Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 187 (2007) 3–17

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Modelchecking Correctness of Refactorings - Some Experiments

H.-Christian Estler, Thomas Ruhroth, Heike Wehrheim[1]

Institut für Informatik
Universität Paderborn
33098 Paderborn, Germany

Abstract

Refactorings are changes made to programs, models or specifications with the intention of improving their structure and thus making them clearer, more readable and re-usable. Refactorings are required to be *behaviour-preserving* in that the external behaviour of the program/model/specification remains unchanged. In this paper we show how a simple type of refactorings on object-oriented specifications (written in Object-Z) can be formally shown to be behaviour-preserving using a modelchecker (SAL). The class of refactorings treated covers those operating on a single method only.

Keywords: Refactoring, Object Z, Model Checking, SAL

1 Introduction

Refactoring is a technique which has long been used by programmers to improve the structure of their code once it got unreadable. The Ph.D. thesis of Opdyke [16] and even more the book of Fowler [11] made it popular, coined the term "Refactoring" and started a systematic study of it. According to Fowler [11],

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

Here, improvement of structure refers to object-oriented structuring (i.e. class hierarchies) in as much as the same way as to modularisation of classical imperative programs. Nowadays, refactoring is a technique which is not only applied on programs but also used in modelling, for instance for refactoring UML models [21,17] or formal specifications [12,13]. For a survey on software refactoring see [15].

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

doi:10.1016/j.entcs.2006.08.041

While "improving the structure" is a rather soft requirement and usually debatable (what is a "good" structure?), the sidecondition of not altering the external behaviour is more precise. It should guarantee that users of the code or model, may this be human beings or other software artefacts, can use the code after refactoring as before. The change should remain transparent. For code, behaviour preservation is usually checked by an extensive testing of the code after every refactoring.

In this paper, we are interested in refactorings on *formal specifications*. Large formal (object-oriented) specifications can exhibit the same deficiencies with respect to clarity and well-structuredness as programs. Hence refactoring can also be a help in improving the structure of specifications (or in modifying the structure with a view to a later implementation). Although there is no one-to-one correspondence between concepts in programming languages and those in formal specifications, similarities are (obviously) there and some of Fowler's refactorings have their direct counterparts in formal specifications.

The specification language we are interested in is Object-Z [18], an object-oriented extension of the state-based formalism Z [22]. Refactorings of Object-Z specifications have already been treated in [12,13,14]. In a formal setting the criterion of behaviour preservation can be strictly defined: *Refinement* [6,5] (or even *equivalence*, i.e. refinement in both directions) guarantees the desired degree of substitutability. Thus data refinement (or more precisely, class refinement) is the correctness criterion for the refactorings considered in [13,14]. While [13,14] defines refactorings changing the object-oriented structure (e.g. the class hierarchy) of specifications, here we will only treat refactorings operating on single methods only (e.g. extracting methods, simplifying conditions, substituting algorithms). If the change concerns just one method the condition of behaviour preservation boils down to checking *equivalence* of the method definition before and after refactoring. This trivially results in a refinement relationship between the class before and after the refactoring.

In (Object-)Z, equivalence can be proven by applying the rules of predicate logic, set theory and Z's mathematical tool kit. However, such a proof is usually tedious and often error-prone (though it can be made precise by the use of a theorem prover). Here, we will instead employ a modelchecker (SAL [3]) for showing equivalence. Our technique combines the ideas of [20,7] for modelchecking Z with SAL with those of [19] on formulating refinement as a CTL modelchecking question. We have carried out some experiments on using this type of equivalence checking as a correctness proof for refactorings and here we report on the results.

The paper is structured as follows. We start with a small example of an Object-Z class on which we illustrate four different refactorings of methods, and on which we discuss our notion of behaviour preservation. Section 3 will then give a short introduction into SAL and show how to check correctness of refactorings with SAL. The last section concludes and discusses other type of refactorings, in particular with respect to the notion of correctness needed for them.

2 Example Refactorings

Refactoring is a method for modifying software without changing its behaviour. Using refactorings is common to software developers, but it is also useful to refactor formal specifications. In the following we give an example Object-Z specification and discuss some refactorings on it.

When applying refactorings to specification or programs, structural differences between the refactorings can easily be seen. Some refactorings change methods internally and leave all other components of the class untouched. Other refactorings change parts of the class hierarchy and leave classes outside their range unchanged. A classification of refactorings might thus consider their *scope* as the main differentiating criterion. We found four scopes on which refactorings operate:

- Method,
- class,
- class hierarchy,
- system.

