Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 214 (2008) 331–357
www.elsevier.com/locate/entcs

Alloy as a Refactoring Checker?

H.-Christian Estler, Heike Wehrheim [1]

Institut für Informatik
Universität Paderborn
33098 Paderborn, Germany

Abstract

Refactorings are systematic changes made to programs, models or specifications in order to improve their structure without changing the externally observable behaviour. We will examine how a constraint solver (the Alloy Analyzer) can be used to automatically check if refactorings, applied to a formal specification (written in Z), meet this requirement. Furthermore, we identify a class of refactorings for which the use of this tool is reasonable in general.

Keywords: Alloy, refactoring, refinement, behaviour preservation, Z

1 Introduction

In the process of software development a programmer might have different objectives for changing an already working code. Such could be to simplify long winded operations, to improve reusability of constructs or merely to increase the readability of the code. These changes can be necessary, especially for software systems which evolve over long periods of time, to prevent unmanageable complexity. Although it is debatable what kind of changes might be most suitable to improve the structure, the demand to preserve the observable behaviour of the software is unambiguous. This means, given the same input, the original code and the changed version have to generate the same output.

Nowadays, it is common to use the term *refactoring*, which was first mentioned by Opdyke [13], for these kinds of behaviour-preserving changes. In

[1] Email: {estler,wehrheim}@uni-paderborn.de

doi:10.1016/j.entcs.2008.06.015

addition, besides referring to the actual process of changing software, refactorings are also generic descriptions of how the program should be modified. A popular book by Fowler [6] presents a collection of 72 refactorings to systematically improve the structure of (object-oriented) code. Still, refactorings are not limited to programs only. They are also used in modelling, for instance for refactoring UML models [14,20], or formal specifications [10,11].

In this paper, we focus on refactorings for the state-based formalism Z [19] and especially on ways to check their behaviour-preserving character. With *Refinement* [3,4] Z already provides a theory to prove that one specification meets the requirements set out by another. We will utilize this concept to verify behaviour-preservation. The refinement-proofs, which are based on predicate logic and set theory, are sometimes tedious and error-prone. Therefore, we will employ a constraint solver, namely the Alloy Analyzer [12], to show the correctness of the refactorings.

When it comes to analyze Z specifications, the idea of using Alloy as the tool of choice often appears obvious. Mainly, because the Alloy language, the input language for the Alloy Analyzer, is very similar to Z. In fact, Alloy was developed with the intention of bringing Z specification the kind of automation offered by model checkers. However, fully automatic simulation and checking requires the Alloy language to use first order logic only. Therefore, it is less expressive than Z and can roughly be seen as a subset of it.

The contributions of our paper are:

- We demonstrate exemplary the close relationship of Z and the Alloy language along with some limitations to the translation.
- We show that the Alloy Analyzer cannot verify refactorings by using simple refinement checks and explain the reasons for this.
- We present an approach that allows us to eventually use the Alloy Analyzer as a refactoring checker, as long as certain conditions hold.

In the following section we start with a small example of a Z specification and present its translation into the Alloy language. Section 3 explains why the Alloy Analyzer cannot be used to verify refinements and identifies the conditions under which refactorings can be checked nevertheless. We discuss some limitations of the translation in section 4 and conclude in the last section, along with a review of related works.

2 Translating Z into Alloy

2.1 The Z notation

Z is a declarative specification language which describes the states of a system and how they change under the execution of operations. Its formal semantic is based on set theory, first order predicate logic and lambda calculus. The main constructs of a Z specification are *schemas*, which are used to define the state space as well as the operations. A typical notation for such a schema is an "E-shaped box" (see figure 1). While the upper part of the schema contains one or more declarations of variables, the lower part consists of predicates which are all implicitly conjoined.

$[BOOK, PERSON]$

Library

$lent, lendable, books : \mathbb{P}\, BOOK$

$borrowers : \mathbb{P}\, PERSON$

$lent\_to : BOOK \pfun PERSON$

$lent \cap lendable = \varnothing$

$lent \cup lendable = books$

$\mathrm{dom}\, lent\_to = lent$

$\mathrm{ran}\, lent\_to \subseteq borrowers$

Fig. 1. A state schema in Z

The *Library* schema in figure 1 is a state schema, defining five variables which specify a simple library management system. Variables are always associated with a type. For instance, *borrowers* is of type $\mathbb{P}\, PERSON$, where $PERSON$ is an unspecified set of elements, a so called given set. The variable *lent_to*, however, represents a set of tuples (b, p), where $b \in BOOK$ and $p \in PERSON$. The way valid tuples can be constructed is defined by a partial function $BOOK \pfun PERSON$, indicating that several books can be lent to a single person. Constraints like $lent \cap lendable = \varnothing$ determine invariant properties of variables.

An operation describes the transition from one state (before-state) to another (after-state). Thus, using a schema to determine preconditions and postconditions in terms of states is sufficient to declaratively define the operation. Figure 2 shows such an operation, named *Add_book_ok*, to add a book

to the library system. The expression $\Delta Library$ is a shorthand notation that

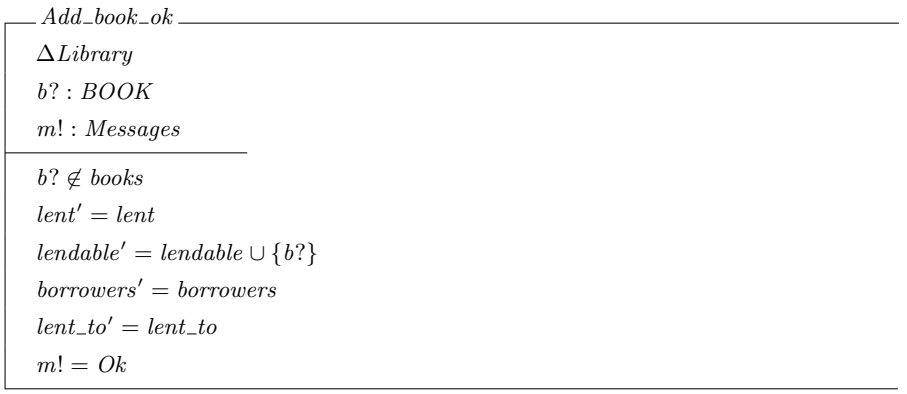

Fig. 2. An operation schema in Z

introduces all variables of the *Library* schema to the operation schema in an undecorated manner as well as a dashed one. Undecorated variables represent the before-state while dashed variables describe the after-state. Furthermore, it is a convention that variables which are followed by a "?", like $b?$, represent input variables and those followed by a "!" are output variables. An extended specification of this simple library system is shown in the appendix.

