Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 116 (2005) 73–84
www.elsevier.com/locate/entcs

Toward Translating Design Constraints to Run-Time Assertions

Luciano Baresi[1]

Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy

Michal Young[2]

Department of Computer Science
University of Oregon
Eugene, USA

Abstract

Inconsistency between design descriptions and implementation might be reduced if constraints associated with entities (e.g., OCL assertions in UML) were propagated to run-time assertions in corresponding parts of an implementation.
We describe an approach in which constraints in a fixed design assertion language are propagated using translation rules that can be selected or customized for different implementation programming languages or assertion packages. Translation rules are kept simple by exploiting existing assertion packages where possible.
We have implemented key parts of a prototype tool for translating OCL assertions to implementation assertions. We illustrate the approach by applying the prototype tool to an example, contrast it with other current proposals which rely more on run-time interpretation, and discuss some issues in design assertion propagation.

Keywords: Design assertions, UML, OCL, transformation rules.

[1] Email: baresi@elet.polimi.it
[2] Email: michal@cs.uoregon.edu

doi:10.1016/j.entcs.2004.02.085

1 Introduction

Inconsistency between design descriptions and implementation might be easier to detect and prevent if constraints associated with design entities were propagated to run-time assertions in corresponding parts of an implementation.

Tools for propagating design constraints to run-time assertions should permit a uniform style of design assertions with minimum commitment to implementation details. In particular, one would prefer design assertions to be independent of the implementation programming language.

The approach described here maps a fixed design assertion language to different implementation assertion languages using translation rules that can be selected or customized for different implementation programming languages or assertion packages. Currently we are focusing on the Unified Modeling Language (UML) as a design notation, and its Object Constraint Language (OCL) as the design assertion language [13].

This translation approach may be contrasted with approaches that essentially embed an assertion language interpreter in the run-time environment. A translational approach is in some ways less powerful than the interpreter approach, because it can only support design assertion constructs that have fairly straightforward translations into an implementation assertion language. The benefit of simplicity is flexibility: It is relatively easy to modify translation rules, possibly creating multiple variants, and creating a new binding to a different implementation language or a different assertion language is simpler than producing a new interpreter. The translation approach should be particularly appropriate for systems implemented in a mix of languages, like component-based systems.

The remainder of the paper is organized as follows: Section 2 illustrates the main ideas of the approach and some issues through application to a hypothetical system. Section 3 describes an initial prototype implementation which includes a customizable translator for OCL. Section 4 discusses the relation of our current work to related research in assertion systems, and particularly alternative approaches to supporting OCL. Section 5 concludes the paper.

2 Our Approach

We illustrate an approach to propagating design assertions to implementation source code with a hypothetical bus transit information system. The data model underlying the bus transit system must support several different functions, from printing route schedules and answering web queries to allocating

routes to buses and drivers. We have designed a data model and some functions of the system using UML. The class diagram – with basic elements only – is presented in Figure 1.

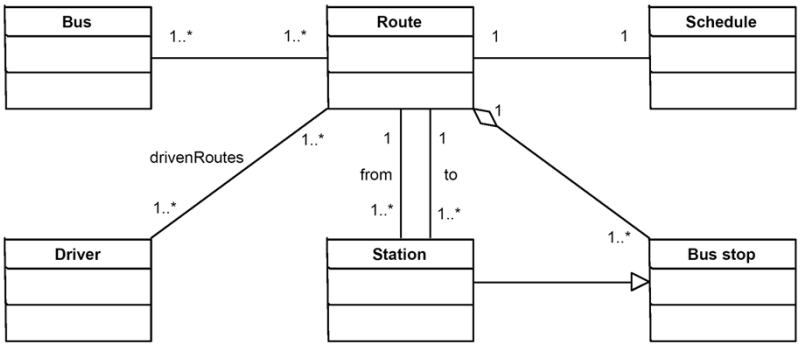

Fig. 1. UML class diagram of the example system

A single *Route* (e.g., #7 Bailey Hill) is associated with one *Schedule*, that is, a named collection of days (e.g., Weekdays) along with a scheduling interval (e.g., every 15 minutes from 8am to 12am). Each *Schedule* is associated with a single *Route* (e.g., a shorter version of the route may be run after 6pm). A route is composed of a sequence of stops, each of which is a *Bus stop*, and the end-points of which are time points. A time point is a *Station* at which a bus, if it arrives early, will wait until its scheduled departure time. All potential transfer points are time points and end-points of route segments.

