Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java

Wojciech J. Dzidek[2], Lionel C. Briand[1,2], Yvan Labiche[1]

[1] Software Quality Engineering Laboratory, Department of Systems and Computer Engineering – Carleton University, 1125 Colonel By Drive, Ottawa, ON, K1S 5B6, Canada
{briand, labiche}@sce.carleton.ca
[2] Simula Research Laboratory
Lysaker, Norway
dzidek@simula.no

Abstract. Analysis and design by contract allows the definition of a formal agreement between a class and its clients, expressing each party's rights and obligations. Contracts written in the Object Constraint Language (OCL) are known to be a useful technique to specify the precondition and postcondition of operations and class invariants in a UML context, making the definition of object-oriented analysis or design elements more precise while also helping in testing and debugging. In this article, we report on the experiences with the development of ocl2j, a tool that automatically instruments OCL constraints in Java programs using aspect-oriented programming (AOP). The approach strives for automatic and efficient generation of contract code, and a non-intrusive instrumentation technique. A summary of our approach is given along with the results of an initial case study, the discussion of encountered problems, and the necessary future work to resolve the encountered issues.

1 Introduction

The usefulness of analysis and design by contract (ADBC) has been recognized by current and emerging software paradigms. For example, in [1], a book on component software, an entire chapter is devoted to the subject of contracts, and the author argues that using a formal language to specify them would be ideal except for the disadvantage of the complexity associated with the usage of a formal language. However, recent experiments have shown that OCL provides a number of advantages in the context of UML modeling [2], thus suggesting its complexity to be manageable by software engineers. Likewise in [3], a book discussing distributed object-oriented technologies, Emmerich argues that the notion of contracts is paramount in distributed systems as client and server are often developed autonomously. Last, model driven architecture (MDA), also known as model driven development (MDD), is perceived by many as a promising approach to software development [4]. In [4], the authors note that the combination of UML with OCL is at the moment probably the best way to develop high-quality and high-level models, as this results in precise, unambiguous, and consistent models. Having discussed the advantages of OCL, it

comes as a surprise that the language is not used more widely for ADBC. One reason for this might be the well-established prejudices against any formal elements among software development experts and many influential methodologists. Another reason for the unsatisfactory utilization of OCL is the lack of industrial strength tools, e.g., tools to generate code assertions from OCL contracts.

The benefits of using contract assertions in source code is shown in [5], where a rigorous empirical study showed that such assertions detected a large percentage of failures and thus can be considered acceptable substitutes to hard-coded oracles in test drivers. This study also showed that contract assertions can be used to significantly lower the effort of locating faults after the detection of a failure, and that the contracts need not be perfect to be highly effective. Based on such results, the next step was therefore to address the automation of using OCL contracts to instrument Java systems. This paper reports on our experience with the development and use of *ocl2j*, a tool for the automated verification of OCL contracts in Java systems [6]. These verifications are dynamic, i.e., are performed during the execution of the application.

The paper briefly starts with background information, motivation, and related work. Then we go through an overview of our approach, followed by a discussion of some of the main technical and methodological issues with respect to transformation of constraints from OCL to Java. Next, an initial case study, aimed at showing the feasibility of the ocl2j approach, is presented. Finally, difficulties with using OCL for this purpose are outlined, conclusions are then provided.

2 Motivation and Related Work

Currently, two tools exist for the purpose of dynamic enforcement of OCL constraints in Java systems: the Dresden OCL toolkit (DOT) [7, 8] and the Object Constraint Language Environment (OCLE) [9]. We decided to implement our own solution as DOT did not fulfill all of our requirements and OCLE did not exist at the time, though it doesn't fully address our needs either.

Our aim was to have a tool that would: (1) support all the core OCL 1.4 functionality, (2) correctly enforce constraints, (3) instrument (insert the contract checking and enforcement code) program code at the bytecode level (as opposed to altering the source-code), (4) allow for optional dynamic enforcement to the Liskov Substitution Principle (LSP) [10], (5) support for separate compilation (i.e., allowing modifications of the application source code without recompiling assertion code or vice-versa), (6) correctly check constraints when exceptions are thrown, (7) have the ability for assertion code to use private members, (8) have the option to use either compile-time or load-time instrumentation (with load-time instrumentation constraint checking code can be installed or removed without requiring recompilation), and (9) have the ability to add assertions to classes for which the source-code is not available.