2.2 The Alloy language

The declarative specification language Alloy [8,9] is developed by the Software Design Group at MIT CSAIL. Unlike Z, it is strictly based on first order logic which facilitates automatic analysis. The Alloy Analyzer [12] provides fully automatic simulation and checking on specifications written in Alloy. It translates a given model into a boolean formula [7] and hands it to a SAT solver. The solver then tries to find a model of this formula. All examples we show in this chapter refer to the Alloy Analyzer in version 4.0 RC11. It is available free of charge at [12].

The structure of any Alloy model consists of *atoms* and *relations* between those. Atoms are indivisible, immutable, uninterpreted objects which are introduced through *signatures*. A simple but complete model would be:

```
module Example1

sig BOOK {}

sig Library{
   books: set BOOK
}
```

This model introduces two signatures, named BOOK and Library. While BOOK defines a set containing atoms BOOK0, BOOK1, BOOK2 and so on, the Library signature contains a field named books which relates an atom of a Library to a **set** of atoms of type BOOK. An instance of this model might consist of the following sets:

$$BOOK = \{(BOOK0), (BOOK1), (BOOK2)\}$$
$$Library = \{(Library0)\}$$
$$books = \{(Library0, BOOK1), (Library0, BOOK2)\}$$

In order to constrain a signature, Alloy allows us to append *facts*. For example, if we want to prohibit the existence of a library without any book, we could write:

```
sig Library{
    books: set BOOK
}{
some books
}
```

Using the multiplicity keyword **some** on the set books, we ensure that every library atom is related to at least one book atom. By adding an assertion, the Alloy Analyzer can check whether our model meets the desired behaviour:

```
assert NrBooks {
    all l: Library | #l.books > 0
}
```

Figure 3 shows a translation of the Z *Library* state schema we already used in the last subsection. The great conformity is eye-catching but not surprising. As we mentioned before, Z had a major influence on Alloy's development. A translation of Z *operations* is likewise simple. Alloy uses *predicates* to

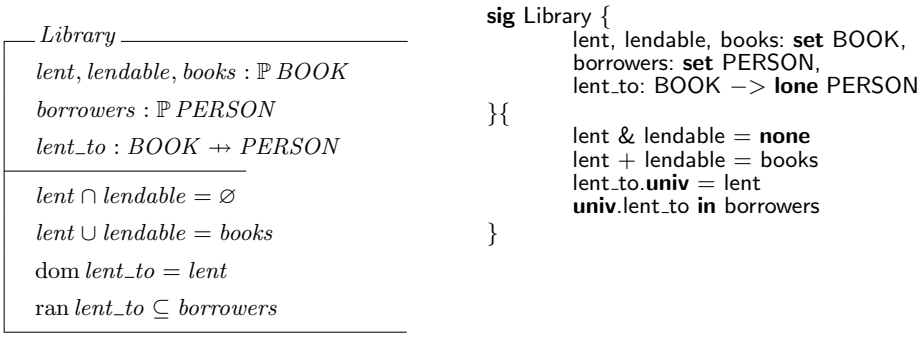

Fig. 3. Translation of a Z state schema into Alloy

describe them. In figure 4 we see that the identifier of a predicate is followed by a list of arguments which represent the declaration part of the Z schema.

Notice that the $\Delta Library$ shorthand is resolved according to its definition by introducing two (not necessarily distinct) variables l and l'. This way of

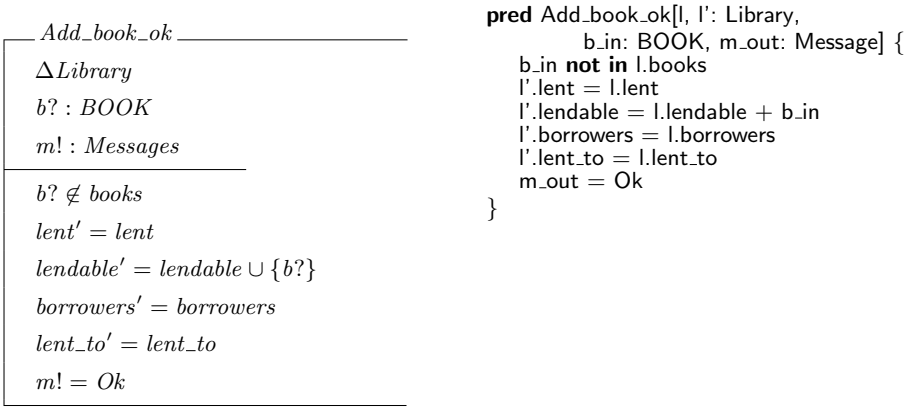

Fig. 4. Translation of a Z operation schema into Alloy

encoding Z specifications into Alloy is not the only possibility. Bolton [1], for instance, proposes another solution which we discuss in more detail in section 5. However, the approach presented here is probably the most intuitive and surely the least complex one.

3 Checking refactorings

Knowing how to translate Z specifications into Alloy we are now able to evaluate its use for checking refactorings. Before we start to analyse an exemplary refactoring let us again define the term refactoring by quoting Fowler from his book [6]:

> *Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.*

3.1 Examplary refactoring: Extract Method

The left hand side of figure 5 shows an extended version of the operation needed to add a book to our library system. The fact that there is a disjunction of two big "blocks" in the predicate part indicates that we can apply a common refactoring called Extract Method. Fowler [6, p.110] subsumes when and how it should be applied:

Status: You have a code fragment that can be grouped together.
Action: *Turn the fragment into a method whose name explains the purpose of the method.*

Even though this description is originally intended for regular source code, it still tells us how to improve the *Add_book* operation. We introduce two new operation schemas, one for each predicate block.