This strict classification however cannot always be kept since there may be refactorings which are applied on methods but still neccessitate consequent changes on the rest of the specification (e.g. renaming a method). Thus, method refactorings can furthermore be divided into outer refactorings (with consequent changes on the rest of the specification) and inner method refactorings. It is the latter which we treat in this paper.

As a running example for our refactorings we use the specification of a sauna given in Figure 1. The first part of the class is the visibility list. This list declares the visible members of the class. All members listed in here are visible outside the class (like public in Java), otherwise they are not visible outside the class (like private/protected in Java). The state schema defines some variables and global constraints on the variables. Following the state schema there is an initialisation schema (here left open since it is of no particular importance for the refactorings) and a number of operation schemas. We have two operations: *updateControlLights* and *heat*. The method *updateControlLights* updates the control indicators for the right temperature (*clOk*), temperature too hot (*clHot*) and temperature too cold (*clCold*). If the current temperature differs from the target temperature for less or equal than 2 degrees the temperature is considered to be in order, otherwise the control indicators are set to "too hot" or "too cold", respectively. If the sauna is getting cold, it can be heated by the operation *heat*. Heating also changes the *currentHumidity*.

Looking at the class *Sauna* we find that some method definitions (in particular *updateControlLights*) are not easy to understand. We will next change them according to some refactoring rules. Here, we apply three kinds of refactorings on *updateControlLights* and one on *heat*. The names of the refactorings are chosen in accordance with the naming in Fowler's book.

Sauna

$\upharpoonright (\mathit{updateControlLights}, \mathit{heat}, \ldots)$

$$\begin{array}{l}
\mathit{targetTemp} : \mathbb{Z} \\
\mathit{currentTemp} : \mathbb{Z} \\
\mathit{clOk} : \mathbb{B} \\
\mathit{clCold} : \mathbb{B} \\
\mathit{clHot} : \mathbb{B} \\
\mathit{currentHumidity} : \mathbb{Z} \\
\hline
70 \leq \mathit{targetTemp} \leq 100 \\
\mathit{targetTemp} \bmod 5 = 0 \\
60 \leq \mathit{currentTemp} \leq 100 \\
0 \leq \mathit{currentHumidity} \leq 100
\end{array}$$

INIT

$$\vdots$$

updateControlLights

$$\begin{array}{l}
\Delta(\mathit{clOk}, \mathit{clHot}, \mathit{clCold}) \\
\hline
\mathit{currentTemp} - \mathit{targetTemp} < -2 \Rightarrow \\
\quad (\mathit{clOk}' = \text{false} \wedge \mathit{clHot}' = \text{false} \wedge \mathit{clCold}' = \text{true}) \\
\mathit{currentTemp} - \mathit{targetTemp} > 2 \Rightarrow \\
\quad (\mathit{clOk}' = \text{false} \wedge \mathit{clHot}' = \text{true} \wedge \mathit{clCold}' = \text{false}) \\
0 \geq \mathit{currentTemp} - \mathit{targetTemp} \geq -2 \Rightarrow \\
\quad (\mathit{clOk}' = \text{true} \wedge \mathit{clHot}' = \text{false} \wedge \mathit{clCold}' = \text{false}) \\
0 \leq \mathit{currentTemp} - \mathit{targetTemp} \leq 2 \Rightarrow \\
\quad (\mathit{clOk}' = \text{true} \wedge \mathit{clHot}' = \text{false} \wedge \mathit{clCold}' = \text{false})
\end{array}$$

heat

$$\begin{array}{l}
\Delta(\mathit{currentHumidity}, \mathit{currentTemp}) \\
\hline
\mathit{currentTemp} \leq \mathit{targetTemp} + 2 \\
\mathit{currentTemp}' = \mathit{currentTemp} + 2 \\
\mathit{currentHumidity}' = \mathit{currentTemp}' \operatorname{div} 5
\end{array}$$

$$\vdots$$

Fig. 1. Sample class: Sauna

Introduce Explaining Variable

The first refactoring exploits the fact that the expression *currentTemp* − *targetTemp* repeatedly occurs in the predicate of *updateControlLights*. We apply the refactoring rule "Introduce Explaining Variable" which – according to Fowler [11] – is defined as follows: "You have a complicated expression. Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.”

