Model Checking Object-Z Classes: Some Experiments with FDR

Geoff Kassel and Graeme Smith
Software Verification Research Centre
University of Queensland
Australia

Abstract

This paper investigates model checking Object-Z classes via their translation to the input notation of the CSP model checker FDR. Such a translation must not only be concerned with preserving the semantics of the original specification, but also with how efficiently the resulting specification can be model checked. Hence, the paper investigates alternative translation schemes and compares how efficiently the resulting specifications can be checked.

1 Introduction

Model checking [4] is an automatic technique for proving properties of systems specified in a formal notation. A model checker exhaustively checks the state space of a specified system for a state in which the property does not hold. If no such state is found, the model checker indicates that the property is true of the specified system. If such a state is found, the model checker provides the sequence of steps leading to that state as a counter-example (possibly as one among a set of counter-examples).

Model checkers have been applied extensively in both hardware and protocol verification [10, 1]. However, their application to other systems, including general software systems, has been limited for a number of reasons. One of these is that the state space of such systems is often too large. This can result in the model checker requiring an unacceptable amount of both memory and time, or being unable to handle the specification at all. This is referred to as *state explosion*. Another reason is that the formal notations of existing model checkers support only simple types and type constructors since they are aimed at modelling hardware [9], or event-based notations (*process algebras*) which are not particularly suited to modelling data structures [12].

Rather than take on the significant task of building a model checker for more expressive notations, one approach to the latter problem is to translate from such notations to those of existing model checkers [3, 8]. For such translations to be practically useful, they must, as far as is possible, avoid the problem of state explosion. Hence, the translation must be concerned not only with preserving the semantics of the original specification, but also with how efficiently the resulting specification can be model checked.

Object-Z [17, 5] is an object-oriented extension of the formal specification language Z [19]. It includes, in addition to the constructs of Z, a class construct for defining the behaviour of the objects which comprise a specified system. The relationship between the semantics of Object-Z classes and processes in a language such as CSP [13] was first investigated by Smith [15] and has subsequently formed the basis of a number of integrations of Object-Z and CSP [16, 7, 18]. In this paper, we show how this relationship also provides a basis for translations between Object-Z classes and the input notation of the CSP model checker FDR [12, 6][1].

In Section 2, we introduce the notion of classes in Object-Z and outline their semantics and its relationship to CSP processes. In Section 3, we provide two translation schemes. The first is based on existing work of Fischer and Wehrheim [8] for the integration of Object-Z and CSP called CSP-OZ [7]. The second aims at avoiding the use of certain constructs in the Fischer and Wehrheim approach argued to be inefficient by Mota and Sampaio [11]. In Section 4, we compare the efficiency of the translation schemes via a number of experiments. These reveal that neither of the schemes is more efficient in all cases and we conclude the section by developing and running the experiments on a third translation scheme using aspects of both. In Section 5, we conclude with a discussion of future work.

2 Classes in Object-Z

Object-Z [17, 5] is an object-oriented extension of Z [19], a state-based formal specification language in which system states, initial states and operations are modelled by *schemas* comprising a set of variable declarations constrained by a predicate. A class in Object-Z encapsulates a state schema, and associated initial state schema, with all the operation schemas which may change its variables. As an example, consider the following Object-Z specification of a credit card account [5].

The class has an axiomatic definition of a constant *limit* of type natural number ($\mathbb{N}$) whose value is in the set $\{1000, 2000, 5000\}$. It has a single state variable *balance* of type integer ($\mathbb{Z}$) whose value is constrained to be more than $-limit$. Initially, the value of *balance* is zero and it can be decreased or increased by an input value *amount*? via the operations *withdraw* and *deposit* respectively. The notation $\Delta(balance)$ in these schemas denotes that they may change the value of *balance*. The *balance* can also be reduced to $-limit$ by the operation *withdrawAvail* which outputs (via the output variable *amount*!) the total funds available.

[1] A description of FDR can also be found in Roscoe's book on CSP [13].

CreditCard

$limit : \mathbb{N}$

$limit \in \{1000, 2000, 5000\}$

$balance : \mathbb{Z}$

$balance + limit \geqslant 0$

INIT

$balance = 0$

withdraw

$\Delta(balance)$
$amount? : \mathbb{N}$

$amount? \leqslant balance + limit$
$balance' = balance - amount?$

deposit

$\Delta(balance)$
$amount? : \mathbb{N}$

$balance' = balance + amount?$

withdrawAvail

$\Delta(balance)$
$amount! : \mathbb{N}$

$amount! = balance + limit$
$balance' = -limit$

A semantics of Object-Z classes has been given by Smith [15] in which a class is represented by the set of *histories*, i.e., sequences of states and operations, its objects may undergo. In this semantics, an object may undergo any operation that is *enabled*, i.e., whose predicate can be met. If an operation is not enabled, it is said to be *blocked*, i.e., it cannot occur. This is in contrast to the semantics of Z where operations which are not enabled can occur but with an undefined outcome.

