



ELSEVIER

Available online at www.sciencedirect.com

 ScienceDirect

Electronic Notes in
Theoretical Computer
Science

Electronic Notes in Theoretical Computer Science 164 (2006) 81–96

www.elsevier.com/locate/entcs

Incremental Confined Types Analysis

Michael Eichberg¹ Sebastian Kanthak² Sven Kloppenburg³
Mira Mezini⁴ Tobias Schuh⁵

*Department of Computer Science
University of Technology
Darmstadt, Germany*

Abstract

Research related to alias protection and related concepts, such as, confined types and ownership types has a long tradition and is a promising concept for the design and implementation of more reliable and secure software. Unfortunately, the use of these concepts is not widespread as most implementations are proofs of concept and fall short with respect to the integration with standard software development tools and processes.

In this paper, we discuss an implementation of confined types based on Java 5 annotations. The contribution of this paper is twofold: First, we discuss the incrementalization of the confined types analysis and second, we present the integration of the analysis into Eclipse using the static analysis platform Magellan.

Keywords: Aliasing, confined types, ownership types, Eclipse, static analysis

1 Introduction

Unintended aliasing is causing many kinds of problems. For example aliasing makes modular reasoning more difficult, as it is hard to reason about the effect of updating an object *o* when it is unknown which other objects also keep a reference to *o*.

Besides being a source of programming errors that can be detected when testing an application, unintended aliasing can also lead to security errors, which are hard to detect using standard development techniques. For example, when a reference to an object is passed to another object and, hence, an alias is created for the first object, then the alias can later on be used to update the first object in an unanticipated manner. In [19] a security breach caused by a reference leaking bug in the JDK 1.1 is discussed (shown in Listing 1). In the JDK's implementation, each instance of a

¹ Email: eichberg@informatik.tu-darmstadt.de

² Email: kanthak@st.informatik.tu-darmstadt.de

³ Email: kloppenburg@informatik.tu-darmstadt.de

⁴ Email: mezini@informatik.tu-darmstadt.de

⁵ Email: schuh@informatik.tu-darmstadt.de

```
1| public class Class {  
2|   private Identity[] signers;  
3|   public Identity[] getSigners() {  
4|     return signers;  
5|   } }
```

Listing 1: `Class.getSigners()` without Confined Types

Java `Class` object holds an array of signers (Line 2) that represents the principals under which the class acts. The problem is that the `getSigners` method returns a reference to the original `signers` array (Line 4). Hence, the attackers can then freely update the signatures based on their needs.

To solve the problems related to the creation of unintended aliases, we need means to enforce that important data structures can not escape the scope of a well defined protection domain. For example, to assure that the reference to the original signers array does not escape the declaring class. In [19], Vitek et Bokowski propose the concept of Confined Types to solve issues related to object aliasing.

In this paper, we present an incremental analysis for the confined types concept proposed in [19] and integrate this analysis into the incremental build process of the Eclipse IDE [7]. One goal of our work was compatibility with the Java language specification and existing tools. This was a major reason why we have chosen Confined Types [19]; using Java annotations we were able to simulate the necessary language extensions proposed by Vitek et Bokowski.

The main contribution of our work is to provide an implementation of the confined type checking that is tightly integrated with a standard software development environment and where the analysis exhibits a behavior that is indistinguishable from other (standard) compile time analyses. This fits well in the development philosophy supported by modern IDEs such as Eclipse, where the developer expects to see e.g., typing problems as soon as they emerge as the project evolves.

In general, we argue that — whenever possible — checking various program properties should be done by IDEs. This avoids bloated compilers and ensures that application-specific checkers can be introduced when needed. However, (re)checking the entire project after a change is prohibitively expensive w. r. t. the time required for the analysis. Hence, violations of the typing rules for confined types should be checked for incrementally.

In vein of these considerations, we have implemented the confinement rules defined in [19] using the open, extensible static analysis platform Magellan [9], which is tightly integrated into the Eclipse IDE [7]. By choosing Magellan and Eclipse as the underlying frameworks many issues related to tool adoption [1,10] are already solved. By building on top of Magellan, our analysis is automatically integrated with the incremental build process. Hence, the user will — after activation — perceive no difference between the checks carried out by the standard Java compiler and our analysis. This flattens the learning curve, as it is not necessary to learn how to use the tool, provided the developer is already familiar with Eclipse. Additionally, since we (re)use the standard Eclipse views to visualize errors no user

interface related issues arise.

This paper is structured as follows: In the following, we first give an overview of confined types. After that, we introduce the static analysis platform Magellan on top of which we have build our analysis. We continue with a presentation of the implementation of the confined types analysis, and in particular, the issues related to the incrementalization of the analysis. After that, we evaluate our approach by using confined types in a large project. We conclude with a discussion of related work and a summary.

2 Confined Types

Confined Types were proposed by Vitek and Bokowski [19] as a machine checkable programming discipline that prevents leaks of sensitive object references. A motivation for their work was the security breach mentioned in the introduction (shown in Listing 1).