$$
\begin{array}{|l}
\textit{updateControlLightsIEV} \\
\hline
\Delta(\textit{clOk}, \textit{clHot}, \textit{clCold}) \\
\hline
\exists\, \textit{diff} : \mathbb{Z} \bullet \textit{diff} = \textit{currentTemp} - \textit{targetTemp} \land \\
\quad \textit{diff} < -2 \Rightarrow \\
\quad\quad (\textit{clOk}' = \text{false} \land \textit{clHot}' = \text{false} \land \textit{clCold}' = \text{true}) \land \\
\quad \textit{diff} > 2 \Rightarrow \\
\quad\quad (\textit{clOk}' = \text{false} \land \textit{clHot}' = \text{true} \land \textit{clCold}' = \text{false}) \land \\
\quad 0 \leq \textit{diff} \leq 2 \Rightarrow \\
\quad\quad (\textit{clOk}' = \text{true} \land \textit{clHot}' = \text{false} \land \textit{clCold}' = \text{false}) \land \\
\quad 0 \geq \textit{diff} \geq -2 \Rightarrow \\
\quad\quad (\textit{clOk}' = \text{true} \land \textit{clHot}' = \text{false} \land \textit{clCold}' = \text{false}) \\
\hline
\end{array}
$$

Here, we have introduced the new variable *diff*. The introduction of a new variable is expressed via an existential quantifier.

Consolidate Condition by Function

An alternative way of refactoring the recurring expression is applying ”Consolidate Condition”. Instead of using an extra variable we use functions to express the conditionals. Again according to Fowler: ”You have a sequence of conditional tests with the same result. Combine them into a single conditional expression and extract it.”

$$
\begin{array}{|l}
\textit{abs} : \mathbb{Z} \rightarrow \mathbb{N} \\
\hline
x < 0 \Rightarrow \textit{abs}(x) = -x \\
x \geq 0 \Rightarrow \textit{abs}(x) = x
\end{array}
\qquad\qquad
\begin{array}{|l}
\textit{tempdiff} : \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Z} \\
\hline
\forall\, n, m : \mathbb{Z} \bullet \textit{tempdiff}(m, n) = m - n
\end{array}
$$

$$
\begin{array}{|l}
\textit{updateControlLightsCCF} \\
\hline
\Delta(\textit{clOk}, \textit{clHot}, \textit{clCold}) \\
\hline
\textit{tempdiff}(\textit{currentTemp}, \textit{targetTemp}) < -2 \Rightarrow \\
\quad (\textit{clOk}' = \text{false} \land \textit{clHot}' = \text{false} \land \textit{clCold}' = \text{true}) \\
\textit{tempdiff}(\textit{currentTemp}, \textit{targetTemp}) > 2 \Rightarrow \\
\quad (\textit{clOk}' = \text{false} \land \textit{clHot}' = \text{true} \land \textit{clCold}' = \text{false}) \\
\textit{abs}(\textit{tempdiff}(\textit{currentTemp}, \textit{targetTemp})) \leq 2 \Rightarrow \\
\quad (\textit{clOk}' = \text{true} \land \textit{clHot}' = \text{false} \land \textit{clCold}' = \text{false}) \\
\hline
\end{array}
$$

In this example we proceed by introducing two new functions (*abs* and *tempdiff*). These are then used within the predicates.

Substitute Algorithm

All versions of *updateControlLights* used up to here are still long winded. The next refactoring will change this. Instead of testing temperature range and adjusting all lights we define the temperature range for each light and adjust each light separately. Thus *updateControlLights* is replaced by a completely new definition (*updateControlLightsSA*). This is an application of the refactoring "Substitute Algorithm".

updateControlLightsSAe

$$\Delta(\mathit{clOk}, \mathit{clHot}, \mathit{clCold})$$

$$\begin{array}{l}
\mathit{clCold}' = \mathit{currentTemp} \leq \mathit{targetTemp} - 2 \\
\mathit{clHot}' = \mathit{currentTemp} \geq \mathit{targetTemp} + 2 \\
\mathit{clOk}' = 2 \geq \mathit{currentTemp} - \mathit{targetTemp} \geq -2
\end{array}$$

Extract Method

Last, we take a look at method *heat*. The method adjusts both *currentTemp* and *currentHumidity*. This is now split into two new methods which are then combined via sequential composition (a special case of the refactoring "Extract Method"). The Δ-lists are trimmed to fit only modified variables. Because the new operations are not in the visibility list, they cannot be invoked directly from other classes.