DOT was the pioneering work for this problem and is open-source software. It relies on the following technical choices. First, the instrumentation occurs at the source code level: original methods are renamed and wrapped, and supplementary code is added. OCL types are implemented in Java and Java variables (attributes, method parameters or return value) used in assertions are wrapped with equivalent

OCL types. Last, the generated code is constructed in such a way that it uses Java reflection mechanisms at runtime to determine implementation details. Additional logic is inserted that tries to minimize the checking of invariants. Those technical decisions result in a large memory and performance penalty as a direct consequence of the virtual cloning (of all objects) and the wrapping (of all objects involved in OCL constraints). Support for OCL is also incomplete as, for example, query operations are not supported. Furthermore, constraints on elements in collections are not properly enforced as changes to elements in the collection can go unnoticed [6]. (Source-code level instrumentation suffers from two main disadvantages: it makes the developer deal with two versions of the source-code and it makes it much harder to debug the application, e.g., when single stepping through the source-code.)

OCLE is a UML CASE tool offering OCL support both at the UML metamodel and model level, though we only look at the latter: i.e., support for dynamic OCL constraint enforcement. Like DOT, OCLE instruments the source code and is limited in its support of OCL (e.g. the `@pre` keyword is not supported). Furthermore, it cannot instrument existing (reverse-engineered) source code.

Note that although other tools exist that add design by contract support to Java [11, 12], they are not discussed in this paper as they do not address the transformation of OCL expressions into assertions.

3 The ocl2j Approach

This section presents our approach (ocl2j) towards the automatic generation and instrumentation of OCL constraints in Java. Our approach consists of Java code being created from OCL expressions and the target system then being instrumented: (1) The necessary information is retrieved from the target system's UML model and source code; (2) Every OCL expression is parsed, an abstract syntax tree (AST) is generated [7], and the AST is used to create the assertion code (the OCL to Java transformation rules were defined as semantic actions associated with production rules of the OCL grammar [13]. The generation of Java assertions from OCL constraints is thus rigorously defined and easily automated.); (3) The target system is then instrumented with the assertion code, using AspectJ which is the main Java implementation of Aspect Oriented Programming (AOP) [14]. The techniques involved in step (3) are omitted as they're already described in [15]. It is important to emphasize that this strategy played a large role in helping us achieve the goals outlines in Section 2.

The section starts (Section 3.1) with a discussion of how OCL types are transformed in Java types. Next, Section 3.2 discusses the topic of equality with respect to OCL and Java. Section 3.3 shows how the OCL `@pre` construct is addressed. Finally, Section 3.4 shows how we were able to use AspectJ to provide clean and efficient support for `oclAny::oclIsNew()`.

3.1 OCL to Java Transformations

The checking of contracts at runtime slows down the execution of the program. If this slowdown is too great the developers will not use the technology. For this reason it is important to focus on techniques that enable faster checking of contracts. One of these techniques is to translate OCL expressions directly into Java using the types retrieved from the target system (through reflection) at the assertion-code generation stage, instead of wrapping Java types and operations with OCL-like types and operations [7, 8]. The translation time is thus spent during instrumentation rather than execution. This distinction becomes critical during maintenance of large systems since changes to the system only occur to the subsystem under development. For this reason it is both unnecessary and inefficient to perform the OCL to Java type resolution over the whole system every time the system is executed.

Our OCL to Java type resolution relies on the following principles. First, whenever a simple mapping exists between OCL and Java types/operations, the translation is straightforward. For instance, OCL collection operation `size()` maps directly to the `size()` operation of the `java.util.Collection` interface (which every collection class in Java implements). When OCL types/operations cannot be directly converted to types/operations from standard Java libraries, the instrumentation code (aspect code) provides the functionality that is "missing" in the libraries. This ensures that no wrapping is necessary, and no additions to the target system are required. The instrumentation code (i.e., the aspect) contains inner classes with operations that provide additional functionality to complete the mapping to Java such as the `collection->count(obj):Integer` operation, that counts the number of times object `obj` occurs in `collection` and does not have any counterpart in Java collection classes/interfaces. The aspect code thus contains inner class `OclCollection` with a `count()` static operation that takes two arguments: the collection on which to count, and the object that needs to be counted.