A possible solution to avoid the breach is a programming style that encourages the developers of classes with sensitive information to return a reference to a copy of the sensitive data, in our case a copy of the signers array. While programming styles cannot be enforced, using confined types ensures that none of the key data structures used in code signing escape the scope of their defining package.

For this purpose, types whose instances should not leave their defining package are marked as *confined*. *Confinement* ensures that objects of a confined type can only be accessed within a certain protection domain. A type is confined to this domain if all references to objects of that type originate from within the domain. Code outside the protection domain is never allowed to manipulate confined objects directly. In contrast to existing access control mechanisms in Java (such as the Java `private` keyword), confinement constrains access to object references rather than classes. It prevents class-based restrictions from being circumvented by casting the protected object to one of its unrestricted super-types.

In this paper, we describe an incremental analysis for the proposal in [19], integrated into the incremental build process of the Eclipse IDE. As proposed in [19], we also use Java packages as protection domains. Instead of the new modifiers, `confined` and `anon`, introduced in [19], we use the metadata facility (*annotations*) introduced in Java 5.0 and define two annotation types: `@confined` and `@anon`.

Listing 2 shows, how the code from Listing 1 can be rewritten using confined types. The annotation `@confined` is used with a class, whose objects should be confined to the containing package. In Listing 2, annotating `SecureIdentity` as `@confined` (Line 3) enforces references to `SecureIdentity` objects to be confined to the package `java.security`. Thus, code outside this package can never access instances of type `SecureIdentity`. Renaming the old `Identity` class to `SecureIdentity` and introducing a new `Identity` class (Line 4 – 8) preserves the functionality of the original interface.

The `@anon` annotation enables confined types to safely use methods from unconfined types. Methods that do not reveal the current object's identity are marked

```

1| package java.security;
2| abstract class AbstractIdentity { @anon equals(){...}; }
3| @confined class SecureIdentity extends AbstractIdentity { ... }
4| public class Identity {
5|     SecureIdentity target;
6|     Identity(SecureIdentity t) { target = t; }
7|     ... // public operations on identities;
8| }
9| public class Class {
10| private SecureIdentity[] signers;
11| public Identity[] getSigners( ) {
12|     Identity[] pub = new Identity[signers.length];
13|     for (int i = 0; i < signers.length; i++)
14|         pub[i] = new Identity(signers[i]);
15|     return pub;
16| }
17| }

```

Listing 2: Class.getSigners() using Confined Types

as *anonymous* by annotating them with `@anon` to show this intention and to make this property checkable⁶. In Listing 2, the method `equals` in line 2 is marked with `@anon` to show that it never reveals the current instance’s identity (`this`-reference). Therefore, `SecureIdentity` can safely extend `AbstractIdentity` and call `equals` on `this`, because no method marked `@anon` will breach the confinement.

The constraints in Table 1 and 2 are defined in [19] and define the semantics of `confined` and `anon`. Constraints in Table 1 restrict class and interface declarations ($C1$, $C2$), prevent widening ($C3$), hidden widening ($C4$, $C5$), and transfers from inside ($C6$) and outside ($C7$, $C8$) the protection domain. The rules defined in Table 2 constrain the usage of the self-reference `this` in method implementations, so that `this` is not revealed to code outside the method.

$C1$	A confined class or interface must not be declared public and must not belong to the unnamed global package.
$C2$	Subtypes of a confined type must be confined as well.
$C3$	Widening of references from a confined type to an unconfined type is forbidden in assignments, method call arguments, return statements, and explicit casts.
$C4$	Methods invoked on a confined object must either be non-native methods defined in a confined class or be anonymous methods.
$C5$	Constructors called from the constructor of a confined class must either be defined by a confined class or be anonymous constructors.
$C6$	Subtypes of <code>java.lang.Throwable</code> and <code>java.lang.Thread</code> may not be confined.
$C7$	The declared type of public and protected fields in unconfined types may not be confined.
$C8$	The return type of public and protected methods in unconfined types may not be confined.

Table 1
Constraints for confined types

Using confined types as an extension to the Java type system, the programming style of returning only copies of sensitive data can be supported in such a way that once a type is marked as `@confined`, the safety of the program with respect to avoiding unintended reference leaking can be guaranteed.

⁶ Another possibility would be to infer the `@anon` property. But having it explicit as an annotation in the code serves as a documented design decision.

A1	The reference <code>this</code> can only be used for accessing fields and calling anonymous methods of the current instance or for object reference comparisons.
A2	Anonymity of methods and constructors must be preserved in subtypes.
A3	Constructors called from an anonymous constructor must be anonymous.
A4	Native methods may not be declared anonymous.