Sauna

$\vdots$

heatT

$$\Delta_{(}\mathit{currentTemp})$$

$$\begin{array}{l}
\mathit{currentTemp} \leq \mathit{targetTemp} + 2 \\
\mathit{currentTemp}' = \mathit{currentTemp} + 2
\end{array}$$

heatH

$$\Delta(\mathit{currentHumidity})$$

$$\mathit{currentHumidity}' = \mathit{currentTemp}\ \mathit{div}\ 5$$

$$\mathit{heat} \mathrel{\widehat=} \mathit{heatT} \mathbin{⨾} \mathit{heatH}$$

$\vdots$

After having carried out these refactorings, we finally have to make sure that these are all *behaviour-preserving.* For this we first have to define behaviour preservation for inner method refactorings. Our intention was to get methods which are *equivalent* to the old ones. As a consequence the refactored and the original class would then be refinements of each other.

Let $(State, Init, (Op_i)_{i \in I})$ be the original class and $(State, Init, (Op_i)_{i \in I \setminus \{j\}}, \widehat{Op_j})$ the refactored class with operation Op_j changed to $\widehat{Op_j}$. Method Op_j is *equivalent* to method $\widehat{Op_j}$ if the following two conditions hold:

$$(1)\ \forall\, State \bullet \mathrm{pre}\, Op_j \Leftrightarrow \mathrm{pre}\, \widehat{Op_j}$$
$$(2)\ \forall\, State, State' \bullet Op_j \Leftrightarrow \widehat{Op_j}$$

Condition (1) is the analogon of the applicability rule of refinement and rule 2 the correctness rule. Both need only be checked for refactored methods.

3 Checking Correctness

In this section we show how to assure that our refactorings given in the last section are *behaviour-preserving* in the above defined sense. To this end we will translate our Object-Z schemas into the SAL language and verify equivalence of methods using SAL's CTL model checker. Our following approaches of translation and verification are based upon the ideas described in [19] and [20].

3.1 SAL

The Symbolic Analysis Laboratory (SAL) [3] is developed by the Formal Methods Program at SRI International. It is a framework combining different tools for program analysis, theorem proving and model checking of state-transition systems. Currently SAL provides four different model checkers, one of them for checking CTL properties [9], a simulator and some other tools which all work on the same input language, called SAL language.

The SAL language was developed as an intermediate language, serving as a target platform for translators of high-level languages like Java. Therefore it supports a wide range of type definitions and expressions. Nevertheless, the SAL language can also be used to describe transition systems in their own right. It is not that different from languages used by verification tools like SMV, Murphi or SPIN. A complete specification of the SAL Language can be found in the SAL Language Manual [4]. At this point we introduce some of the basic constructs necessary to understand the SAL encodings of our Z specifications.

Context

All SAL inputs like declarations, modules or theorems are grouped together in a `CONTEXT`. We consider a translation for our sauna specification to illustrate the SAL language structure.

```
sauna: CONTEXT =
BEGIN
  Int: TYPE = [-100..100]; %our own integer type
  Nat: TYPE = [0..100];     % our own natural type
  neg(n: Nat) : Int = -n;  %function to negate n
                           %not yet used in context
main: MODULE =
BEGIN
 LOCAL targetTemp, currentTemp: Int
 LOCAL clOk, clCold, clHot : BOOLEAN
 LOCAL currentHumidity: Int
 LOCAL Invar: BOOLEAN
DEFINITION
     Invar = (70 <= targetTemp) AND (targetTemp <= 100) AND
             (targetTemp MOD 5 = 0) AND
             (60 <= currentTemp) AND (currentTemp <= 100) AND
             (0 <= currentHumidity) AND (currentHumidity <= 100)
INITIALIZATION
[    Invar
     --> targetTemp IN {i: Int| TRUE};
         currentTemp IN {i: Int| TRUE};
         clCold IN {b: BOOLEAN| TRUE};
         clHot IN {b: BOOLEAN| TRUE};
         clOk IN {b: BOOLEAN| TRUE};
         currentHumidity IN {i: Int| TRUE}
]
TRANSITION
[
updateCL:
      (currentTemp - targetTemp < -2)
      => (clOk' = FALSE AND clHot' = FALSE AND clCold' = TRUE) AND
      (currentTemp - targetTemp > -2)
      => (clOk' = FALSE AND clHot' = TRUE AND clCold' = FALSE) AND
      (0 >= currentTemp - targetTemp) AND (currentTemp - targetTemp >= -2)
      => (clOk' = TRUE AND clHot' = FALSE AND clCold' = FALSE) AND
      (0 <= currentTemp - targetTemp) AND (currentTemp - targetTemp <= 2)
      => (clOk' = TRUE AND clHot' = FALSE AND clCold' = FALSE) AND
     Invar
     --> clCold' IN {b: BOOLEAN| TRUE};
         clHot' IN {b: BOOLEAN| TRUE};
         clOk' IN {b: BOOLEAN| TRUE}
[]
heat:
     (currentTemp <= targetTemp + 2) AND
     (currentTemp' = currentTemp + 2) AND
     (currentHumidity' = currentTemp' div 5) AND
     Invar
     --> currentTemp' IN {i: Int| TRUE};
         currentHumidity' IN {i: Int| TRUE}
[]
ELSE -->
]
END;
END
```