One of the functions of the bus system would be producing or maintaining assignments of *Buses* and *Drivers* to routes. A bus driver can drive multiple routes concurrently, provided those routes have a common starting point[3]. This consistency condition might be stated as an OCL constraint on class `Driver` of the model:

```
context Driver
  inv: drivenRoutes->size > 1 implies
  drivenRoutes->forAll(r: Route |
    r.from = drivenRoutes->at(1).from)
```

If this part of the system were implemented in Java, we might want to bind the object model invariant with the corresponding Java class `Driver` If source code is generated automatically from a relatively implementation-oriented UML model, then the association of design entity with source code could likewise be automatic. If, on the other hand, assertions are associated

[3] One can identify more complex conditions (e.g., the running times of the routes should interleave without overlap), but for presentation purposes we limit ourselves to this simple property.

with a logical data model, and we do not require that implementation language classes be named identically to entities in the UML model, we might require an explicit binding, for example:

```
/**
 * @title BusDriver
 * @represents Driver
 */
class BusDriver {  ... }
```

Here a comment provides the link to the UML model and triggers a translator that would insert assertions in (for example) the iContract assertion language [3]. We use this assertion system to exemplify the approach, but several other Java assertions languages would have helped in the same way. A slightly different solution would have been the use of `assert` statements of the Java programming language. The tradeoff is between flexibility and more special-purpose constructs. The direct use of the features offered by the language allows for flexibility, but lets the user free to state the assertions he/she prefers. A special-purpose assertion system better constraints what the user can do.

It would be straightforward also to log which UML entities were accounted for in `@represents` clauses, noting any mismatches and/or unimplemented entities, as well as any OCL assertions that could not be translated. Since iContract provides most of the functionality of OCL assertions (preconditions, postconditions, invariants), then the translation is straightforward.

A *Trip* (not shown in Figure 1) is a plan for traveling from one bus stop to another, starting at a particular time, and transferring from one bus to another zero or more times. If a user of the bus system asks how to reach Far Foodle from Dofft starting around 9am, the web query might return the following schedule:

Bus	On at		Off at	
#17 Bailey Hill	Dofft	9:05a	Va-Vode	9:20a
#23 University	Va-Vode	9:30a	Mercedd	9:35a
#02 Keck Loop	Mercedd	9:45a	Far Foodle	10:00a

Several properties of a trip can be specified at the design level using OCL.

The times in each column should be in an ascending order, one should get off and on at the same bus stop when transferring buses, and there should be at least 5 minutes between leaving one bus and entering the next[4]. These constraints might be expressed in the following OCL postcondition:

```
context findRoute()
```

[4] The obvious constraint that Off at must always be greater than On at is not the same as what stated here. The fact that the times in each row must increase moving left to right is quite obvious. Less obvious is that each leg of the trip obeys the constraints defined above.

```
post: Sequence{1..n}->forAll(i: Integer |
(trip->at(i-1).onTime <= trip->at(i).onTime) and
(trip->at(i-1).offTime <= trip->at(i).offTime) and
(trip->at(i-1).offStop = trip->at(i).onStop) and
((trip->at(i).onTime - trip->at(i-1).offTime) >= 5))
```

Web-based queries might be implemented in C using specialized tables that are pre-computed off-line. We might wish to bind them to assertions in the Nana assertion language [5]. Unlike iContract, though, Nana does not directly support preconditions, postconditions, and invariants; the binding to implementation entities would have to indicate the points at which these assertions should be evaluated. For example:

```
/**
 * @represents findRoute n=nsegments
 */
int whatroute( char *start, *dest,
           struct timeval starttime,
           struct segment[] trip)
{  ...
  /*@post*/
  return nsegments;
}
```

where `nsegments` is the number of segments that belong to the `trip`. In general it might not be enough to insert the translation of a postcondition at one point in the implementation. Since Nana does not automatically store *before* values of variables mentioned in postconditions, it could also be necessary for the assertion propagation tool to insert explicit instructions for capturing the mentioned *before* values at a point indicated by the developer.

2.1 Issues in Translation

Translating design assertions to an implementation language can present several issues, some of which we discuss here.