Table 2
Constraints for anonymous methods

3 Magellan

In this section, we discuss the static analysis platform Magellan. Magellan is a generic, extensible platform for static analyses, which is tightly integrated into the Eclipse IDE. The part of the architecture of Magellan relevant for this paper is depicted in Fig. 1. The types in the highlighted area (`Checker`, `ProblemsViewRE` and `Report`) are extended or used by classes of our confined types analysis. In the following, we briefly discuss the functionality of the central classes and interfaces and the interaction between them.

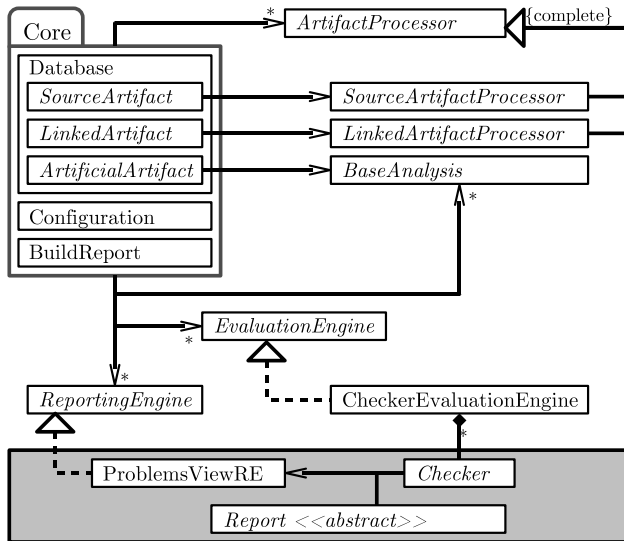


Fig. 1. Diagram of the main classes and interfaces

3.1 Source Artifact Processors

Source artifact processors create representations of program elements defined in the files of a project. The representations, which are appropriate for static analysis are called *source artifacts* and are stored in the database that is part of Magellan’s core module; the database is basically a map that associates an Eclipse resource with artifacts generated by the processors.

The source artifact processor relevant for the confinement analysis uses the Java Bytecode Analysis Toolkit BAT [8] to create a quadruples⁷ based representation of

⁷ Please note, the term *quadruple* and *3-address instruction* are used interchangeable [17, p. 479].

Java class files. The representation is similar to the *Jimple* representation of the Soot framework [18] and is essentially a register based representation of Java bytecode. A quadruples based representation facilitates many analyses, in particular data-flow analyses [17,18], when compared with a direct representation of Java bytecode as generated by other bytecode toolkits, such as e. g., BCEL [2].

3.2 Linked Artifact Processors

A linked artifact processor creates representations of resources that are not directly defined as part of the project, but which are relevant for the analysis of the project. E. g., classes in the Java runtime library are not defined as part of the project, but information about them is required by many analyses.

Representations generated by linked artifact processors are called *linked artifacts* and are also stored in the database. The linked artifact processor used by the confined types analysis processes the source artifacts generated from Java class files. The algorithm that this processor uses to determine the set of linked artifacts to add to the database is described next. The representations of all classes defined in libraries that are directly used in the implementation of the classes of the project are added. Next, the same process is recursively applied to each linked artifact added previously until every class used by any other class is added.

In particular, this algorithm ensures, that representations of the super types of every used type are made available. For illustration, assume that the only class in our project is the following:

```
class A { java.lang.Iterable I; }
```

In the first step, the algorithm adds a representation of the interface `Iterable` to the database — the type of the declared field. Further, the class `java.lang.Object` is added, since every class inherits from it. In the second step, the representations of `Object` and `Iterable` are analyzed. Since `Object` is the top-most type and `Iterable` does not extend any interfaces, no further classes need to be added. To reduce the size of the database, private methods and fields, as well as the methods' implementations are omitted.

3.3 Base Analyses

The program model generated by the source and linked artifact processors is enriched and rendered more precise by applying base analyses that exploit general-purpose program analysis techniques, e.g., class hierarchy, control-flow or data-flow analysis. Our confinement analysis uses the following two base analyses provided by Magellan: (a) the hierarchy analysis to make information about the super-/subtypes of a class directly available, and (b) an analysis to bring the quadruples representation in SSA form [6]. When the representation is in SSA form, the local variables' definition-use and use-definition information is directly available. This enables a straightforward implementation of the check that the `this` reference of a confined type is not passed to another object. For each value passed to another object or returned by the method, we have to check if the `this` reference is potentially assigned

to it. To do so, we analyze the explicitly available use-definition information.

3.4 Evaluation Engines

Conceptually, an evaluation engine is a mediator between the Magellan core and a set of so-called checker modules. A checker module analyzes the models about the program, generated by the processors and base analyses, to derive higher-level information about the “correctness” of the program. The result of a checker is directly presented to the end user of a Magellan enabled IDE. For example, a result could be that a confinement rule [19] of a class is violated. The evaluation engine we are using for the confinement analysis enables to write checkers that directly work on the quadruples based representation. This evaluation engine provides a lightweight plug-in interface: Each checker must implement a small **Checker** interface to enable the evaluation engine (a) to determine the analyses and processors required by the checker and (b) to start the evaluation process.