Comments are preceeded by the % symbol and terminated by an end-of-line. The SAL Language is case sensitive and reserved words like `CONTEXT, BEGIN` and `END` are written in capital letters. In the following we explain the parts of our SAL specification step by step.

Declaration

SAL supports built-in basic types like `BOOLEAN, NATURAL, INTEGER` or `REAL`. All of them can be used with the infinite-bounded model checker of the SAL toolbox. But as the CTL model checker works on finite types we have to declare our own types by implementing them in the form `<Name>: <Type>`. To define a new type we use the following construct `<Name>: <Type>= <Expression>`, which can also be used to express functions. In our example we have defined our own natural and integer types (`Int, Nat`) as well as a function to negate a natural number.

Module

A Module basically describes a state-transition system. It consists of variables which can be `INPUT, LOCAL, GLOBAL` or `OUTPUT`. Invariants are described in the `DEFINITION` section, initial values are declared in the `INITIALIZATION` part and

the transition functions are marked by the keyword `TRANSITION`.
For a single transition function we always use a *Guarded Command* that consists of a *Guard* and an *Assignment*.

$$Guard --> Assignment$$

SAL nondeterministically chooses values for the variables given in the assignments if the conditions defined by the guard are fullfilled. Let us consider the following example:

```
currentTemp < 60
--> currentTemp’ = currentTemp + 1
```

This guarded command is nothing more than just an *if..then* expression. We can retype it as

```
currentTemp’ = IF (currentTemp < 60) THEN currentTemp + 1.
```

Much more interesting are constructs like

```
58 <= currentTemp’ AND
currentTemp’ <= 62
--> currentTemp’ IN {i: Int| TRUE}
```

If a transition with such an command is executed we get 5 successor states with values for *currentTemp* between 58 and 62. Hence guarded commands are much more powerful constructs than just simple *if..then* expressions. As described in [20] we use them as an universal method to encode predicates of Z schemas into SAL.

Finally, attention should be paid to the last guarded command `ELSE -->` in our example. To receive correct results from the modelchecker it is necessary that the transition relation is total such that a module cannot deadlock. This special guard evaluates to true iff all other guards evaluate to false and leaves the states unchanged.

Formulas
The properties to be checked on a SAL specification have to be written in theorems below the module they refer to. For example, `th1: THEOREM main |- AG(EX(clOk = TRUE));` defines a theorem named *th1* that states the following property (given in CTL [9]): on all execution paths every state has a next state where `clOk` is true.

3.2 Model checking refactorings using SAL

G. Smith and J. Derrick have shown how to verify data refinements of Z specifactions using the SAL model checker and a CTL encoding of the simulation rules [19]. Our refactorings of Object-Z specifications only work on simple methods, not on the object-oriented structure, therefore we can reuse this technique for checking equivalence of methods.

As defined in the last section we need to prove an applicability and a correctness condition. Basically, this means that we have to verify that our original sauna specification $S = (\mathit{State}, \mathit{Init}, (\mathit{Op}_i)_{i \in I})$ and the specification $\widehat{S} = (\mathit{State}, \mathit{Init}, (\mathit{Op}_i)_{i \in I \setminus \{j\}}, \widehat{\mathit{Op}_j})$ with the refactored operation $\widehat{\mathit{Op}_j}$ act exactly the same way.