Language mapping A design assertion language is likely to support constructs that are not found in the target language. In some cases these can be relatively simple but tedious to cope with, such as translating the OCL exclusive OR operation into Java. Not all language issues may be so easy to deal with. Fundamental mismatches between the type systems of the assertion language and the implementation language may require more complex solutions — but this is as true for interpretive systems as for translation.

Name mapping While names of implementation entities are sometimes identical to the names of corresponding design model entities, or can be derived by some systematic transformation, this is not always the case. In the most general case, one might need to specify abstraction functions (in the usual sense of abstract data type implementations) to relate *sets* of implementation variables to design entities. In such cases, a facility like Anna's "virtual

text" becomes invaluable [4]. In most cases, though, simple rules for substituting names (like `n=nsgements` in the example above) should suffice.

Quantifiers Universal and existential quantifiers are used heavily in specifying program functionality. Increasingly, quantification over collection data structures is also supported in implementation assertion languages, but it is potentially very expensive. For example, one can write an assertion that the output of a sorting procedure should be a permutation of its input, but the cost of interpreting the nested quantifiers in the definition of "is a permutation of" would be at least quadratic in the length of the array being sorted. One would not want to restrict design assertions to those that can be interpreted efficiently. It appears that the only reasonable choices are to suppress some of the assertions (either in translation, or using facilities of the target assertion language), or to allow developers to explicitly substitute implementation-level assertions where translated assertions would be unacceptable.

2.2 Translation versus Interpretation

The most salient characteristic of the translational approach to supporting design-language assertions is its simplicity. The simplicity is obtained primarily through dependence on pre-existing assertion systems for particular implementation programming languages. When the assertion system supports most OCL features directly, the translation is straightforward, and the burden of linking implementation entities to design entities is minimized. Interpretive systems for design assertions, in contrast, must re-implement facilities like "before" value caching even when they are already supported in an existing assertion system.

The primary consequence of simplicity is flexibility: It is easy to modify or write translation rules. One can easily switch assertion systems (e.g., from Nana to APP [12]) or, with a little more effort, switch implementation programming languages. The burden placed on the programmer is proportional to the semantic gap between the design constraint language and the target assertion language.

As in programming language processors, there is no bright line separating translation and interpretation. Interpreters usually involve at least some translation to an intermediate form, and translators usually depend at least partly on provision of run-time libraries. One can easily imagine solutions in which design constraints are partly translated, with library support filling the gaps between the design constraint language and the functionality provided by existing assertion languages.

Without significant additional run-time support, it is inevitable that translation will at least partly fail for some design constraints, unless one significantly limits the use of the design constraint language. Our approach is to accept the inherent partiality of the approach, not insisting that every design assertion is fully translated. Which design assertions have been represented in the implementation, and to what degree, is treated merely as useful information for the developer. One may even choose not to translate some constructs for reasons of efficiency. For example, one might have one set of rules that faithfully translates bound quantifiers into iteration, and another set of translation for the same target assertion language that omits translation of quantifiers.

3 Prototype Implementation

We have implemented a prototype tool to transform OCL constraints to run-time assertions, as part of an overall effort to refine and evaluate the approach described above. In the future we expect the translation tool will be a component of a larger environment for coordinating design artifacts with run-time testing.

The assertion translation tool includes a parser for OCL and a transformer. It takes an OCL assertion, builds the corresponding abstract syntax tree (AST), decorates it with relevant information from the UML model, and produces the equivalent run-time assertion by dumping the nodes of the AST according to externally defined rules.

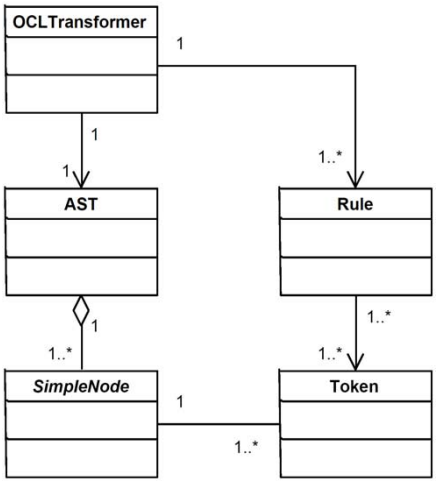

Fig. 2. Main classes of the OCL transformer

The main classes of the translation tool are presented in Figure2. Class *OCLTransformer* is the entry point and provides the right interfaces. It allows the tool to create the *AST* (Abstract Syntax Tree) and parse the *Rules*. The AST comprises instances of the subclasses of *Simple Node*. This class is abstract to factor out all the properties that are common to all actual nodes. There is a special-purpose node type for each relevant syntactical element (for example, we have nodes for expressions, literals, and many others). Finally, class *Token* links nodes and *Rules*: Each node comprises some tokens that are then interpreted according to the chosen rule.