This semantics can be mapped to the *failures* semantics of the process algebra CSP [13]. In the latter semantics, processes are represented by *failures* which comprises a sequence of events the process can undergo together with the set of events which can be refused after this sequence. A mapping between the two semantics in which Object-Z classes are associated with CSP processes and Object-Z operations with CSP events has been presented by Smith [16] in order to integrate Object-Z and CSP.

CSP	**CSP$_M$**	**Explanation**
STOP	*STOP*	Deadlock process
$c \in \Sigma$	*channel c*	Definition of event c
$c.v \in \Sigma$	*channel c : t*	Parameterised event c.v (v in t)
$c \rightarrow p$	$c\ {-}{>}\ p$	Process p prefixed by event c
$c.v \rightarrow p$	$c.v\ {-}{>}\ p$	Process p prefixed by event c.v
$p \Box q$	$p[\,]q$	Simple internal choice
$\Box$ v:t $\bullet$ p	$[\,]v : t \bullet p$	Replicated internal choice
$p \sqcap q$	$p\ \vert\tilde{}\vert\ q$	Simple external choice
$\sqcap$ v:t $\bullet$ p	$\vert\tilde{}\vert\ v : t \bullet p$	Replicated external choice
P = ...	$P = \ldots$	Simple process definition
P(v) = ...	$P(v) = \ldots$	Parameterised process
if b then p else STOP	$b\ \&\ p$	Simple guarded command
if b then p else q	*if b then p else q*	Complex guarded commands

Table 1: A summary of the CSP$_M$ notation, as compared to CSP.

In the following sections, we use this mapping as the basis for translating between Object-Z classes and CSP processes in the input notation of the model checker FDR [12, 6]. The mapping faithfully represents the semantics of Object-Z classes in terms of failures and is, therefore, more appropriate for our needs than the mappings between the semantics of Object-Z and CSP developed for other integrations of the languages [7, 18][2].

3 Model checking Object-Z Classes

As noted previously, Fischer and Wehrheim [8] have developed an approach for translating CSP-OZ [7] for checking with the model checker FDR [12, 6]. This translation provides a method of encoding Object-Z classes in the FDR input notation, a machine-readable dialect of CSP called CSP$_M$ [14, 6] that incorporates a functional language notation similar to Haskell [2]. It is this functional language which gives specifications written in CSP$_M$ the expressive power to represent Object-Z, drawing on the CSP$_M$ implementation of set theoretical and logic concepts to encode Object-Z schemas. A summary of the CSP$_M$ notation as compared to CSP is in Table 1, and similarly, a summary of the CSP$_M$ functional language notation as compared to Object-Z is found in Table 2.

To encode Object-Z classes in CSP$_M$ by this approach, classes are decomposed into their component *aspects* – state and operation definitions. State definitions comprise axiomatic definitions and the state and initial state schemas, and are encoded by set constructions based on their declarations and predicates. Operation definitions are further decomposed into their communications,

[2]In CSP-OZ [7], Object-Z classes are given a non-blocking semantics similar to that of Z. In both CSP-OZ and the work of Smith and Derrick [18], outputs of operations may not be constrained by the environment, in contrast to outputs in Object-Z.

Object-Z	**CSP$_M$**	**Explanation**
t == a | b	$\mathit{datatype}\ t = a \mid b$	Free type
a ∧ b	$a\ \mathit{and}\ b$	Logical and
a ∨ b	$a\ \mathit{or}\ b$	Logical or
a = b	$a == b$	Equality
a ⩽ b	$a <= b$	Less than or equal
a ⩾ b	$a >= b$	Greater than or equal
a .. b	$\{a..b\}$	Ranged set
# s	$\mathit{card}(s)$	Set cardinality
a ∈ b	$\mathit{member}(a, b)$	Set membership
$\{v : t \mid b\}$	$\{v \mid v <\!\!- t, b\}$	Simple set construction
$\{c : s;\ d : t \mid b\}$	$\{(c, d) \mid c <\!\!- s, d <\!\!- t, b\}$	Complex set construction

Table 2: A summary of the CSP$_M$ functional language notation, as compared to Object-Z.

enabling precondition, and a post-state 'effect'. A CSP communication event is also used to represent the occurrence of the operation, parameterised with the values to be communicated. The class aspects are re-composed via a *Semantics* process, which gives the non-blocking semantics of CSP-OZ – an event may occur at any time if it is enabled, but if there is no valid post-state, chaos may occur. Chaos is modelled in CSP via *divergence* – a process which undergoes infinite recursion or an infinite sequence of hidden events.