3.5 Reporting Engines

A reporting engine displays the results of an analysis. During the evaluation of the database, reports that describe findings of the checkers are generated and passed to the reporting engines. Each reporting engine supports a specific reporting format. In our case, we use the simplest form of a report: a short descriptive text such as “*this must not be passed to another class*” associated with a particular artifact element. The reporting engine for this simple format uses the **Problems View** of Eclipse to display the generated reports. These reports consist of a short message, a severity level, a reference to the underlying resource and the specification of a source range to which the message refers.

3.6 The Magellan Core

The Magellan core is responsible for controlling the analysis process. The analysis process is triggered by an incremental or a full build. We will first describe the incremental build process. A high-level overview of the analysis process triggered by an incremental build is depicted in Fig. 2.

First, a **BuildReport** is created and used to record all changes to the database. The core uses the information passed to it (by Eclipse) to remove all artifacts from the database whose underlying resource has changed or was removed. The removed artifacts are added to the build report and are available until the end of the analysis process. The core passes each resource that has changed or which was added to all processors to obtain respective artifacts. When an artifact is returned, the core adds it to the database and to the build report. Second, the core passes the build report to the first linked artifact processor. The processor uses the information stored in the database and in the build report to determine the set of resources for which it needs to create linked artifacts. Third, the base analyses are executed. After performing all analyses, the build report also records the information about

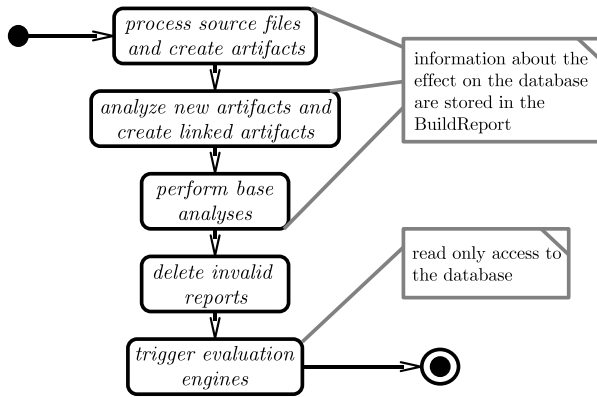


Fig. 2. Activity diagram of the build process

the artifacts where the analysis information has changed. Finally, the core calls the evaluation engines which then execute the registered checkers.

In case of a full build, the process is basically the same as described above, except that first the entire database is cleared and the source artifact processors are called for all files of the project. After that, the same steps are executed as in case of an incremental build.

4 Incremental Analysis

As stated in [19], checking the confinement rules is modular in the sense that each class can be analyzed separately. However, in addition to modularity and dynamic loading [19], our aim is to also support (a) continuous checking of confinement constraints during a programming task, and (b) IDE-Integration of the checking process with an integrated error reporting and source code navigation, as illustrated by the screenshot in Fig. 3.

In such a setting, checking all constraints on all classes after every change is obviously prohibitive in terms of incremental build performance. However, determining which classes have to be reanalyzed after a set of arbitrary changes to the project's source code is non-trivial. For an example of how a small change can impact the confinement rules at a seemingly unrelated location, consider Listing 3.

```

1| package x;
2|   public class X1 {
3|     @anon public void m() { /* ... */ }
4|   }
5|   public class X2 {
6|     public void m() { /* ... */ }
7|   }
8|
9| package y;
10|  public class Y extends X1 { } /* change: ... extends X2 */
11|
12| package z;
13|  @confined class Z extends Y { /* ... */ }
14|  class W {
15|    public void foo() {
16|      Z z = new Z();
17|      z.m(); /* will violate C4 after change */
  