The AST builder is based on a slightly modified version of the OCL grammar that is distributed with UML version 1.3 [8]. The original grammar did not distinguish between *featureCalls*, i.e., model-dependent features, and OCL-specific operations over collections. To make the translation process easier, we distinguish between *featureCalls* and *collectionOperationCalls*: OCL makes this difference evident because it uses a dot (`.`) to refer to "proper" feature calls and an arrow (`->`) to refer to calls to operations on collections. Besides this, we listed all operations explicitly, so that each operation can be translated in a particular way. We did not distinguish among the different collection types (i.e., sets, bags, sequences, and collections), but we exploited overloading to make operations be specific to the proper collection type.

The modified grammar was used with `jjtree` and `javacc`[5] to construct the parser and the AST builder. Each grammar element defines a class of AST nodes, which inherits from a general-purpose class `simpleNode`. The super class provides a trivial – and often useless – method for dumping generic nodes, while subclasses can redefine the method to implement special purpose translations. The rule-based approach made this redefinition simple and straightforward: The translation is not hard-coded in any method, but it is accomplished by applying externally defined rules, which can vary according to the target notation and users' needs.

Rules are identified by the name of the element they refer to and allow users to define any transformation obtained by concatenating fixed strings with the result of translating (some of) the child nodes of the element they are applied on. Rules range from simple lexical transformations to more complex syntactical transformations. For example, a logical AND is rendered into C or Java syntax using the simple rule:

```
AND ::= par1 ' && ' par2;
```

The mix of fixed strings and to-be-translated nodes allows us both to consider translating such operations using pre(post)fix notations and to obtain

[5] See https://javacc.dev.java.net/

complex operations by composing primitive ones. For example, the OCL exclusive OR can be "simulated" using AND, OR, and NOT operators by applying the rule:

```
XOR ::= '(' '(' par1 ')' ' && '
        '!' '(' par2 ')' ')' ' || '
        '(' '!' '(' par1 ')' ' && '
        '(' par2 ')' ')';
```

More complex rules are necessary to render OCL operations on collections. The following rule translates OCL universal quantifications (`forAll`) in the format required by iContract:

```
FORALL ::= 'forall ' par3 ' ' par2
           ' in ' par1 '.elements() | ' par4;
```

The order of parameters is the same as the order of the child nodes of the `forall` node in the AST, i.e., in the OCL constraint: `par1` is the collection to be searched, `par2` is the iterator, `par3` is the class the iterator belongs to, and `par4` is the expression that must hold true for all collection elements. For the sake of simplicity, we have explicitly stated the class of the iterator, but this information could have been retrieved from the UML model. If we applied the rule to translating the OCL invariant on class "Driver", we would obtain:

```
/**
 * @invariant drivenRoutes.size() > 1 implies
 *  forall Route r
 *    in drivenRoutes.elements() |
 *  r.from ==
 *    ((Route) drivenRoutes.get(0)).from
 */
```

The same problem, i.e., universal quantifications, would have required the following rule to produce Nana-compliant expressions:

```
FORALL ::= 'A(' par3 ' ' par2 '=1, ' par2
           ' <= n, ' par2 '++ , ' par4 ')';
```

and the rule for the OCL postcondition on function `findRoute` is:

```
I(A(int i=1, i <= n, i++,
   ((trip[i-1].onTime <= trip[i].onTime) &&
   (trip[i-1].offTime <= trip[i].offTime) &&
   (trip[i-1].offStop == trip[i].onStop) &&
   ((trip[i].onTime - trip[i-1].offTime) >= 5))))
```

Rules permit also bindings between design (OCL) constraints and implementation code through Anna-like *virtual text*. Rules could easily refer to special-purpose operations (implementations) that are linked with the application during validation, but in this case, the approach cannot work alone, but it must suitably be paired with the "link" [6] features offered by the target programming language.