This approach can be adapted to the blocking semantics of Object-Z by a modification of the *Semantics* process – altered so that an event may not occur unless it is guaranteed to succeed. When no event can occur, deadlock occurs. Due to this modification, the model state-space, as derived by the model checker, is reduced significantly – in some cases, by up to a factor of four. This highlights the significance of the choice of semantics of a specification language with respect to the efficiency of model checking.

To investigate the efficiency of different encodings of Object-Z classes, the following trial encodings of the *CreditCard* class were made – an initial encoding based on the CSP-OZ encoding, altered for Object-Z semantics, and a trial encoding which attempts to provide a more efficient representation of state and operation definitions. These are compared in Section 4.

3.1 The initial encoding

The initial approach chosen to model Object-Z semantics in CSP$_M$ differs from the CSP-OZ encoding approach detailed in [8] only by the different semantics chosen when re-composing class aspects. Hence, the *CreditCard* class is decomposed into its state, communication, precondition, and post-state aspects. These are represented by *state* and *init* set constructors, operation *event* definitions, and *enable* and *effect* clauses. The following demonstrates the encoding of the axiomatic constant and state schema of the *CreditCard* class in the CSP$_M$

functional language (see Table 2).

$$
\begin{aligned}
&\mathit{state} = \{(\mathit{limit}, \mathit{balance}) \mid\\
&\quad \mathit{limit} <\!\!-\mathit{Nats},\\
&\quad \mathit{balance} <\!\!-\mathit{Ints},\\
&\quad \mathit{member}(\mathit{limit}, \{1000, 2000, 5000\}),\\
&\quad \mathit{balance} + \mathit{limit} >= 0\}
\end{aligned}
$$

This set constructor constructs all possible values of the axiomatic constant *limit* and the state variable *balance*, typed respectively by the natural numbers ('*Nats*') and the integers ('*Ints*')[3], and restricted by the predicates of the axiomatic definition and the state schema.

The *Init* schema is derived similarly. A predicate $(\mathit{limit}, \mathit{balance}) <\!\!-\mathit{state}$ ensures that the constraints on the axiomatic constant and state variable are met.

$$
\begin{aligned}
&\mathit{init} = \{(\mathit{limit}, \mathit{balance}) \mid\\
&\quad (\mathit{limit}, \mathit{balance}) <\!\!-\mathit{state},\\
&\quad \mathit{balance} == 0\}
\end{aligned}
$$

Operation communications are encoded next. CSP events are used to represent both communication and the occurence of operations – discrete events represent the occurrence of a similarly named operation, parameterised with values to be communicated [16, 7]. CSP$_M$ requires the prior definition of such events using the reserved word *channel*. The operations *withdraw*, *deposit* and *withdrawAvail* hence produce the following definitions, as each communicates a natural number:

$$
\begin{aligned}
&\mathit{channel}\ \mathit{withdraw} : \mathit{Nats}\\
&\mathit{channel}\ \mathit{deposit} : \mathit{Nats}\\
&\mathit{channel}\ \mathit{withdrawAvail} : \mathit{Nats}
\end{aligned}
$$

To distinguish between input and output communications, each operation has an *event* definition, which matches given inputs and outputs to the parameters of the operation event:

$$
\begin{aligned}
&\mathit{event}(\mathit{withdraw}, \mathit{in}, \mathit{out}) = \mathit{withdraw}.\mathit{in}\\
&\mathit{event}(\mathit{deposit}, \mathit{in}, \mathit{out}) = \mathit{deposit}.\mathit{in}\\
&\mathit{event}(\mathit{withdrawAvail}, \mathit{in}, \mathit{out}) = \mathit{withdrawAvail}.\mathit{out}
\end{aligned}
$$

The types of these inputs and outputs are given as sets so that they can be used later in replicated choice operations (see Table 1):

$$
\begin{aligned}
&\mathit{in}(\mathit{withdraw}) = \mathit{Nats}\\
&\mathit{in}(\mathit{deposit}) = \mathit{Nats}\\
&\mathit{in}(\mathit{withdrawAvail}) = \{\{\}\}\\
&\mathit{out}(\mathit{withdraw}) = \{\{\}\}\\
&\mathit{out}(\mathit{deposit}) = \{\{\}\}\\
&\mathit{out}(\mathit{withdrawAvail}) = \mathit{Nats}
\end{aligned}
$$

[3]The CSP$_M$ definitions of these types are given in Section 4.