Next, OCL, unlike Java, has no notion of primitive types (e.g., `int`) as everything is considered an object. Java, on the other hand, supports primitive types and corresponding primitive value wrapper classes, or simply wrapper classes (e.g., `Integer`). OCL provides four, so-called, basic types: `Boolean`, `Integer`, `Real` and `String`. There is one exception to these differences in OCL and Java type systems: strings are objects in both OCL and Java. Having both primitive types and wrapper classes has a major impact on the process of OCL to Java transformation (unless the system is written in Java 1.5 where the autoboxing feature is available). For example, consider the following OCL constraint: `someCollection->includes(5)`. When transforming the OCL expression into Java source code, `5` has to be transformed into either primitive value `5` or an instance of wrapper class Integer (`new Integer(5)`). As Java collections only take objects as elements, the latter is the correct choice. A general, trivial solution to this problem would be to convert every literal value into an object, but as already discussed, this is inefficient. A more efficient solution consists in analyzing the types used in the OCL expression, the types required in the corresponding Java source code, as well as the characteristics of the expression, and converting objects to their primitive types when possible (i.e. values used in logical, addition, multiplication, and unary operations).

OCL has three collection types, namely `Set`, `Bag`, and `Sequence`, whereas, Java only has two main collection interfaces, namely `java.util.Set` and `java.util.List` (we assume that user-define collections directly or indirectly implement `java.util.Collection`). There is a direct mapping between OCL `Set` and `java.util.Set` and between OCL `Sequence` and `java.util.List`. However, OCL `Bag` does not have a direct Java counterpart. A bag is a collection in which duplicates are allowed [16]. `java.util.Set` cannot be used to implement an OCL `Bag` as it does not allow duplicates. The only possible alternative, which is assumed in the ocl2j approach, is to implement OCL `Bag` with `java.util.List`.

The following 3 scenarios are encountered when translating a collection operation:

1. There is a direct mapping between the OCL operation and a Java operation.
2. The OCL operation does not have a direct counterpart but its functionality can easily be derived from existing Java operations.
3. OCL operations that iterate over collections and evaluate an OCL expression (passed as a parameter to the operation) on elements in the collection are more complex. They do not have a direct Java counterpart and cannot be simply implemented using the operations provided by `java.util.Set` or `java.util.List`. These OCL operations are `exists`, `forAll`, `isUnique`, `sortedBy`, `select`, `reject`, `collect`, and `iterate`. They require more attention as the parameter is an OCL expression which requires to be instrumented as well in the aspect code. Templates and transformation rules are used to generate a unique method (residing in the aspect) for every distinct use of these operations [6].

3.2 Testing for Equality

Assertion code that tests for equality can take any one of three forms. First, if the values to be compared are of primitive type then the Java "`==`" construct is used in the equality test. Next, if the values being compared (or just one of them) are of reference type wrapping a primitive then the primitive value is extracted from the object using the appropriate method (e.g., `intValue()` for an object of type `Integer`) and again the values are tested for equality using the Java "`==`" construct. In other cases, objects are tested for equality using their `equals(o:Object):boolean` method. This is done as equality in OCL is defined at the object level, not the reference level. For example, let's take a look at the `java.awt.Point` class which has two attributes: `x:int` and `y:int`. Given two points `Point a = new Point(5, 5)` and `Point b = new Point(5, 5)`. If we compare these points at a "reference level" they will not be equal (`a == b` evaluates to `false`), even though they the two objects `a` and `b` do represent the same point. Thus, `Point`'s `equals` method must be used to evaluate their equality (`a.equals(b)` evaluates to `true`).

We assume that the `equals()` method is properly implemented [17] so that objects are deemed equal when their key attributes are equal. We define *key attributes* as attributes that define an object's identity (e.g., attributes `x` and `y` in the case of the `Point` class). Sometimes each instance of a class is unique (no clones are possible) in which case the default `equals()` functionality (i.e., inherited from

`java.lang.Object`, considers each instance only equal to itself) will suffice as this functionality only compares reference values for equality, but when this is not the case the `equals()` method must be overridden. Note that this last point is often neglected by developers of Java-based systems [17].