```

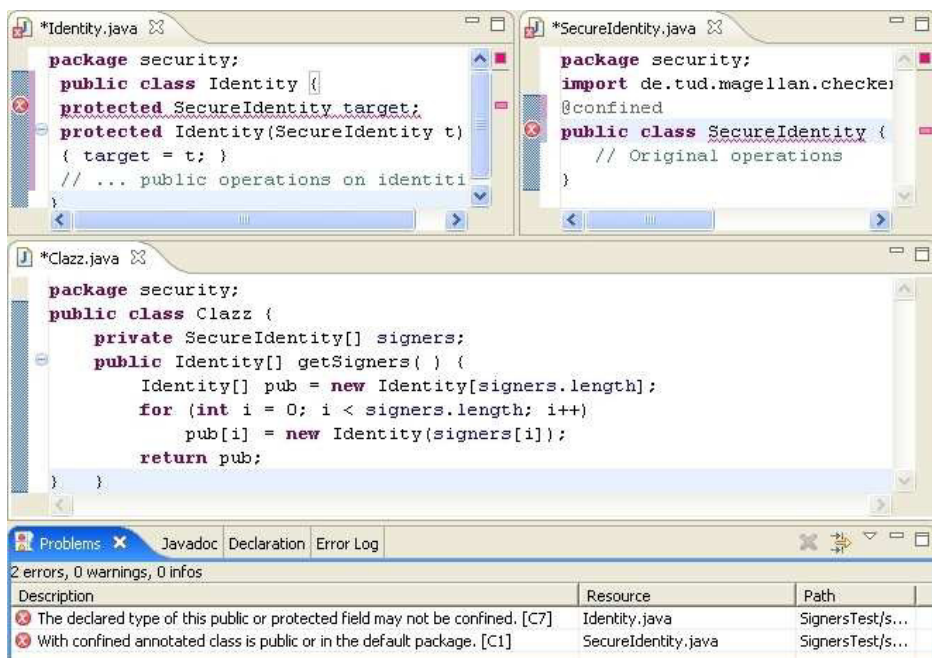


Fig. 3. Screenshot of Eclipse when using Confined Types

18| } }

Listing 3: Indirect violation of confinement constraints

The example consists of Java classes in three different packages. Class W calls a method m on a confined class Z . $C4$ is satisfied because Z inherits m from class $X1$ where it is declared anonymous. Now, let us assume that Y is changed to inherit from $X2$ instead of $X1$. Since $X2$ does not declare m as anonymous, the method call in Line 17 now violates constraint $C4$. Hence, a change in package y (which does not contain any confined or anonymous declarations) yields a confinement error in a class in package z that is neither a subtype nor a supertype of the changed class Y .

The example shows that when a class changes, it is not sufficient to only check classes in the same package / protection domain or all super-types and subtypes of the changed class. We therefore employ a more systematic approach to develop an incremental algorithm for checking the confinement rules.

Our checking algorithm is designed in two steps. First, given a list of classes that have been changed a set of classes is identified that must be reanalyzed to discover any new constraint violation and to remove any error message for constraints that are no longer violated. Next, the constraint rules are checked for all classes returned by the first step. Whenever a check fails an error report for the Eclipse problems view is created and presented to the user (see Fig. 3). Hence, after editing a source file the developer is immediately informed about constraint violations.

We regard all the constraints from Table 1 and Table 2 as predicates over classes,

respectively over methods. For any class c , $C_i(c)$ is true, if and only if c satisfies C_i for any method m , $A_i(m)$ is true only if m satisfies the constraint A_i . Each predicate can be evaluated on its own, since the definitions of the constraints do not depend on each other. For example, for a class c to satisfy constraint $C4$ it suffices that methods called on confined types within c are declared as anonymous. Whether these methods, in turn, satisfy the constraints for anonymous methods is irrelevant for $C4$, though. The reason is that error messages are directly related to predicates that are not fulfilled. Violations of the constraints for anonymous methods will be displayed as separate errors when analyzing the respective methods.

Now we can state our problem as follows: Given a program, the predicate values for all its classes and methods, and a set of classes changed in the process of an incremental build, update the predicate values so that they reflect the program changes. This update process should be *correct* in the sense that it produces the same results as a whole-program analysis.

Since a constraint must only be reevaluated if some information it depends on has been invalidated by a program change we determine for each constraint the set of information it depends on.

Before doing so, we slightly modify the constraints $C2$ to $C2'$: “If a direct super-type of a type t is confined, t must be confined as well.”, and $A2$ to $A2'$: “If a method m directly overrides an anonymous method, m must be anonymous as well.” These modifications, while reducing the information on which the values of $C2$ and $A2$ predicates depend on, do not affect the semantics of the confined types: A program satisfies all the constraints from Table 1 and Table 2 if and only if it satisfies them with $C2$ and $A2$ replaced by $C2'$ and $A2'$.

We start our analysis by investigating the rules for anonymous methods, as defined in Table 2.

- $A1(m)$ depends on the anonymous attribute of all methods called on `this` inside m . These methods have been declared either in m 's class or in a super-type of the latter. Hence, for any changed class c , $A1(m)$ must be reevaluated for any m in c or any of its subtypes.
- $A2'(m)$ depends on the anonymous attribute of the method overridden by m . Since such a method must be declared in a super-type of m 's class, the same invalidation strategy as for $A1$ applies.
- Since calls to constructors from within a constructor can be seen as a special kind of method calls on `this`, we can treat $A3$ in the same way as $A1$.
- $A4$ does not depend on any non-local information. Thus, it suffices to reevaluate $A4$ on all methods of a changed class.

This leads to the following incremental algorithm for checking the constraints from Table 2. Whenever a type t changes, we have to reevaluate constraints $A1$ – $A3$ on all subtypes of t (including t itself). Constraint $A4$ only has to be reevaluated for types that have been changed.

Next, the constraints in Table 1 are analyzed in the same way.

