ (*linearity*).

• All of $M$’s required modules have to be present either as other submodules of $N$, or have to be required by $N$, or have to be subsumed by $N$ itself (*dependency resolution*).

• $M$ has to identify at least those modules identified by $N$ (*coherence*).

• The class field dependencies specified by $M$ involving required modules have to also hold for the resolved concrete context dependencies in $N$ (*consistency*).

While the uniqueness and linearity restrictions rule out ambiguities, it is the dependency resolution rule which guarantees that all requirements of submodules are met by the deployment context. The coherence rule addresses type equivalence issues discussed in Section 5.2. The consistency rule is responsible for validating class field dependencies in a concrete deployment context. Here is an example for an illegal aggregation of a submodule $M$ in $N$:

```
module interface A {
    interface I { void foo(); }
    class C implements I;
}
module AI implements A {
    interface J { void bar(); }
    class C implements I, J = {
        ...
    }
}
module M requires A {
    class D extends A.C implements A.I = {
        ...
    }
}
module N {
    module AI;
    module M;
}
```

The module definition of $M$ is well-formed because class field $D$ implements all interfaces implemented by $A.C$. As a submodule of $N$, module $M$ would get linked with the other submodule $AI$. But since $AI$ refines $C$ covariantly by implementing an additional interface $J$, we cannot successfully link it with $M$ who’s class field $C$ does not support $J$.

Please note that the linearity restriction from the list above does not rule out cases with indirect recursion; i.e. according to the rules above, it is legal for a module to aggregate an instance of itself indirectly. Here is an example:

```
module M {
    module N;
}
module N {
    module M;
}
```

Since KERIS initializes modules lazily at the time a module gets accessed for the first time, such a program would not necessarily create an infinite number of nested module
instances. On the other hand, it would be straightforward to statically detect arbitrary recursion among submodules and to exclude these cases.

5.4 Module Refinement
Module refinements have to be backward compatible so that an instance of a refinement can replace an instance of the original module. Module refinements inherit all required modules and all submodules from the module they refine. It is possible to add new requirements or to refine requirements to express a simultaneous mutual refinement of several modules. Furthermore, one can covariantly override submodules with instances of refined modules. Like in JAVA, it is not possible to override variables. Method overriding is invariant in the arguments and covariant in the return type, similar to GJ [10].

5.5 Module Specialization
A module specialization yields a new distinct module which inherits members from an existing “prototype”. Excluded from inheritance are the implemented module interfaces. Integrating specialized modules into existing systems requires a mechanism for replacing a module with a specialized version. This rewiring is possible for the requirements of module specializations as well as their submodules, as explained in Section 3.4. The rewiring of requirements is essential to express that a set of modules has to be specialized together. The type checker has to check that depending modules are specialized consistently. Here is an example for an inconsistent specialization:

```plaintext
module M {}
module N requires M {}
module O requires M, N {}
module M1 specializes M {}
module N1 specializes N requires M1 as M {}
module O1 specializes O requires N1 as N {}
```

We define three modules M, N, and O, where module N depends on M and module O depends on M and N. Next we specialize O to O1 and N1 to N where the specialized module N1 now refers to M1 instead of M. The specialization of O rewrites the original requirement N to N1 and inherits the old requirement M. This is a violation of an invariant established in the original module O. Here, module O and depending module N agree on a single implementation of module M, according to the coherence discussion in Section 5.2. In the specialization O1 on the other hand, there is no agreement on M anymore, since the rewired dependency N1 now refers to M1 instead of M. A consistent specialization of O rewrites both N to N1 and M to M1.

For checking the consistent specialization of a module, the type checker first has to collect all specializations of requirements and submodules. In a second step, it has to verify that the agreement on modules in the old system still persists in the rewired system (possibly involving instances of specialized modules). This check can be done modularly, but if involves all interfaces of modules contained in the closure of the old and the new module’s requirements.

This example and the examples given in the previous sections demonstrate how the type system can help to evolve systems consistently by enforcing invariants established by
the original system in refined or specialized variants. Without explicit support for software evolution on the programming language level, programmers are often required to check compatibility and consistency between different versions at runtime or with special tools at link-time.

6 Implementation

We implemented a compiler prototype for Keris. The compiler reads Keris source code and produces standard Java classfiles for classes as well as modules. Since Keris is designed to be a conservative extension of Java that fully interoperates with regular Java classes, the Keris compiler can also be used as a drop-in replacement for javac.

The compiler is currently implemented as an extension of the extensible Java compiler JACO [68, 69]. JACO itself is designed to support unanticipated extensions without the need for source code modifications. It is written in a slightly extended Java dialect using an architectural design pattern that allows refinements in a similar way like Keris. This simplifies our current efforts to reimplement JACO in the programming language Keris itself significantly. With this project we hope to gain experience with the language and its capabilities to statically evolve software through module refinements and specializations.

7 Related Work

Classical module systems like the one of Modula-2 [66], Modula-3 [16], Oberon-2 [45], and Ada 95 [61] can be used to model modular aspects of software components well, but they have severe restrictions concerning extensibility and reuse. These systems allow type-safe separate compilation, but they hard-wire module dependencies by referring to other modules by name. This makes it impossible to plug in a module with a different name but a compatible specification without performing a consistent renaming on the source code level.

The module systems of Oberon-2 and C# [28] allow to define local aliases for imported modules or classes. Here, one can easily replace an imported module with a compatible version just by modifying an alias definition. Such a modification would be destructive and would require a global recompilation, but it would not require extensive source code renaming.