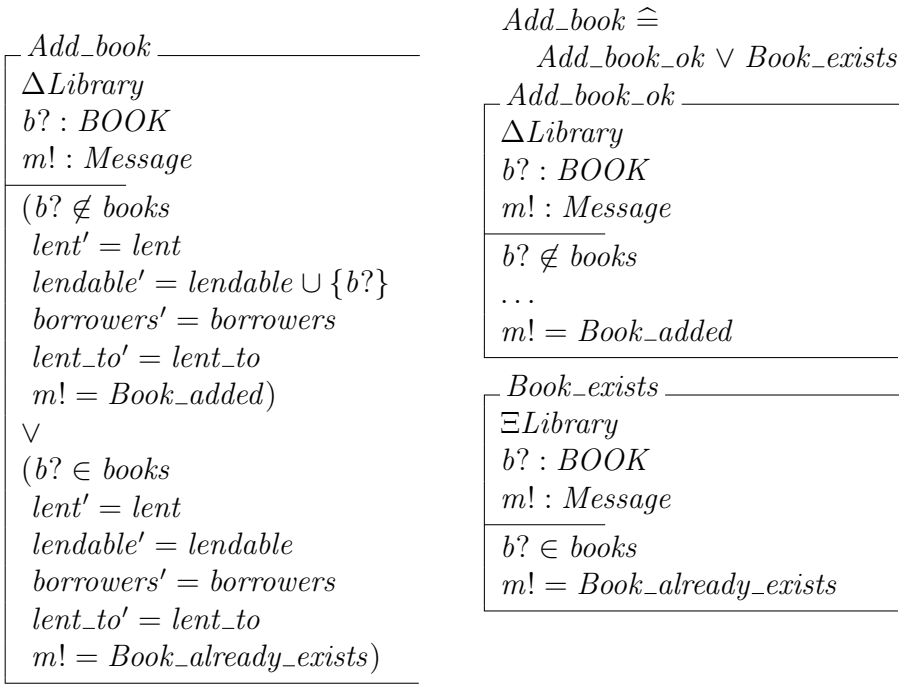

Fig. 5. Refactoring *Extract Method*

The right hand side of figure 5 shows the refactored operation. In order to be accurate, we used the Extract Method twice, first to extract *Add_book_ok* and second to extract *Book_exists*. The *Add_book* schema now only contains a disjunction of the two new schemas. The translation of the refactored operation into Alloy is again straightforward and can be found in the appendix.

3.2 Proving refactorings through refinement

In order to check the correctness of any refactoring we apply to a Z specification, a formal definition of behaviour-preservation is required. *Refinement* [3,4] allows us to change a specification while ensuring that certain properties remain the same. These properties refer to the external behaviour of the

specification. They must not change. This requirement is close to the one we have for refactorings. However, a refinement might change a specification in a one-sided manner (e.g. by removing non-determinism) so that its changes cannot be revoked. This is considered to be an undesirable behaviour for a refactoring. Therefore, we demand that not only a refactored specification is a refinement of the original one. Additionally, the original specification has to be a refinement of the refactored one. Thus we can define behaviour-preservation as follows:

Definition 3.1 A refactoring on a specification S, leading to a specification S_{Ref} is *behaviour-preserving* iff S_{Ref} is a refinement of S ($S \sqsubseteq S_{Ref}$) and S is a refinement of S_{Ref} ($S_{Ref} \sqsubseteq S$).

There is a solid theory on how refinements can be proven in Z. With the aid of a *downward simulation*, we can show that a Z data type C refines a data type A. With I being a fixed set of operations, [4, p. 90] gives a definition:

Definition 3.2 Given Z data types $A = (AState, AInit, \{AOp_i\}_{i \in I})$ and $C = (CState, CInit, \{COp_i\}_{i \in I})$. The relation R on $AState \wedge CState$ is a *downward simulation* from A to C if

(i) *the initial condition (Init) holds:*
$$\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \wedge R'$$

(ii) *for all $i \in I$ the correctness condition (Corr) holds:*
$$\forall\, AState;\ CState;\ CState' \bullet R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i$$

(iii) *for all $i \in I$, if (Corr) holds then the applicability condition (App) holds:*
$$\forall\, AState;\ CState \bullet R \Rightarrow (\mathrm{pre}\ COp_i \Leftrightarrow \mathrm{pre}\ AOp_i)$$

In this context, the operator pre is defined as follows (cp. [4, p. 33]):

$$\mathrm{pre}\ COp_i \ ::=\ \exists\, CState';\ Outs \bullet COp_i$$

Note that definition 3.2 assumes a *blocking semantics* which is typical for Object-Z [16] rather than Z. This is due to the fact that we are interested in using the results of this paper for Object-Z as well. Furthermore, definition 3.2 uses a *fixed* set of operations. A refactoring like *Extract Method* introduces new operations to the specification, though. Unlike Object-Z, Z does not allow us to highlight such operations as internal ones. These operations are, however, not part of the interface in any implementation. Therefore, the refinement conditions shown above need to be applied to the operations of such an interface.

If we think about a possible formulation of these conditions in Alloy,

we might be tempted to translate them directly. For instance, the (*Corr*) condition for the refactoring from the last subsection is:

```
assert Corr {
  all la: LibraryA, lc, lc': LibraryC, b: BOOK, m: Message |
        R[la, lc] and Add_book_ref[lc, lc', b, m]
        => {some la': LibraryA | R[la', lc'] and Add_book[la, la', b, m]}
           }
```

In this case, R[ ] is a predicate representing the relation R while LibraryA and LibraryC refer to the *Library* state schema before and after the refactoring, respectively. However, checking Corr, the Alloy Analyzer will fail to verify the assertion even though our refinement is correct. The reason for this is the use of an existential quantifier in the consequence of the implication.

3.3 Existential quantification in Alloy

In order to clarify the existential quantification problem, we start out with an example used by Daniel Jackson [9, p. 156]. Suppose, we want to check that sets are closed under the union operator. We write a model to specify sets:

```
sig Element {}

sig Set {
   elements: set Element
}
```

We want to check if for any two sets $s0$, $s1$, there is a set $s2$ containing both their elements. Thus, we write an assertion:

```
assert Closed {
   all s0, s1: Set | some s2: Set |
      s2.elements = s0.elements + s1.elements
}
```

Checking the assertion returns a counterexample. It shows an instance where the signature Set has only 2 atoms. Hence, there is no set which could represent the union.

The reason for this is the fact the Alloy Analyzer is a finite model-finder. Checking assertions is equal to finding an instance of the negated assertion. So given our assertion Closed, Alloy tries to find an instance that holds for the predicate:

```
   some s0, s1: Set | all s2: Set |
```

not (s2.elements = s0.elements + s1.elements)

If Set only contains two elements which are bound to the existential quantifier **some**, then s2 must be the empty set. Due to this, the complete expression trivially holds and the Analyzer returns the instance as a counterexample.