3.3 Using Previous Property Values in OCL Postconditions

This section discusses the practical implementation of the OCL language construct `@pre`, used in postconditions to access the value of an object property at the start of the execution of the operation. Depending on the property that the `@pre` is associated with different values and amount of data must be stored temporarily until the constrained method finishes executing so that the postcondition can be checked. `@pre` can be used with respect to one of the following:

1. *Java types corresponding to OCL Basic types or query methods that return a value of such a type*. The mapping between these types is discussed in Section 3.1. In the case of a primitive type, the primitive value is stored in a temporary variable. In the case of an object, the reference to the object is stored in a temporary variable. Only the reference is stored as these types are immutable and thus they cannot change (during the execution of the constrained method).
2. *Query methods that return an object*. In this case the objects are handled in the same way as described above, only the reference to that object is stored in a temporary variable (duplicated), the object itself is not cloned. The object is not cloned as we assume that the target system is written with proper encapsulation techniques, meaning that query methods that return an object to which the context class (the class containing the query method) is related via composite aggregation return a clone of the object, not the object itself. This is standard practice as discussed in Item 24 of [17]. Note that this is a necessary requirement as the following example will demonstrate: Consider a query method returning a reference to an object `X`, used in a method `M`'s postcondition with the `@pre` keyword (i.e., we are interested in the value of `X` at precondition-time): i.e., the postcondition reads `…=…query()@pre`. Further assume that `M` modifies `X` during its execution. Once `M` finishes execution the postcondition is verified. Since the query method returns a reference to `X` (instead of a clone of `X`), the postcondition will use the new version of `X`, as opposed to the original version at precondition-time.
3. *Objects (references to objects)*. The object types in this discussion exclude the ones discussed in the points above. In this case a clone of the object is taken and stored in a temporary variable. We assume that the programmer properly implements cloneability support (as will be discussed).
4. *Collections*. A collection's identity is defined by the elements in that collection, thus a clone of a collection contains a clone of every element in the original collection. Using `@pre` on a collection will result in such a duplication of the collection in most cases. When the OCL collection operation being invoked on `someCollection@pre` is `size():Integer`, `isEmpty():Boolean`, `notEmpty():Boolean`, or `sum():T` then only the result of the operation is stored in the temporary variable. We note that in a lot of cases it may not be necessary to

duplicate the collection in such a manner to enforce the postcondition correctly, but this is a subject for future work.

For a guide to providing support for cloneability see Item 10 in [17]. Essentially, two types of cloning methods exist. In a shallow copy, the fields declared in a class and its parents (if any) will have values identical to those of the object being cloned. In the case of a class exhibiting one or more composite relationships the shallow copy is not sufficient and a deep copy must be used. In a deep copy, all the objects in the composition hierarchy must also be cloned. To understand why, recall our objective: We need access to the objects, as they were, before the constrained method executed. Objects are uniquely identified by their key attributes (key attributes are discussed in Section 3.2). If these objects have composite links to other objects (i.e., their class has composite relationships), thus forming a hierarchy of objects, the key attributes may be located anywhere in the hierarchy. A deep copy is therefore necessary.

3.4 `oclAny::oclIsNew()` Support

Any OCL type in a UML model, including user-defined classes, is an instance of `OclType`: it allows access to meta-level information regarding the UML model. In addition, every type in OCL is a child class of `OclAny`, i.e., all model types inherit the properties of `OclAny`. Among those properties is operation `oclAny::oclIsNew()` that can only be used in a postcondition: It evaluates to `true` if the object on which it is called has been created during the execution of the constrained method.

The ocl2j solution to the problem of implementing operation `oclAny::oclIsNew()` is the following. If this operation is used on a type in an OCL expression, a collection is added to an AspectJ aspect. This collection will store references to all the instances of the type created during the execution of the constrained method (as `oclAny::oclIsNew()` can only be used in the context of a postcondition): This is easily achieved with AspectJ as it only requires that the aspect comprises an advice to add, at the end of the execution of any constructor of the type of interest or its subtypes, the reference of the newly created instance. This raises the question of the choice of the Java data structure to store those references and the impact of aspect code on object garbage collection in Java: Objects in the instrumented program should be garbage collected if they are not used in the application code, even though they may be referenced by the aspect code. A solution to this problem is to use class `java.util.WeakHashMap` to store these references in the aspect. This collection was specifically designed so as to store references that would not be accounted by the garbage collector. It is based on a hash map where the keys are weak references to the objects we are monitoring. The garbage collector can get rid of an object, even when this object is still referenced, provided that these references are only used in instances of class `WeakHashMap`. When this is the case, the object is garbage collected and any reference to it removed from instances of the `WeakHashMap`.