A symbolic model checker like the `sal-wmc` builds up a symbolic representation of a Kripke structure that represents all reachable states. A intuitive idea may be to prove if the structures of S and $\widehat{S}$ are identically and leads us to same-valued states given an arbitrary sequence of operations, respectively. But since model checkers are not designed to compare different structures we need to combine S and $\widehat{S}$ into a single system and define this combined system in such a way that we can actually verify applicability and correctness. The following technique is essentially based on [19].

The starting point is the SAL specification of our original class. In a first step we have to solve the problem that CTL does not allow propositions referring to operations. Therefore we augment $\mathit{State} : \mathit{Exp}$ by a variable ev. Furthermore we augment every operation Op_i by $ev' = \mathit{Op}_i$ so that ev gives us unique information which operation led to the actual state. To ensure that the transition relation is total we introduce another operation *Choose* which is always enabled and chooses a new state (*Choose* replaces the `ELSE -->` guard).

Next, we additionally extend our system by a copy of $\mathit{State} : \mathit{Exp}$: $\widehat{\mathit{State}}$ (since we need to compare the effect of the old method with that of the new). For every variable x in $\mathit{State} : \mathit{Exp}$ we declare a new variable $\widehat{x}$. The refactored operation $\widehat{\mathit{Op}_j}$ is now added to the specification, it sets $ev' = \widehat{\mathit{Op}_i}$ and is modified as to work on $\widehat{x}$ and $\widehat{x}'$ instead of x and x'. Furthermore, for our comparison we need to be able to relate the copy of the state variables with the original version. To this end, we define a relation $R_{\mathit{State},\widehat{\mathit{State}}}$ relating all x's with their $\widehat{x}$'s. R can be seen as some kind of retrieve relation and constitutes the identity between $\mathit{State} : \mathit{Exp}$ and $\widehat{\mathit{State}}$.

Finally, we fix the initialisation of this combined system: it is set to any possible state where $R_{\mathit{State},\widehat{\mathit{State}}}$ is true. We ignore consciously any presetting claimed by *Init* in the Object-Z specification as the set of states where *Init* is true is a subset of the set of states where R is true. On this SAL specification the following two CTL formulae can be used for checking equivalence of old and new method:

applicability check:

$$EX(ev = \mathit{Op}_j) \Leftrightarrow EX(ev = \widehat{\mathit{Op}_j})$$

correctness check:

$$\begin{gathered} AX(ev = \mathit{Op}_j \Rightarrow EX(ev = \widehat{\mathit{Op}_j} \wedge R_{\mathit{State},\widehat{\mathit{State}}})) \\ \wedge \\ AX(ev = \widehat{\mathit{Op}_j} \Rightarrow EX(ev = \mathit{Op}_j \wedge R_{\mathit{State},\widehat{\mathit{State}}})) \end{gathered}$$

The first formula states that the old operation can be executed in a state if and only if the new operation can be executed in the corresponding copy of the state. The two formulae for correctness state that whenever the old method has been executed thereby modifying the state, an execution of the new method is also possible leading to an equivalent (under R) state, and vice versa with old and new reversed. We get two formulae here (instead of one for refinement), since we want equivalence and not only an implication. Since we now know how to verify our refactorings we will take a closer look at their translation into the SAL language.

3.2.1 Introduce Explaining Variable

Our first refactoring introduces the new variable *diff*.

updateControlLightsIEV

$\Delta(clOk, clHot, clCold)$

$\exists\, diff : \mathbb{Z} \bullet diff = currentTemp - targetTemp$

$\quad diff < -toleranceTemp \Rightarrow$

$\quad\quad (clOk' = \text{false} \wedge clHot' = \text{false} \wedge clCold' = \text{true})$

$\vdots$

The SAL Language supports the quantifier ∃ and ∀ so the translation of this method is straightforward. We solely have to consider that the scope of the quantifier covers the whole predicate. This is ensured by bracketing the whole expression.

```
updateControlLightsIEV:
(EXISTS(diff: Int): diff = currentTemp_N - targetTemp_N AND
((diff <= neg(toleranceTemp_N))
=> (clOk_N' = FALSE AND clHot_N' = FALSE AND clCold_N' = TRUE)) AND
((diff >= toleranceTemp_N)
=> (clOk_N' = FALSE AND clHot_N' = TRUE AND clCold_N' = FALSE)) AND
((0 <= diff) AND (diff <= toleranceTemp_N)
=> (clOk_N' = TRUE AND clHot_N' = FALSE AND clCold_N' = FALSE)) AND
((0 >= diff) AND (diff >= neg(toleranceTemp_N))
=> (clOk_N' = TRUE AND clHot_N' = FALSE AND clCold_N' = FALSE)) AND
(ev' = clUpdateNew))
--> clCold_N' IN {b: BOOLEAN| TRUE};
    clHot_N' IN {b: BOOLEAN| TRUE};
    clOk_N' IN {b: BOOLEAN| TRUE};
    ev' IN {ev: EVENT| TRUE}
```