In other words, whenever the Alloy Analyzer checks a finite approximation of a full model, there might be no witness for the existential quantifier inside the approximation.

Jackson [9, p. 161] says: "Perhaps the most serious consequence of this issue is that assertions about preconditions, in which the precondition is asserted to be at least as weak as some property, cannot generally be checked."

One solution to this problem would be to constraint the Alloy model such that any producible object already exists. For the example above, such a constraint looks like this:

```
fact SetGenerator {
    some s: Set | no s.elements
    all s: Set, e: Element |
        some s': Set | s'.elements = s.elements + e
}
```

Due to its purpose, such a constraint is also called a *generator axiom*. It allows us to check a Z expression $\exists\, x : X \bullet E$, given that X is a finite data type.

Besides the problem that it can get complicated to figure out what a generator axiom looks like for a certain model, also the analysis becomes intractable as the scope explodes. For instance, if we specify a scope that bounds Set by n, only instances with at most $\log(n)$ atoms in Element will be considered. Using more complex data structures involving multirelations, e.g. to model graphs, we found that it is almost impossible to do any analysis in an appropriate time at all. Therefore, the use of generator axioms is not desirable.

Facing the result that we are unable to simply check refinement conditions, the question arises if Alloy can be of any use to check the correctness of refactorings at all. Fortunately, we can utilize certain properties of refactorings to improve the chances.

3.4 Simplifying the refinement conditions

If we think about the refactoring *Extract Method* from subsection 3.1, we can ask why it should be necessary to check a condition like (*App*). Intuitively, it is obvious that such a refactoring does not alter the applicability of the

operation. We will show that this intuition is correct. Knowing that the existential quantifier causes problems in the verification, we must formulate conditions which meet our definition of behaviour-preservation, but still are analysable in Alloy.

Note that when checking refactorings, we have the same downward simulation relation R for checking $A \sqsubseteq C$ and $C \sqsubseteq A$. Furthermore, if R has a particular form, the check for two downward simulations can be replaced by one *equality* check.

Definition 3.3 Given Z data types $A = (AState, AInit, (AOp_i)_{i\in I})$ and $C = (CState, CInit, (COp_i)_{i\in I})$. A and C are said to be *equivalent* $(A \equiv C)$ if there is a representation relation R between $AState$ and $CState$ such that

(i) $\widehat{Init}$: $\forall\, AState';\ CState' \bullet R' \Rightarrow (AInit \Leftrightarrow CInit)$,

(ii) $\widehat{Corr}$: $\forall\, AState;\ AState';\ CState;\ CState' \bullet R \wedge R' \Rightarrow (AOp_i \Leftrightarrow COp_i)$.

We write $A \equiv_R C$ when referring to a particular R.

Those specifications are equal if the representation relation is a homomorphism from A to C. Note that there is neither an applicability condition here nor any existential quantifier. In the case of a *total, bijective* retrieve relation R, equality checks guarantee downward simulations in both directions.

Theorem 3.4 *Let A, C be Z specifications, R a total bijective retrieve relation between $AState$ and $CState$. Then*

$$A \sqsubseteq_R C \wedge C \sqsubseteq_R A \text{ iff } A \equiv_R C$$

Proof: First note that totality and bijectivity guarantees the following property to hold :

$$(*) \quad \forall\, AState \bullet \exists_1\, CState \bullet R \quad \wedge \quad \forall\, CState \bullet \exists_1\, AState \bullet R$$

This is the main argument used within the proof. We start with the initialisation property:

$$\begin{aligned}
& \forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \wedge R' \\
\wedge\ & \forall\, AState' \bullet AInit \Rightarrow \exists\, CState' \bullet CInit \wedge R' \\
\overset{(*)}{\Leftrightarrow}\ & \forall\, CState';\ AState' \bullet (CInit \wedge R' \Rightarrow AInit) \\
\wedge\ & \forall\, AState';\ CState' \bullet (AInit \bullet R' \Rightarrow CInit) \\
\Leftrightarrow\ & \forall\, AState';\ CState' \bullet R' \Rightarrow (AInit \Leftrightarrow CInit)
\end{aligned}$$

The next condition is correctness:

$$\begin{aligned}
& \forall\, AState;\ CState;\ CState' \bullet R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i \\
\wedge\ & \forall\, AState;\ CState;\ AState' \bullet R \wedge AOp_i \Rightarrow \exists\, CState' \bullet R' \wedge COp_i \\
\overset{(*)}{\Leftrightarrow}\ & \forall\, AState;\ CState;\ CState';\ AState' \bullet R \wedge R' \wedge COp_i \Rightarrow AOp_i \\
\wedge\ & \forall\, AState;\ CState;\ CState';\ AState' \bullet R \wedge R' \wedge AOp_i \Rightarrow COp_i \\
\Leftrightarrow\ & \forall\, AState;\ CState;\ CState';\ AState' \bullet R \wedge R' \Rightarrow (COp_i \Leftrightarrow AOp_i)
\end{aligned}$$

Finally, we show that the applicability conditions of the two downward simulations already hold when we show the correctness condition of equality. Assume that $\widehat{Corr}$ holds, i.e. $\forall\, AState;\ AState';\ CState;\ CState' \bullet R \wedge R' \Rightarrow (AOp_i \Leftrightarrow COp_i)$ but App, i.e. $\forall\, AState;\ CState \bullet R \Rightarrow (\mathrm{pre}\ COp_i \Leftrightarrow AOp_i)$ does not hold. We consider the case where $\exists\, AState;\ CState \bullet R \wedge \mathrm{pre}\ COp_i \wedge \neg\, \mathrm{pre}\ AOp_i$, the other case is analogue.

$$\begin{aligned}
& \exists\, AState;\ CState \bullet R \wedge \mathrm{pre}\ COp_i \wedge \neg\, \mathrm{pre}\ AOp_i \\
\Rightarrow\ & \exists\, AState;\ CState;\ CState' \bullet R \wedge COp_i \wedge \neg AOp_i \\
\Rightarrow\ & \exists\, AState;\ CState;\ CState' \bullet R \wedge COp_i \wedge \neg\, \exists\, AState' \bullet AOp_i \\
\overset{(*)}{\Rightarrow}\ & \exists\, AState;\ CState;\ AState';\ CState' \bullet R \wedge R' \wedge COp_i \wedge \neg AOp_i
\end{aligned}$$

which contradicts $\widehat{Corr}$. □

Using defintion 3.1, 3.3 and theorem 3.4 we summarise:

Summary 1 *A refactoring on a specification S, leading to a specification S_{Ref} is behaviour-preserving iff S is equivalent to S_{Ref} ($S \equiv_R S_{Ref}$), where R is a total bijective representation relation.*

3.5 Checking equivalence of refactorings using Alloy

The results of the last subsections have shown that it is sufficient to check the two equivalence conditions from definition 3.3 in order to verify our exemplary refactoring *Extract Method*. For this refactoring, the representation relation R is total and bijective, in fact, it is simply the identity. Furthermore, we do not have to check the $\widehat{Init}$ condition, as the refactoring has no impact on the *Init* schema.

Figure 6 shows a shortened translation of the original *Add_Book* operation. We can simply add the refactored operations, given in figure 7, to our Alloy model.

In order to check the $\widehat{Corr}$ condition, we formulate the assertion shown in figure 8. Since the refactoring does not alter the state space of the

```
pred Add_book[l, l': Library, b_in: BOOK, m_out: Message] {
  { b_in not in l.books
    ...
    m_out = Book_added }
  or
  { b_in in l.books
    ...
    m_out = Book_already_exists }
}
```

Fig. 6. Alloy predicate of the *Add_book* operation

```
pred Ref_Add_book[l, l': Library, b_in: BOOK, m_out: Message] {
  Add_book_ok[l, l', b_in, m_out] or Book_exists[l, l', b_in, m_out]
}

pred Add_book_ok[l, l': Library, b_in: BOOK, m_out: Message] {
  b_in not in l.books
  ...
  m_out = Book_added
}

pred Book_exists[l, l': Library, b_in: BOOK, m_out: Message] {
  XiLibrary[l, l'] /* models Z's Xi convention */
  b_in in l.books
  m_out = Book_already_exists
}
```

Fig. 7. Refactored Alloy predicate of the *Add_book* operation

specification, we do not need to model an Alloy predicate to describe the representation relation R. Whenever R is the identity, a corresponding predicate will trivially hold. Hence, we can reduce the correctness to:

The assertion expresses the required equivalence without using any existential quantifier. Thus, we are now able to apply the Alloy Analyzer and it verifies the correctness of the refactoring within the actual scope.

We used Alloy to successfully verify further refactorings. Among them are

- Inline Method (see [6, p. 117]),

```
assert corr_ExtractMethod {
  all l, l': Library, b: BOOK, m: Message |
            Add_book[l, l', b, m] <=> Ref_Add_book[l, l', b, m]
}

check corr_ExtractMethod for 10 but 2 Library
```

Fig. 8. Assertion to check the refactoring *Extract Method*

- Substitute Algorithm (see [6, p. 139]) and
- Consolidate Conditional Expression (see [6, p. 240])

which can be found in the library example given in the appendix. All those refactorings do not change the state space. Therefore, they do not require the modelling of a relation representation R. A simple refactoring like *Rename Variables*, however, does (as it alters the state schema). In such a case the Alloy model needs to be extended not only with a signature for the refactored state schema and a predicate R, but also with modified versions of the operations (so that they use the renamed variables). Such an extended model allows us to check the correctness of the refactoring.

4 Limitations of the translation

It is obvious that Alloy's handling of existential quantifiers can complicate the translation of an arbitrary Z specification. During our work we faced this problem as well as some others which can be regarded as peculiarities of Alloy.

4.1 Integers and arithmetics

One of the basic types in Z is $\mathbb{Z}$, the set of integers. It can be used with the typical arithmetic operations, among them multiplication, division and modulo. Alloy has a predefined signature **Int** which by default contains atoms representing the set of integers $\{-8, -7, \ldots, 7\}$. However, the only operators supported are addition, subtraction and numerical comparison.

Working with integers in an Alloy model is somewhat inept. One reason is that **Int** is handled like every other signature. This means, it does not contain integers but atoms representing numbers. In order to get the numerical value of an atom a, the function **int**[a] has to be used. In contrast, the function **Int** [6] returns the atom corresponding to the numerical value 6. So, depending on the actual context, the operators + and − have a different meaning. If we talk about numerical values, they represent addition and subtraction, whereas

for atoms they denote union and difference.

Another reason is that we cannot exactly specify the range of the integers we would like to use. By default, the scope of **Int** is set to 4 **int**. This is $2^4 = 16$ integer values, ranging from -8 to 7. In a scope command we are only allowed to set the integer bitwidth. Hence, the next bigger scope will be 5 **int** with values from -16 to 15.

A motive for this limited integer support are technical issues. The Alloy Analyzer translates a model into a CNF formula and hands it to a SAT-Solver. Encoding numeric manipulations into CNF as well as converting from **Int** to **int** (and vice versa) is expensive. Therefore, Alloy is not well suited to work with heavily numerical specifications.

4.2 Schema calculus and Alloy signatures

Alloy signatures allow some sort of inheritance which is very useful in its own right (cp. [9, p. 91]). Nevertheless, this concept cannot generally be utilised to implement the schema operations available in Z. Examples for schema operations are *schema conjunction* and *schema extension*. In order to make a translation possible, we suggest to resolve all schema calculus operations on state schemas before translating the Z specification. This means, a single state schema containing all state data needs to be constructed. We must consider, though, that the information about schema operations will be lost in case of a retranslation.

4.3 Existential quantifier

The problem with the existential quantifier does not only affect the analysis, it also restricts a simple translation. Occasionally, we might want to define a local variable in the predicate part of an operation, e.g. by using a predicate like $\exists\, b : BOOK \bullet b = b?$ where $b?$ is an input variable of type $BOOK$. Then, the Alloy Analyzer will find instances where such a variable b does not exist. This also results in problems with the translation of refactorings like *Introduce Explaining Variable (IEV)* or *Extract Method (EM)*. Sometimes we are able to circumvent them. For instance, for the *IEV* refactoring we can use Alloy's **let** expression (cp. [9, p. 73]) which basically produces a syntactical replacement instead of creating a new variable. In other situations, however, there is no solution available yet. An example is the sequential composition of operations which can occur through the refactoring *EM*. Currently, Alloy can only be used to check if a composition of operations, e.g. $Op1 \fatsemi Op2$, implies (in terms of behaviour) a single operation Op. Verifying the opposite direction is not possible.