Initially, functional programming languages introduced module systems that obey the principle of external connections [22], i.e. the separation of component definition and component connections. These module systems maximize reuse, but they yield modules that are not extensible, since everything is hard-wired internally. Module systems with external linking facilities include SML’s functors [39] and MzScheme’s units [23]. Opposed to ML functors, units offer separate compilation of independent modules with cyclic dependencies. Units provide first-class module abstractions and linking facilities to compose modules hierarchically. A general problem of unit-style module systems is scalability due to modules importing fine-grained entities like classes, functions, etc. and due to explicit module wiring. For this reason, MzScheme offers signed units that support bundles of variables, called signatures, which get linked in one step [22].

In [50], Pierce compares various module systems regarding genericity and coherence
mechanism. He focuses on practical issues like scalability to show that some more advanced module system features like genericity are sometimes really needed to build large software systems effectively.

Only recently, proposals have been put forward to bundle class-based object-oriented languages with similar module systems [21, 4]. So far, we only know about two attempts to integrate a module system directly into the JAVA programming language. The proposal by Ancona and Zucca is rather theoretical, leaving unclear if their work is feasible in practice [4].

Independently to our work, Ichisugi and Tanaka observed that extensibility on the level of modules greatly enhances the ability to extend applications [30]. Ichisugi and Tanaka describe a practical module system for JAVA based on the notion of difference-based modules. Their modules are solely linked by a form of multiple inheritance which also merges module members. Since their modules are not expressive enough to abstract over context dependencies (which are hard-wired), this module system must be seen rather as a tool for aspect-oriented programming [35] than for developing reusable, context independent software components. In Ichisugi and Tanaka’s language, modules get exclusively linked by inheritance. Based on a similar idea, we investigated in former work a component calculus that explains component composition in terms of component refinements [67]. This component calculus supports a mixin-based composition scheme.

Duggan and Sourelis propose mixin-modules to make ML modules extensible [18]. An alternative proposal which is targeted towards OCaml [37] got recently published by Hirschowitz and Leroy [29]. Their work is based on CMS [3, 5], a simple but expressive module calculus which can be instantiated over an arbitrary core calculus. The calculus supports various module composition mechanisms including mixin module composition with overriding. The work on mixin-based composition goes back to Bracha who observed that inheritance can be seen as a general mechanism for modular program composition [9, 8]. With his work on the programming language JIGSAW [7], he lifts the notion of class-based inheritance and overriding to the level of modules. A consistent refinement of a family of classes is possible with the notion of mixin layers, introduced by Smaragdakis and Batory [56]. Related to mixins is the concept of delegation. Integrated into a statically typed object-oriented language, delegation yields a powerful mechanism for object-based inheritance [36, 15].

Rüping analyzes the modularity of object-oriented systems during design and specification in [52]. He substantiates the need for modules in object-oriented languages as a means to encapsulate cooperating classes. Our module refinement and specialization mechanisms implement his abstract notion of compatibility between modules which is supposed to facilitate the type-safe extension of systems by the substitution of compatible modules.

Linguistic abstractions for component-oriented programming often have similar properties like module systems. Component-oriented programming languages that are built on top of JAVA-like object-oriented languages are COMPONENTJ [54, 53], ACOEL [57], and ARCHJAVA [1]. [67] gives a short overview over these languages. JIAZZI [42] is a system for creating large-scale binary components in JAVA based on MzSCHEME’s units. JIAZZI’s units are conceptually containers of compiled JAVA classes with support for well-defined connections, externally specified through a set of imported and exported classes.

Component-oriented programming languages feature concepts originating from architectural description languages [43] like ACME [26], Aesop [25], Darwin [41], Rapide [38],
Wright [2], SOFA/DCUP [51] etc. In general, architectural description languages are used to specify a software architecture formally. A software architecture describes the organization of a software system in terms of a collection of components, connections between these components, and constraints on the interactions [49, 55, 59]. By using architectural description languages, the details of a design get explicit and more precise, enabling formal analysis techniques. Furthermore, they can help in understanding the structure of a system, its implementation and reuse.

8 Conclusion

The paper presented Keris, an extension of the programming language Java with linguistic support for the evolution of software components. The main contributions are

• a module system that combines the benefits of classical module systems for imperative languages with the advantages of modern component-oriented formalisms. In particular, modules are reusable, generic software components that can be linked with different cooperating modules without the need for resolving context dependencies by hand. Instead, Keris implicitly infers the module-wiring.

• a module composition scheme based on aggregation that makes the static architecture of a system explicit, and

• a type-safe mechanism for extending atomic modules as well as fully linked systems statically. This mechanism relies on two concepts: module refinements and module specializations. Both of them are based on inheritance on the module level. While refinements yield new backward compatible versions of existing modules, specializations are used to derive new (independent) modules from a given “prototype”. Keris’ extensibility mechanism is non-invasive; i.e. the extension of a module preserves the original version and does not require access to source code. Thus, extending modules does not invalidate existing code. The extensibility mechanism is expressive enough to express mutual dependent refinements and specializations, allowing to refine or specialize a complete system consistently.

The overall design of the language was guided by the aim to develop a pragmatic, implementable, and conservative extension of Java which supports software development according to the open/closed principle: Systems written in Keris are closed in the sense that they can be executed, but they are open for extensions that add, refine or replace modules or whole subsystems without planning extensibility ahead. Another constraint was that we did not want to change Java’s compilation model or use a modified target platform.

The Keris compiler is based on an extensible Java compiler developed in previous work [69]. We are currently reimplementing this compiler in Keris for two reasons: First, it enables us to bootstrap the system. Furthermore, we would gain experience in using Keris for building large, extensible software.

Acknowledgments

I would like to thank the reviewers of the First International Workshop on Unanticipated Software Evolution for their detailed and constructive comments. Furthermore, I am grate-
ful to Martin Odersky and Christoph Zenger for many fruitful discussions about related topics.

References


[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.


[37] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.00, documentation and user’s manual, April 2000.