- $C1(c)$ only depends on information from the class c . Thus, for every c , which has changed, $C1(c)$ must be reevaluated.
- $C2'(c)$ depends on the confined attribute of all direct super-types of c . Thus, we have to reevaluate $C2'(c)$ for any c that is a direct subtype of a changed class c' .
- $C3(c)$ depends on the confined attribute of the types used in widenings inside one of c 's methods. The value of $C3(c)$ can change only if either c is changed (so that the list of widenings performed inside c has changed) or if the confined attribute of a type t that is used in a widening changes. For each such t , the following holds: t has been confined at some point (i. e., before or after the change), hence, t is defined within the same package as c . Therefore, for each class c whose confined attribute has changed $C3$ needs to be reevaluated for any class in the same package as a class c .
- $C4(c)$ depends on method calls in c where the static type of the receiver is confined. More specifically, it depends on the confined attribute of the method's declaring type and the method's anonymous attribute.

Since the static receiver type is confined, it must be in the same package as the class that contains the method call. Thus, whenever the confined attribute of a type t changes, $C4(c)$ must be reevaluated for any class c in the same package as t to recheck all relevant method calls on t .

Additionally, we have to reevaluate $C4$ when the anonymous attribute of the called method changes. This can happen indirectly as we have seen in the example from Listing 3. Thus, whenever a type t is changed we have to determine all classes that call a method on a confined subtype t' of t . Since a confined type can only be package visible, such a class must be in the same package as t' . For every confined subclass t' of t we check $C4(c)$ for all classes c in t' 's package.

- The constraint $C5(c)$ considers constructor calls in constructors of confined classes. Since constructors are not inherited in Java, they have to be in the same class or in the direct superclass (can be called via `super(...)`). This implies that $C5$ depends only on the class itself and its superclass. When a class c is changed, we reevaluate $C5$ for c and all direct subtypes.
- $C6(c)$ depends on all super-classes of c . Thus, it suffices to reevaluate $C6$ for all subclasses of c whenever c is changed. As an optimization, we can ignore changes to c that do not change c 's super-types.
- $C7(c)$ can change whenever the confined attribute of a type used in a public or protected field declaration of c changes. Since such a field type either was confined before the change or has become confined after the change, it has to be in the same package as c . Thus, whenever a type t changes $C7$ needs to be reevaluated for all classes in the same package as t .
- The constraint $C8(c)$ checks return types of methods that are declared as public or protected. The strategy for evaluating $C8$ is the same as for $C7$.

Given a set of files that have been changed, we process every constraint separately. For every changed class we compute the set of classes that have to be

reanalyzed and then reevaluate the constraint against all classes in this set⁸. This process is correct even if multiple changes have been performed, because it analyzes the same classes that would have been analyzed if an incremental analysis had been performed after every change.

By definition, the rules for computing the set of classes to be checked after a change guarantee that a constraint is reevaluated if any information it depends on has been invalidated. Hence, the value of all predicates is the same as if they had been evaluated by performing a whole-program analysis. Thus, our incremental algorithm is correct. Regarding its efficiency, with the current rules we often have to reevaluate a constraint for all subtypes of some type. Obviously, this may be a very big set. Suppose, for example, that the class `Object` is changed somehow. Now, constraints *A1–A3* for example have to be reevaluated for all subtypes of `Object` which essentially is every type.

A possible optimization is to use a call-graph analysis to reduce the reevaluations of constraints *A1* and *C4*. This is because, we could determine all method call statements that are affected by a given change. For the change from Listing 3, for example, the call-graph analysis would tell us that the method called in Line 17 has changed and we could reevaluate *C4* for this location. This avoids having to check constraints *A1* and *C4* for all classes in a package. The challenge, of course, is to make call-graph analysis incremental as the cost would be prohibitive otherwise and to make it fast enough to pay off compared to our current algorithm.

5 Performance Evaluation

The runtime complexity of static analyses is an important obstacle for their widespread adoption; performance is especially crucial for an integration into the build process. To assess our analysis in this respect, we measured its runtime while refactoring the Java runtime library to implement the suggestions made in [12]. The experiment was conducted on a dual Xeon 3.0Ghz workstation with 2GB RAM running Windows XP and the Sun Java 5 JDK.

We edited the “public” part of the Java 5 runtime library (`rt.jar`) delivered with the Sun JDK, which consists of 4992 classes in `java.*`, `javax.*` and `org.*`. Furthermore, 441 classes were added to the database by the linked artifact processor for classes in `sun*` and `com.sun*`; these classes are used in the implementation of the public classes.

To keep the artifacts, the hierarchy information and the results of the confinement analysis in memory $\approx 85\text{MB}$ are required. The overall time for the first analysis process (full build, without confinement annotations) of the project is 46.5 seconds; the supporting analyses require 45.7 seconds and the analysis of the confined types (`Confinement Analysis`) 0.7 seconds.