Determining whether an object was created during the execution of the constrained method involves checking the `WeakHashMap` collection for the presence of the object in question. Finally, after the constrained method finishes executing and the

postcondition is checked, the collection of instances (created during the execution of that method) is discarded.

Please note that this solution is not easily mapped to a solution that enables the use of the `oclAny::allInstances()` construct as there is no way to *force* the JVM to run the garbage collection operation (though `Runtime.getRuntime().gc()` can be used to *suggest* this to the JVM). Thus, such an implementation of `oclAny::allInstances()` could, in certain instances, return a collection of objects including ones that are designated for garbage collection (no longer referenced).

4 Preliminary Case Study

The case study is based on the system presented in [16]: The "Royal and Loyal" system example. Though modest in size, this system was chosen due to the large number of diverse constraints being already defined for it, including some quite complex ones. It should then provide initial evidence that ocl2j works for a wide variety of constraints. The UML model in [16] was expanded in this work to the system shown in [6] in order to be implementable. Once expanded, it was implemented in Java and consisted of 381 LOCs, including 14 classes, 47 OCL constraints, 53 attributes, and 46 operations.

The original version of the R&L system and the version with the assertion code (instrumented) are compared for a set of scenarios where various numbers of customers are added and various amounts of purchases are made. We use the following three criteria for comparison: (1) bytecode size of the classes, (2) time it takes to execute the program (in various scenarios), and (3) memory footprint (again, in various scenarios). From the case study we conclude that programs that have relatively large collections with many complicated constraints associated with these collections can expect, as a ballpark figure, a degradation in execution time of 2 to 3 times. Otherwise, the degradation in performance is smaller as the execution speed is slowed down by roughly 60%. This is significant but does not prevent the use of instrumented contracts in most cases during testing, unless the system's behavior is extremely sensitive to execution deadlines. The sources of degradation in performance have been further investigated in [6] where solutions are proposed for optimization. With respect to criteria (1), the target system grew 2.5 times in size, and (3), the maximum overhead percentage observed for the above scenarios were 14% and 10.5%, respectively.

5 Future Challenges

While developing ocl2j we ran into several non-trivial issues that require significant work to address. Among others:

- Providing support for the `@pre` keyword leaves a lot of room for performance optimizations. For example, to properly evaluate the postcondition `self.aCollection@pre = self.aCollection` in every scenario, one must

create a new collection (say `tempCollection`) that holds a clone of every element present in `self.aCollection`. If `aCollection` is large or if the elements in that collection are expensive to clone, then the evaluation of this postcondition becomes very expensive. Furthermore, this potentially expensive operation is not even necessary if all the designer intended to check was whether `self.aCollection@pre` and `self.aCollection` point to the same object (i.e. hold the same reference). In such a situation the designer should be allowed to distinguish weather a deep or shallow copy is meant by the `@pre`. One way of addressing this would be by adding the keyword `@preShallow` to OCL.

- The use of `@pre` may lead to un-computable expressions. As shown in [18], the expression `self.b.c@pre` with respect to the example in Section 7.5.15 in [20] is not computable: "Before invocation of the method, it is not yet known what the future value of the `b` property will be, and therefore it is not possible to store the value of `self.b.c@pre` for later use in the postcondition!".
- Our experience revealed that, by far, the largest performance penalties (execution time overhead) of checking the OCL constraints during the execution of the system came from OCL collection operations [6]. For this reason we have started working on an approach to minimize these performance penalties. In general the strategy involves checking a constraint on a collection whenever the state of the collection changes in such a way that it could invalidate the constraint. For example, consider the constraint `aCollection->forAll(anExpression)`. If this constraint is an invariant then it will be checked before and after any public method executes, even if neither the state of `aCollection` nor its elements changes. An alternative to this would be to check that `anExpression` holds for a newly added element to `aCollection`, and that `anExpression` holds for elements in the collection that undergo changes that may invalidate it. This alterative will be more efficient on a large, often-checked, collection that does not undergo large changes. Note that this kind of strategy is facilitated by the use of AOP as the instrumentation technology.
- The implementation of the `OclAny::allInstances():Set(T)` functionality in Java is challenging since Java uses automatic garbage collection, i.e., objects do not have to be explicitly destroyed. Thus, the only way to know whether an object is ready to be garbage collected (and therefore not be in the `allInstances` set) is to run the garbage collection operation (costly execution-wise) after every state change in the system involving the destruction of a reference.