When we take this specification and use the above scheme for validating equivalence of old and new method, SAL returns a positive answer. Thus this refactoring is behaviour-preserving.

3.2.2 Consolidate Condition by Function

We have defined two new functions, *tempdiff* and *abs*, to simplify our specifiation.

$$\begin{array}{|l} abs : \mathbb{Z} \rightarrow \mathbb{N} \\ \hline x < 0 \Rightarrow abs(x) = -x \\ x \geq 0 \Rightarrow abs(x) = x \end{array} \qquad \begin{array}{|l} tempdiff : \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Z} \\ \hline \forall\, n, m : \mathbb{Z} \bullet tempdiff(m, n) = m - n \end{array}$$

In the beginning we have already seen that functions can be implemented inside a context. The translations turn out to be as follows:

```
abs(i: Int) : NATURAL = IF i < 0 THEN -i ELSE i ENDIF;

tempdiff(n: Int, m: Int) : Int = n - m;
```

By now these functions can be used in the method to update the controllights.

```
updateControlLightsCCF:
((tempdiff(currentTemp_N, targetTemp_N) <= neg(toleranceTemp_N))
=> (clOk_N' = FALSE AND clHot_N' = FALSE AND clCold_N' = TRUE)) AND
...
```

Again, SAL's answer to the equivalence check is positive.

3.2.3 Substitute Algorithm

Next, we try to validate equality between *updateControlLights* and its refactoring *updateControlLightsSAe*. In this case the verification fails and SAL tells us that our correctness theorem is invalid. Regarding a counter-example created by the model checker we find that *updateControlLightsSAe* allows more than one light to be turned on which was not possible before. Thus old and new method are in fact not equivalent. We consequently have to correct our refactoring and replace *updateControlLightSAe* in our module by the translation of *updateControlLightSAo* and this time succeed in proving equivalence.

$$\begin{array}{|l} \hline \textit{updateControlLightsSAo} \\ \Delta(clOk, clHot, clCold) \\ \hline clCold' = currentTemp < targetTemp - 2 \\ clHot' = currentTemp > targetTemp + 2 \\ clOk' = 2 \geq currentTemp - targetTemp \geq -2 \\ \hline \end{array}$$

Even if this is only a small example and the failure might have been found on closer inspection it demonstrates that model checking is an effective way to debug refactorings.

3.2.4 Split Method

Finally, we take a look at the splitting of *heat*. SAL does not supply us with an operator equal to sequential schema composition. Hence we have to find another way of translating it. Keeping in mind the general idea of automating the translation between Z and SAL it makes sense to apply the definition of the composition. According to [22] the composition of two schemas S and T can be described as follows:

$$S \mathbin{\S} T = \exists\, State'' \bullet S[State''/State'] \wedge T[State''/State]$$

Using this definition we can encode the refactoring and verify the equivalence.

It may appear irritating that we consolidate the two methods which we have split during the refactoring. Nevertheless we must not forget that this is only an effect of the translation between Z and SAL. Imagine a tool that creates correct SAL code for any Object-Z specification it will reveal us any failure in an inner method refactoring.

4 Discussion and conclusion

In this paper we investigated the use of a modelchecker for proving behaviour-preservation of refactorings. The class of refactorings we looked at were those changing a single method only. Besides clearing up the specification of methods these refactorings are also often the prerequisite for larger refactorings, for instance `Pull Up Method` or `Introduce Inheritance` which can only be applied when methods of different classes are known to be equivalent. In general, an automatic check of behaviour-preservation of refactorings is thus a useful tool.