The time required to perform the analysis during incremental builds is shown on the y-axis in Fig. 4. The numbers on the x-axis are identifiers for different

⁸ For simplicity, we just compute the union of all these sets and check all constraints against every class in this set.

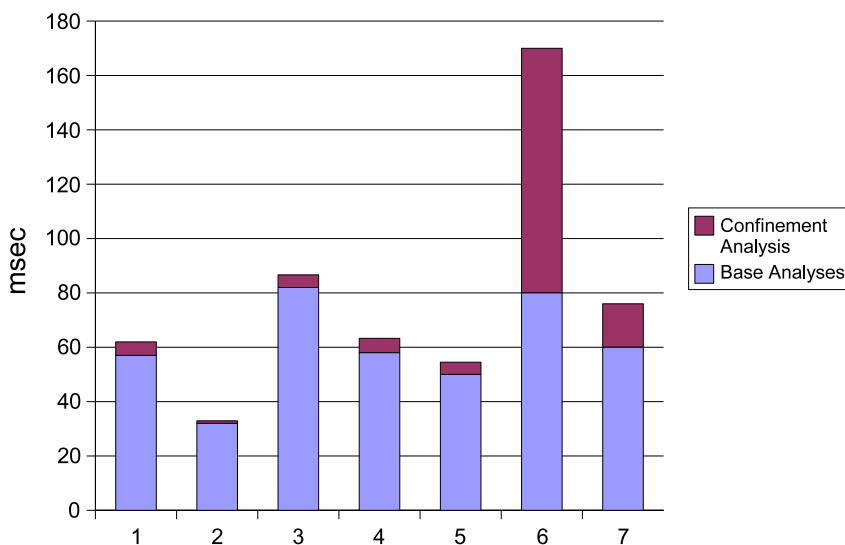


Fig. 4. Incremental build times (msecs)

refactorings that we performed:

- (i) A complex class (1578 LOC) is changed; the change does not affect confined types.
- (ii) The implementation of an anonymous method is changed, whose declaring class does not have confined subclasses.
- (iii) A method in a confined class is changed; the change does not violate any confinement constraints.
- (iv) A method is annotated as anonymous which is inherited and used by a confined subclass.
- (v) The annotation added in the previous case is removed to force the creation of an error message.
- (vi) A class is annotated as confined in a package that previously did not define any confined types.
- (vii) The confined annotation of the most recently annotated class in a package is removed.

The results show that in case of an incremental build the time required to perform the necessary analyses is in general less than 200 milliseconds⁹. Further, the additional amount of memory required is at most 85MB. These results indicate that it is feasible to run the confinement analysis along with the incremental build process.

⁹ Please note that the automatic parallelization of the artifact processors reduces the required time for processing the source files by $\approx 35\% - 40\%$ when compared to a single CPU configuration.

6 Related Work

When dealing with aliasing, four categories of work are considered [14]: detection, prevention, control and advertisement of aliasing. The works we are interested in, mostly fall under the category of prevention and control.

The notion of alias protection for object-oriented languages was introduced by Hogg [13] in order to enable modular reasoning for groups of classes. These groups are called *islands* and ensure the restriction of aliasing to classes on the island. Hogg differentiates between static and dynamic aliases. Static aliases are aliases via instance variables and dynamic aliases are those via parameters or local variables. Static aliasing can lead to undesired side effects in later invocations of the aliased object. Dynamic aliases were seen as unproblematic, because they disappear at the end of the execution of the method in which they are defined. Means to control static aliasing were introduced with islands. Islands are the transitive closure of a set of objects accessible from a *bridge* object. A bridge object is the sole access point to a set of instances that make up an island.

To ensure that no static aliases are created from outside the island to objects on the island, the methods of the bridge object are restricted. Only methods with parameters and return values that either do not modify the state of the system, or have only parameters and return values that have at most one static alias are allowed. This avoids the creation of unwanted aliases. For example, a return value of a method can be tagged with *unique* to state that exactly one reference to its value exists. The value can only be assigned to other variables, if the original reference is released.

The full encapsulation of aliases of this approach is too restrictive for many common design idioms used in OO programming. E. g., no object could be a member of two collections simultaneously if either collection was fully protected against aliases. In this case, one collection would be an island, prohibiting that references to its members show up outside the island.

In [15], Noble et al. present a more flexible approach to control aliasing when compared with *islands*. The approach taken by Noble et al. is to enable aliasing by introducing explicit aliasing modes. The authors differentiate between the *representation* of an object, which corresponds to its fields, and *arguments*, which are parameters to methods of the object. The representation of objects should only be accessible via the object's interface, e. g., in Java fields would have to be marked as `private` and aliases to them should not be returned via getter methods. The state of the object should only depend on arguments with an immutable state. If the state of the object was dependent on the mutable part of arguments to its methods, the state of the object could be changed by changing the state of the arguments long after the call, bypassing the objects interface. The approach uses tags to annotate types and enables the compiler to enforce the restrictions mentioned on the creation of aliases. A formalization of this model is discussed by Clarke et al. [5]. Even though both approaches enable flexible alias control, they are designed for a language without inheritance or subtyping.