6 Conclusions

We have presented a methodology, supported by a prototype tool (ocl2j), to automatically transform OCL constraints into Java assertions. The user of ocl2j can then specify whether a runtime exception is thrown or an error message is printed to the standard error output upon the falsification of an assertion during execution. This has shown, in past studies [5], to be extremely valuable during testing to detect failures and help debugging.

Transformation rules to translate OCL constraints into Java assertions have been derived in a systematic manner with the goal that upon instrumentation the generated

assertion code will be efficient in terms of execution time and memory overhead. This was largely achieved thanks to the systematic definition of efficient semantic actions on production rules in the OCL grammar, and the minimization of reflection use at runtime. An initial case study has shown that the overhead due to instrumentation compares very well to previous approaches [8] and is likely to be acceptable in most situations, at least as far as testing is concerned. More empirical studies are however required. Furthermore, we have shown how we dealt with aspects of the OCL specification that present serious instrumentation challenges (e.g. providing support for `@pre` and `oclIsNew()`) and reported on issues that we feel require future work (e.g. refinement of the OCL syntax and advanced optimization techniques).

References

1. Szyperski, C., *Component Software*. 2nd ed. 2002: ACM Press.
2. Briand, L.C., et al. *A Controlled Experiment on the Impact of the Object Constraint Language in UML-Based Development.* In *IEEE ICSM* 2004. p. 380-389.
3. Emmerich, W., *Engineering Distributed Objects*. 2000: Wiley.
4. Kleppe, A., J. Warmer, and W. Bast, *MDA Explained - The Model Driven Architecture: Practice and Promise*. 2003: Addison-Wesley.
5. Briand, L.C., Y. Labiche, and H. Sun, *Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code*. Software - Practice and Experience, 2003. **33**(7): p. 637-672.
6. Briand, L.C., W. Dzidek, and Y. Labiche, *Using Aspect-Oriented Programming to Instrument OCL Contracts in Java*. 2004. SCE-04-03. http://www.sce.carleton.ca/squall.
7. Finger, F., *Design and Implementation of a Modular OCL Compiler*. 2000, Dresden University of Technology.
8. Wiebicke, R., *Utility Support for Checking OCL Business Rules in Java Programs*. 2000, Dresden University of Technology.
9. LCI, *Object Constraint Language Environment (OCLE)*. http://lci.cs.ubbcluj.ro/ocle/.
10. Liskov, B., *Data Abstraction and Hierarchy.* SIGPLAN Notices, 1988. **23**(5).
11. Plösch, R., *Evaluation of Assertion Support for the Java Programming Language.* Journal Of Object Technology, 2002. **1**(3).
12. Lackner, M., A. Krall, and F. Puntigam, *Supporting Design by Contract in Java.* Journal Of Object Technology, 2002. **1**(3).
13. Appel, A.W., *Modern Compiler Implementation in Java*. 2nd ed. 2002: Cambridge University Press.
14. Elrad, T., R.E. Filman, and A. Bader, *Aspect-oriented programming: Introduction.* Communications of the ACM, 2001. **44**(10): p. 29-32.
15. Briand, L.C., W.J. Dzidek, and Y. Labiche. *Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging.* In *IEEE International Conference on Software Maintenance.* 2005.
16. Warmer, J. and A. Kleppe, *The Object Constraint Language*. 1999: Addison-Wesley.
17. Bloch, J., *Effective Java: Programming Language Guide*. 2001: Addison Wesley.
18. Hussmann, H., F. Finger, and R. Wiebicke. *Using Previous Property Values in OCL Postconditions - An Implementation Perspective.* in *<<UML>>2000 Workshop - "UML 2.0 - The Future of the UML Constraint Language OCL"*. 2000.