Since we were dealing with single methods, and looked at refactorings improving their internal structure only, the question of what behaviour preservation formally means was easy to answer: behaviour preservation was method equivalence (or data refinement in both directions). However, this does not hold for all of Fowler's refactorings. In fact, a large number of refactorings can immediately be seen not to be standard refinements when looking at a single class in isolation. This does for instance apply to all refactorings changing the parameters of methods (which could be found to be a kind of IO-refinement [1]), refactorings splitting methods into a number of others (a non-atomic refinement [8]) or refactorings adding new methods (behavioural subtyping [10]). Unlike the simple method equivalence we studied here, these refactorings have an influence on other classes: the interface of the class changes and as a consequence all other classes using these methods have to be changed as well. A check for behaviour preservation might thus have to involve larger parts of a specification; what is to be checked then is a *class refinement*, taking one specific class of an Object-Z specification as "main" class and showing equivalence and/or refinement for this class. In the future, we intend to study these type of refactorings and the way of checking their correctness.

As a tool for modelchecking method equivalence we had chosen SAL here. SAL offered itself as a tool since a translation of Z to SAL has recently been given. The choice of using a modelchecker (instead of a theorem prover) for equivalence checking was motivated by the desire to get an automatic correctness check. This, however, limits the applicability of the technique to specifications with finite (and moreover usually small) data domains. Besides these standard restrictions for model checking, we encountered some problems specific to our application. During refactoring, it is common to use methods within other methods. SAL does, however, not allow the use of transition names inside definitions of other transitions. Thus, most of the time these definitions have to be expanded during the translation to SAL and thereby the refactoring is lost. This also applies to some refactorings changing data types: different Z types might be mapped on the same SAL types and thus a refactoring might already disappear during the translation.

As future work we also intend to make some experiments with using Alloy for checking behaviour-preservation of refactorings. Alloy is closer to Z than SAL and might thus open new possibilities for checking equivalence. Bolton [2] uses Alloy for verifying refinements, her work could be taken as the starting point for checking equivalence.

Acknowledgement

We thank Graeme Smith and John Derrick for supplying us with SAL libraries for Z's mathematical tool kit. This work was partially funded by the German Research Council DFG under grant WE 2290/6-1.

References

[1] E. A. Boiten and J. Derrick. IO-refinement in Z. In A. Evans, D. J. Duke, and T. Clark, editors, *3rd BCS-FACS Northern Formal Methods Workshop*. Springer-Verlag, September 1998. http://www.ewic.org.uk/.

[2] C. Bolton. Using the Alloy Analyzer to verify data refinement in Z. In *Refine 2005*, Electronic Notes in Theoretical Computer Science, 2005.

[3] L. de Mouro, S. Owre, H. Rue, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. Sal 2. In R. Alur and D. Peled, editors, *International Conference on Computer Aided Verification (CAV 2004)*, number 3114 in LNCS, pages 496–500. Springer-Verlag, 2004.

[4] L. de Mouro, S. Owre, and N. Shankar. *The SAL language manual*, 2003.

[5] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. CUP, 1998.

[6] J. Derrick and E. A. Boiten. *Refinement in Z and Object-Z*. Springer-Verlag, 2001.

[7] J. Derrick, S. North, and T. Simons. Issues in implementing a model checker for Z. In *ICFEM 2006*, LNCS. Springer, 2006. to appear.

[8] J. Derrick and H. Wehrheim. Using coupled simulations in non-atomic refinement. In *ZB 2003: Formal Specification and Development in Z and B*, number 2651 in LNCS, pages 127–147. Springer, 2003.

[9] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronisation skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[10] C. Fischer and H. Wehrheim. Behavioural subtyping relations for object-oriented formalisms. In T. Rus, editor, *AMAST 2000: International Conference on Algebraic Methodology And Software Technology*, number 1816, pages 469–483. Springer, 2000.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 2004.

[12] T. McComb. Refactoring Object-Z Specifications. In *Fundamental Approaches to Software Engineering (FASE'04)*, Lecture Notes in Computer Science, pages 69 – 83. Springer, 2004.

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

[14] T. McComb and G. Smith. Refactoring object-oriented specifications: A process for deriving designs. Technical Report SSE-2006-01, School of Information Technology and Electrical Engineering, University of Queensland, Australia, May 2006.

[15] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2), 2004.

[16] W.F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks.* PhD thesis, University of Illinois at Urbana champaign, 1992.

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

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

[19] G. Smith and J. Derrick. Verifying data refinements using a model checker. *Formal Aspects of Computing*, 2006. To appear.

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

[21] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.

[22] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.