5 Discussion and conclusion

In this paper we investigated how Alloy can be used to check behaviour-preservation of refactorings. We demonstrated that the translation from a Z specification into the Alloy language is mostly simple and intuitive but at the same time has its limitations. Especially the existential quantifier produces problems within the translation and, even worse, prohibits a simple verification of refinements. Nevertheless, we were able to show that the Alloy Analyzer can be used to check the correctness of refactorings as long as we assume a total bijective representation relation between the original specification and the refactored one. This assumption is legitimate for a lot of refactorings.

The idea of using a verification tool to check refactorings has been investigated before. In [5] we made a case study on how the SAL model checker [2] can be used for this purpose. The results we obtained are comparable to those we presented in this paper. In both cases we only considered refactorings that can be applied to Z specifications. Refactorings changing object-oriented structures were not included. In both cases we successfully proved the correctness of the examined refactorings. Significant differences can be found in the translation of Z into the respective input language. SAL's translation is not as elegant as Alloy's and misses some fundamental operations (e.g. the $\mathbb{P}$ operator or support for partial functions). Smith and Wildman, however, presented possible solutions to this problem in [17]. In [18], Smith and Derrick have shown how to verify data refinements between Z specifications using the SAL CTL model checker. Their results imply that it is not necessary to assume a total bijective representation relation as we did in order to use Alloy. Thus, SAL is generally capable of analysing a wider spectrum of refactorings.

In section 2.2 we already mentioned Christie Bolton's paper *"Using the Alloy Analyzer to Verify Data Refinement in Z"* [1]. Her approach differs from ours significantly in the translation of the Z specification. Boltons idea demands to directly encode the state transition system defined by the Z specification. That is, every possible state needs to be represented by an Alloy atom and the relations between those atoms represent possible operations. Having such a model, the data refinement verification comes down to checking set inclusion. The problem with such an encoding is, that it equals the idea of introducing generator axioms (see subsection 3.3) to the model. It avoids the existential quantifier problem (for finite data types) but causes state explosion. Furthermore, as the effort for such a translation can be enormous even for small specifications, Bolton's approach seems reasonable only in case the

corresponding Alloy models can be fully automatically generated.

The Mondex case study with Alloy [15] by Tahina Ramananandro shows more similarities to our work. It demonstrates a translation of a Z specification into Alloy which is similar to the one we used in this paper. Consequently, Ramananandro experiences the same difficulties when trying to verify refinements. In order to overcome the problem of existential quantification, he introduces predicates to check if the model possesses enough properties to define the quantified object. These predicates were used subsequently in the hypotheses of implications to ensure that assertions will not be rejected due to an under-popularized instance of the model. This technique of checking refinements requires an argumentation why it is sound and leads to some additional modeling effort. As we have shown, we can save this effort whenever the correctness of refactorings needs to be checked.

The Z specification language has often been criticised for its lack of verification tools. Alloy fills this gap by offering the desired kind of fully automatic analysis, at least to a certain degree. The great conformity of the two languages often allows an easy translation and it can thereby support the process of writing specifications. With respect to the idea of building a refactoring-tool that uses Alloy or any other model checker only as a back-end system, however, an elegant translation is not significant. Instead, it would be necessary to identify if our assumption of a total bijective representation relation prohibits the checking of practically relevant refactorings. Probably even more important would be a performance analysis comparing the Alloy Analyzer to other model checkers like SAL. We leave this as future work.

References

[1] Christie Bolton. Using the Alloy Analyzer to Verify Data Refinement in Z. *Electr. Notes Theor. Comput. Sci.*, 137(2):23–44, 2005.

[2] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[3] W. DeRoever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA, 1999.

[4] John Derrick and Eerke Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology. Springer, May 2001.

[5] H.-Christian Estler, Thomas Ruhroth, and Heike Wehrheim. Modelchecking correctness of refactorings - some experiments. *Electronic Notes in Theoretical Computer Science*, 187:3–17, 2007.

[6] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Reading, MA, USA, 1999.

[7] Daniel Jackson. Automating first-order relational logic. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 130–139, New York, NY, USA, 2000. ACM Press.

[8] Daniel Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[9] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[10] T. McComb and G. Smith. Architectural Design in Object-Z. In *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*, pages 77–86, Washington, DC, USA, 2004. IEEE Computer Society Press.

[11] Tim McComb. Refactoring Object-Z Specifications. In Michel Wermelinger and Tiziana Margaria, editors, *FASE*, volume 2984 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2004.

[12] Software Design Group (MIT). The Alloy Analyzer, 2007. http://alloy.mit.edu/alloy4/.

[13] Johnson R. E. Opdyke W. F. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of SOOPPA'90: Symposium on Object-Oriented Programming Emphasizing Practical Applications.* ACM Press: New York NY, 1990.

[14] J. Philipps and Bernhard Rumpe. *Refactoring of Programs and Specifications.*, pages 281 – 297. Kluwer Academic Publishers, 2003.

[15] Tahina Ramananandro. Mondex, an electronic purse: specification and refinement checks with the alloy model-finding method. *Form. Asp. Comput.*, 20(1):21–39, 2007.

[16] G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publisher, 2000.

[17] G. Smith and L. Wildman. Model checking Z specifications using SAL. In *International Conference of B and Z Users*, volume 3455 of *Lecture Notes in Computer Science*, pages 85–103. Springer-Verlag, 2005.

[18] Graeme Smith and John Derrick. Verifying data refinements using a model checker. *Form. Asp. Comput.*, 18(3):264–287, 2006.

[19] J. M. Spivey. *The Z notation: a reference manual.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1992.

[20] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML Models. In Martin Gogolla and Cirs Kobryn, editors, *UML 2001: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 134–148, London, UK, 2001. Springer-Verlag.