A variant of ownership types is used by Boyapati et al. [3] to prevent data races and deadlocks by partitioning locks into a fixed number of equivalence classes and specifying a partial order among these equivalence classes. The type checker then statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in descending order. Ownership types are used to ensure that that the locks that protect an object also protect its encapsulated objects.

The approach of Clarke et al. [4] implements a confinement checker for Java to solve the domain specific problem of passing a `this` reference from one Enterprise Java Bean component to another component. In EJB access to the internal objects implementing each Bean must be prevented, and access to the Bean is permitted only through the container generated wrapper. While confined types are a generic solution to control aliasing, Clarke et al.'s approach solves an EJB specific problem.

The work of Fong [11] describes how to translate the notion of confinement, which is formulated for static analysis of Java source code, to dynamic analysis of Java Bytecode. The approach retains the confinement annotations made in the source code at bytecode level. This enables link time checks of confinement rules. It also describes a form of secure cooperation between mutually suspicious code units, where, for example, a resource object can be shared between two untrusting modules while ensuring its confinement to a given domain. The implementation extends the runtime of the Pluggable Verification Modules of the Aegis Research JVM. Our approach uses static analysis to ensure the confinement properties at compile time and to immediately inform the user of confinement violations.

In [20], the notion of confined types is formalized in the context of Featherweight Java (FJ). In FJ, confined types are extended to confined instantiations of generic classes.

Reverse engineering approaches to the detection of aliasing are described in [12,16]. Kacheck/J [12] is a tool to infer confinement in Java code and was used to test the thesis that all package-scoped classes in Java programs should be confined. About 25% of the classes of their benchmark suite were confined anyway and 45% could be refactored to be confined just by changing visibility modifiers. These numbers are supported by the findings of Potanin et al. [16]. They presented metrics of uniqueness, ownership and confinement by analysing snapshots of Java program's object graphs and found that a third of all objects were strongly confined.

7 Summary & Future Work

In future work, we will extend the analysis to implement confined types with support for generic data types. This would relax the restrictions now posed on the use of confined types as it enables putting confined types in containers, which are parameterized using the confined type. Further, we will add support for a more flexible definition of protection domains to broaden the range of use of the confinement analysis; e. g., to check Enterprise Java Beans for correctly confining `this` to the scope of the bean.

In this paper we have discussed an implementation of an incremental confinement

analysis realized as an Eclipse plug-in that makes use of the static analysis platform Magellan. As the performance figures show, the overhead when always executing the analysis along with the incremental build process is low enough to be able to use confined types in day-to-day usage. Further, using Magellan we were able to overcome the identified tool adoption barriers while being able to focus on the implementation of the analysis.

The confined types analysis plug-in is freely available at:
www.st.informatik.tu-darmstadt.de/Magellan

References

- [1] Robert Balzer, Jens Jahnke, Marin Litoiu, Hausi A. Müller, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Ken Wong. 3rd international workshop on adoption-centric software engineering. In *Proceedings of ICSE 2003*.
- [2] Bcel, 2005.
- [3] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of OOPSLA 2002*, number 11, New York, NY, USA, November. ACM Press.
- [4] Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: deployment-time confinement checking. In *Proceedings of OOPSLA 2003*. ACM Press, 2003.
- [5] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of OOPSLA '98*, New York, NY, USA, 1998. ACM Press.
- [6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4), 1991.
- [7] Eclipse 3.1, 2005.
- [8] Michael Eichberg and Christoph Bockisch. Bat, 2005.
- [9] Michael Eichberg and Christoph Bockisch. Magellan, 2005.
- [10] J. Favre, J. Estublier, and R. Sanlaville. Tool adoption issues in a very large software company. In *Proceedings of ICSE 2003*, 2003.
- [11] Philip W. L. Fong. Link-time enforcement of confined types for jvm bytecode. In *To appear in Proceedings of PST'05*, October 2005.
- [12] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of OOPSLA 2001*. ACM Press, 2001.
- [13] John Hogg. Islands: aliasing protection in object-oriented languages. In *Proceedings of OOPSLA '91*, New York, NY, USA, 1991. ACM Press.
- [14] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.
- [15] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Proceedings of ECCOP '98*, London, UK, 1998. Springer-Verlag.
- [16] Alex Potanin, James Noble, and Robert Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7), April 2004.
- [17] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000.
- [18] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundareshan. Optimizing java bytecode using the soot framework: Is it feasible? In *Proceedings of CC 2000*. Springer, 2000.
- [19] Jan Vitek and Boris Bokowski. Confined types in java. *Software Practice and Experience*, 31(6), 2001.
- [20] Tian Zhao, Jens Palsberg, and Jan Vitek. Lightweight confinement for featherweight java. *SIGPLAN Not.*, 38(11), 2003.