The prototype merges also produced assertions with the code they refer to.

[6] In this context, "link" features do not strictly refer to linker, but they refer also to inheritance and late-binding.

Currently, this is managed using suitable `awk` scripts that merge assertions and code according to notation-specific rules. In fact, different languages impose different requirements and a single and fixed solution would not be sufficient.

4 Related Work

The *Anna* assertion language for Ada is the ancestor of many modern assertion packages. *Anna* features that are particularly important for this work include non-local assertions (e.g., specifying a data structure invariant at one point in the program, to be evaluated whenever a variable is changed), and "virtual text," i.e., ghost variables and computations that are not needed in the actual program computation but which are useful as "bookkeeping" for a correctness argument. Assertion languages that support these Anna-like features are particularly well-suited as targets for translation of design language assertions.

Among the assertion systems that followed *Anna*, *ADL* (Assertion Definition Language) [6] is notable for providing a set of general-purpose concepts that can be rendered into the syntaxes of different programming languages. *ADL/C*, *ADL/C++*, *ADL/Java*, and *ADL/IDL* are examples of specific instantiations of the meta-notation [7]. Like OCL assertions, *ADL* assertions are separate from implementation source code rather than being embedded in it, and as in the approach described here the developer must provide some additional information to bind assertions to program units. The *ADL* approach is conceptually similar to the approach described here, the key differences (besides the assertion language itself, and many technical details) being our table-driven approach to defining new translations.

Prior approaches to supporting OCL have thus far involved special-purpose interpreters. For example, [11] proposes a meta-model for OCL as the starting point for implementing an interpreter. A similar approach is taken by [2], which describes some experiences on interpreting OCL, and by [1], which proposes some efficient ways for executing complex OCL assertions.

There are also translation approaches that aim at generating code that mimics OCL constraints. For example, we can cite the Dresden OCL toolkit([9]) and the OCLE (OCL Environment [10]). They do not aim at language-independence, but transform OCL into "equivalent" Java code to be embedded into more complex systems.

5 Conclusions and Future Work

We have described a customizable translation approach for propagating design assertions into implementation assertion languages. We have implemented key parts of the translation for OCL, and are evaluating and refining the approach through application to examples.

Several extensions to the current approach and its implementation are planned. Support for more complex relations between implementation entities and specification entities is needed, and supporting this will require extensions to the translation rule syntax to manage choices and user-defined mappings. The infrastructure for coordinating design and implementation artifacts is currently primitive. The approach will continue to be evaluated and refined through application to successively larger and more realistic examples as the prototype tools progress.

References

[1] P. Collet and R. Rousseau. Towards efficient support for executing the object constraint language. In *Tools'99*, 1999.

[2] A. Hamie, J. Howse, and S. Kent. Interpreting the object constraint language. In *Asia Pacific Conference in Software Engineering*. IEEE-CS, 1998.

[3] R. Kramer. iContract – The Java Design by Contract Tool. In *Proceedings of TOOLS26: Technology of Object-Oriented Languages and Systems*, pages 295–307. IEEE-CS Publisher, 1998.

[4] D. Luckham, F. von Henke, B. Krieg-Brückner, and O. Owe. *Anna - A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[5] P. Maker. GNU Nana – User's Guide (version 2.4). Technical report, School of Information Technology – Northern Territory University, July 1998.

[6] S. Microsystems. Adl language reference manual, 1993.

[7] M. Obayashi, H. Kubota, S. McCarron, and L. Mallet. The Assertion Based Testing Tool for OOP: ADL2. adl.opengroup.org/exgr/icse/icse98.htm, May 1998.

[8] Object Management Group, 429 Old Connecticut Path, Framingham, MA 01701, USA. *OMG Unified Modeling Language Specification*, version 1.5 edition, 2002. Available for download from www.omg.org.

[9] University of Dresden. The OCL toolkit, 2002. http://dresden-ocl.sourceforge.net/index.html.

[10] University of Cluj Napoca. The Object Constraint Language Environment, 2004. http://lci.cs.ubbcluj.ro/ocle/.

[11] M. Richters and M. Gogolla. A metamodel for OCL. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

[12] D. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Jan. 1995.

[13] J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, Mar. 1999.