6 Appendix

6.1 Z library example (before refactoring)

This specification shows the library example before any refactorings were applied.

$$[BOOK, PERSON]$$

$$
\begin{array}{rcl}
Message & ::= & Book\_added \\
& | & Book\_already\_exists \\
& | & Borrower\_added \\
& | & Not\_a\_library\_book \\
& | & Not\_on\_loan \\
& | & On\_loan \\
& | & No\_valid\_input \\
& | & Loan\_registered
\end{array}
$$

Library

$$
\begin{array}{l}
lent, lendable, books : \mathbb{P}\, BOOK \\
borrowers : \mathbb{P}\, PERSON \\
lent\_to : BOOK \pfun PERSON \\
\hline
lent \cap lendable = \varnothing \\
lent \cup lendable = books \\
\mathrm{dom}\, lent\_to = lent \\
\mathrm{ran}\, lent\_to \subseteq borrowers
\end{array}
$$

Init

$$
\begin{array}{l}
Library' \\
\hline
books' = \varnothing \\
borrowers' = \varnothing
\end{array}
$$

Add_book

$$
\begin{array}{l}
\Delta Library \\
b? : BOOK \\
m! : Message \\
\hline
(b? \notin books \\
\quad lent' = lent \\
\quad lendable' = lendable \cup \{b?\} \\
\quad borrowers' = borrowers \\
\quad lent\_to' = lent\_to \\
\quad m! = Book\_added) \\
\lor \\
(b? \in books \\
\quad lent' = lent \\
\quad lendable' = lendable \\
\quad borrowers' = borrowers \\
\quad lent\_to' = lent\_to \\
\quad m! = Book\_already\_exists)
\end{array}
$$

Add_a_borrower

$$
\begin{array}{l}
\Delta Library \\
p? : PERSON \\
m! : Message \\
\hline
p? \notin borrowers \\
borrowers' = addBorrower(borrowers, p?) \\
lent' = lent \\
lendable' = lendable \\
lent\_to' = lent\_to' \\
m! = Borrower\_added
\end{array}
$$

$$
\begin{array}{l}
addBorrower : \mathbb{P}(PERSON \times PERSON) \to \mathbb{P}\, PERSON \\
\hline
\forall\, b : \mathbb{P}\, PERSON;\ p : PERSON \bullet addBorrower(b, p) = b \cup \{p\}
\end{array}
$$

Enquire_about_a_book

$$\Xi Library$$
$$b? : BOOK$$
$$m! : Message$$

$$\neg(\neg(b? \notin lent \land b? \notin lendable) \lor m! \neq Not\_a\_library\_book)$$
$$\lor$$
$$\neg(\neg(b? \in books \land b? \notin lent) \lor m! \neq Not\_on\_loan)$$
$$\lor$$
$$\neg(\neg(b? \in books \land b? \notin lendable) \lor m! \neq On\_loan)$$

Lend_a_book

$$\Delta Library$$
$$b? : BOOK$$
$$p? : PERSON$$
$$m! : Message$$

$$b? \notin lendable \Rightarrow m! = No\_valid\_input$$
$$p? \notin borrowers \Rightarrow m! = No\_valid\_input$$
$$\lor$$
$$(b? \in lendable$$
$$p? \in borrowers$$
$$books' = books$$
$$lendable' = lendable \backslash \{b?\}$$
$$lent\_to' = lent\_to \cup \{(b?, p?)\}$$
$$m! = Loan\_registered)$$

6.2 Z library example (after refactoring)

This specification shows the library example after applying certain refactorings.

$$[BOOK, PERSON]$$

$$\begin{aligned} Message ::= & \; Book\_added \\ | & \; Book\_already\_exists \\ | & \; Borrower\_added \\ | & \; Not\_a\_library\_book \\ | & \; Not\_on\_loan \\ | & \; On\_loan \\ | & \; No\_valid\_input \\ | & \; Loan\_registered \end{aligned}$$

Library

$$lent, lendable, books : \mathbb{P}\, BOOK$$
$$borrowers : \mathbb{P}\, PERSON$$
$$lent\_to : BOOK \pfun PERSON$$

$$lent \cap lendable = \varnothing$$
$$lent \cup lendable = books$$
$$\mathrm{dom}\, lent\_to = lent$$
$$\mathrm{ran}\, lent\_to \subseteq borrowers$$

Init

$$Library'$$

$$books' = \varnothing$$
$$borrowers' = \varnothing$$

$$Add\_book \mathrel{\widehat=} Add\_book\_ok \lor Book\_exists$$

Add_book_ok

$$\Delta Library$$
$$b? : BOOK$$
$$m! : Message$$

$$b? \notin books$$
$$lent' = lent$$
$$lendable' = lendable \cup \{b?\}$$
$$borrowers' = borrowers$$
$$lent\_to' = lent\_to$$
$$m! = Book\_added$$

Book_exists

$$\Xi Library$$
$$b? : BOOK$$
$$m! : Message$$

$$b? \in books$$
$$m! = Book\_already\_exists$$

Add_a_borrower

$\Delta Library$
$p? : PERSON$
$m! : Message$

$p? \notin borrowers$
$borrowers' = borrowers \cup \{p?\}$
$lent' = lent$
$lendable' = lendable$
$lent\_to' = lent\_to'$
$m! = Borrower\_added$

Enquire_about_a_book

$\Xi Library$
$b? : BOOK$
$m! : Message$

$b? \notin books \Rightarrow m! = Not\_a\_library\_book$
$b? \in lendable \Rightarrow m! = Not\_on\_loan$
$b? \in lent \Rightarrow m! = On\_loan$

Lend_a_book

$\Delta Library$
$b? : BOOK$
$p? : PERSON$
$m! : Message$

$(b? \notin lendable \lor p? \notin borrowers) \Rightarrow m! = No\_valid\_input$
$\lor$
$(b? \in lendable$
$p? \in borrowers$
$books' = books$
$lendable' = lendable \backslash \{b?\}$
$lent\_to' = lent\_to \cup \{(b?, p?)\}$
$m! = Loan\_registered)$

6.3 Alloy module for the library example

This Alloy module contains the translation of both library examples, before and after refactorings were applied. It is used to check the correctness of the refactorings.

```
/*** Example of the library specification having both, the orginial and the refactored method
     and checks ***/
```

```
module Library_checking_Ref


/*** Given sets and free types ***/

sig BOOK, PERSON {}

abstract sig Message {}

one sig Book_added extends Message {}
one sig Book_already_exists extends Message {}
one sig Borrower_added extends Message {}
one sig Not_a_library_book extends Message {}
one sig Not_on_loan extends Message {}
one sig On_loan extends Message {}
one sig No_valid_input extends Message {}
one sig Loan_registered extends Message {}


/*** the "state" signature for the library ***/

sig Library {
        lent, lendable, books: set BOOK,
        borrowers: set PERSON,
        lent_to: BOOK -> lone PERSON
}{
        lent & lendable = none
        lent + lendable = books
        lent_to.univ = lent
        univ.lent_to in borrowers
}


/*** Init prediacte for the library, only need to proof that an instance exists ***/

pred Init [l': Library] {
        l'.books = none
        l'.borrowers= none
}

run Init for 10 but 1 Library


/*** Operation to add a book ***/

pred Add_book[l, l': Library, b_in: BOOK, m_out: Message] {
        { b_in not in l.books
                l'.lent = l.lent
                l'.lendable = l.lendable + b_in
                l'.borrowers = l.borrowers
                l'.lent_to = l.lent_to
                m_out = Book_added }
        or
        { b_in in l.books
                l'.lent = l.lent
                l'.lendable = l.lendable
                l'.borrowers = l.borrowers
                l'.lent_to = l.lent_to
                m_out = Book_already_exists }
}
```

```
/*** Now the refactored operation for adding a book, used Extract Method ***/

pred Ref_Add_book[l, l': Library, b_in: BOOK, m_out: Message] {
        Add_book_ok[l, l', b_in, m_out] or Book_exists[l, l', b_in, m_out]
}


pred Add_book_ok[l, l': Library, b_in: BOOK, m_out: Message] {
        b_in not in l.books
        l'.lent = l.lent
        l'.lendable = l.lendable + b_in
        l'.borrowers = l.borrowers
        l'.lent_to = l.lent_to
        m_out = Book_added
}


pred Book_exists[l, l': Library, b_in: BOOK, m_out: Message] {
        XiLibrary[l, l']
        b_in in l.books
        m_out = Book_already_exists
}

/*** assertion for correctnes of refactoring***/

assert corr_ExtractMethod {
                all l, l': Library, b: BOOK, m: Message |
                                Add_book[l, l', b, m] <=> Ref_Add_book[l, l', b, m]
}


/*** Operation to add a borrower ***/

pred Add_a_borrower[l, l': Library, p_in: PERSON, m_out: Message] {
        p_in not in l.borrowers
        l'.borrowers = addBorrower[l.borrowers, p_in]
        l'.lent = l.lent
        l'.lendable = l.lendable
        l'.lent_to = l.lent_to
        m_out = Borrower_added
}


/*** Function addBorrower used by the Add_a_borrower operation ***/

fun addBorrower[b: set PERSON, p: PERSON]: set PERSON {
        b + p
}


/*** Now the refactored operation for adding a borrower, used Inline Method ***/

pred Ref_Add_a_borrower[l, l': Library, p_in: PERSON, m_out: Message] {
        p_in not in l.borrowers
        l'.borrowers = l.borrowers + p_in
        l'.lent = l.lent
        l'.lendable = l.lendable
        l'.lent_to = l.lent_to
        m_out = Borrower_added
}
```

```
/*** assertion for correctnes of refactoring***/

assert corr_InlineMethod {
                all l, l': Library, p: PERSON, m: Message |
                                Add_a_borrower[l, l', p, m] <=> Ref_Add_a_borrower[l, l', p,
                                    m]
}


/*** Operation to enquire about a book ***/

pred Enquire_about_a_book[l', l: Library, b_in: BOOK, m_out: Message] {
        XiLibrary[l, l']
        not ( not (b_in not in l.lent and b_in not in l.lendable) or m_out !=
            Not_a_library_book)
        or
        not ( not (b_in in l.books and b_in not in l.lent) or m_out != Not_on_loan)
        or
        not( not (b_in in l.books and b_in not in l.lendable) or m_out != On_loan)
}


/*** Now the refactored operation for enquire a book, used Substitue Algorithm ***/

pred Ref_Enquire_about_a_book [l, l': Library, b_in: BOOK, m_out: Message] {
        XiLibrary[l, l']
        b_in not in l.books => m_out = Not_a_library_book
        b_in in l.lendable => m_out = Not_on_loan
        b_in in l.lent => m_out = On_loan
}

/*** assertion for correctnes of refactoring***/

assert corr_SubstituteAlgo {
        all l, l': Library, b: BOOK, m: Message |
                                        Enquire_about_a_book[l, l', b, m] <=>
                                            Ref_Enquire_about_a_book[l, l', b, m]
}


/*** Operation to lend a book ***/

pred Lend_a_book[l, l': Library, b_in: BOOK, p_in: PERSON, m_out: Message] {
        b_in not in l.lendable => m_out = No_valid_input
        p_in not in l.borrowers => m_out = No_valid_input
        or
        { b_in in l.lendable
                p_in in l.borrowers
                l'.books = l.books
                l'.lendable = l.lendable - b_in
                l'.lent_to = l.lent_to + b_in ->p_in
                m_out! = Loan_registered
        }
}


/*** Now the refactored operation for lending a book, used Consolidate Conditional
    Expression ***/
```

```
pred Ref_Lend_a_book[l, l': Library, b_in: BOOK, p_in: PERSON, m_out: Message] {
        (b_in not in l.lendable or p_in not in l.borrowers) => m_out = No_valid_input
        or
        { b_in in l.lendable
                p_in in l.borrowers
                l'.books = l.books
                l'.lendable = l.lendable − b_in
                l'.lent_to = l.lent_to + b_in ->p_in
                m_out! = Loan_registered
        }
}


/*** assertion for correctnes of refactoring***/

assert corr_CCExp {
        all l, l': Library, b: BOOK, p: PERSON, m: Message |
                                        Lend_a_book[l, l', b, p, m] <=> Ref_Lend_a_book[l,
                                                l', b, p, m]
}


/*** XiLibrary is the predicate that we use to simulate the Xi−Notation ***/

pred XiLibrary[l, l': Library] {
        l'.lent = l.lent
        l'.lendable = l.lendable
        l'.books = l.books
        l'.borrowers = l.borrowers
        l'.lent_to = l.lent_to
}


check corr_ExtractMethod for 10 but 2 Library
check corr_InlineMethod for 10 but 2 Library
check corr_SubstituteAlgo for 10 but 2 Library
check corr_CCExp for 10 but 2 Library
```