The events of the class are similarly made accessible to choice operations by definition of a set of all possible class operations:

$$Ops = \{withdraw, deposit, withdrawAvail\}$$

Due to it having a non-blocking semantics, CSP-OZ has a special construct to enforce blocking of operations if desired. To model this, the CSP-OZ encoding has an *enable* clause for each operation. This clause evaluates to true when the operation is not blocked. For Object-Z, blocking is determined by whether or not the operation can occur and hence the *enable* clause is unnecessary. We give it the default value *true*.

$$\begin{array}{l} enable(withdraw)((limit, balance)) = true \\ enable(deposit)((limit, balance)) = true \\ enable(withdrawAvail)((limit, balance)) = true \end{array}$$

The post-state, or 'effect', of the operation is encoded via a set constructor similar to that of the *Init* schema – in addition to constructing the state values, values for the output communications are constructed, and constraints representing the Δ-list of the operation are added. The set constructor for each operation takes as parameters the current state values and the values of the inputs (denoted '_' when there are no relevant inputs).

```
effect(withdraw)((limit, balance), amount) =
      {({}, (limit', balance')) |
            (limit', balance') <- state,
            amount <= balance + limit,
            balance' == balance − amount,
            limit' == limit}
effect(deposit)((limit, balance), amount) =
      {({}, (limit', balance')) |
            (limit', balance') <- state,
            balance' == balance + amount,
            limit' == limit}
effect(withdrawAvail)((limit, balance), _) =
      {(amount, (limit', balance')) |
            (limit', balance') <- state,
            amount <- Nats,
            amount == balance + limit,
            balance' == −limit,
            limit' == limit}
```

Finally, the aspects of the class are re-composed in the *Semantics* process. For CSP-OZ semantics, this is the following, which uses the divergent process *DIV* (trivially defined as $DIV = DIV$ [8]) to model chaos.

```
Semantics(Ops, in, out, enable, effect, init, event) =
    let Z_PART(s) = [] op : Ops @ enable(op)(s) & []i : in(op) @
                if empty(effect(op)(s, i))then
                    (|~| o : out(op) • event(op, i, o)-> DIV)
                else
                    (|~| (o, s') : effect(op)(s, i) @
                        event(op, i, o)-> Z_PART(s'))
        Z_MAIN = |~| s : init @ Z_PART(s)
    within Z_MAIN
```

This process begins by choosing an initial state s from the set *init* (derived from the initial state schema) and then behaves as the (sub)process $Z\_PART(s)$. This latter process after choosing an enabled operation *op* and input i undergoes the event corresponding to that operation with an internally chosen output o. In the case that a valid post-state exists for *op* for the input i and pre-state s, such a post-state s' is internally chosen and the process behaves like $Z\_PART(s')$. Otherwise, the process diverges.

Following Smith [16], the *Semantics* process can be modified to Object-Z's blocking semantics by replacement of the *then* branch of Z_PART with $STOP$, the deadlock process. This ensures the choice of *op* and i such that a valid post-state exists, if possible. This is due to the CSP law, $P \; \Box \; STOP = P$ [13]. This law means a process $\Box v : t \bullet if \; b \; then \; STOP \; else \; P(v)$ will, whenever possible, choose a value of v from t which does not satisfies b – since a value which does satisfy b will result in the process $STOP$. The alteration also requires the use of the external choice operator for initial states, outputs and post-states.

Given that *not b & p* abbreviates *if b then STOP else p* (see Table 1), we have:

```
OZSemantics(Ops, in, out, enable, effect, init, event) =
    let OZ_PART(s) = [] op : Ops @ enable(op)(s) & []i : in(op) @
                not empty(effect(op)(s, i))&
                    ([](o, s') : effect(op)(s, i) @
                        event(op, i, o)-> OZ_PART(s'))
        OZ_MAIN = []s : init @ OZ_PART(s)
    within OZ_MAIN
```

The encoding is completed by a process which calls the *Semantics* process with the derived class aspects:

```
CreditCard = OZSemantics(Ops, in, out, enable, effect, init, event)
```

3.2 A more efficient encoding?

This initial adaption of the CSP-OZ encoding, while reducing the state-space of the compiled model, and hence some portion of the model checking time, is still somewhat time consuming as a model checking technique. A likely culprit for this time inefficiency, as noted by Mota and Sampaio [11], is the extensive use of set construction to determine valid states of the specification. It is deemed that avoidance of this construct improves the efficiency of model checking Object-Z constructs, and hence a replacement construct – replicated external choice – is trialled as an alternative in the following encoding.

Our use of replicated external choice fills the same role as set construction – a valid value can be selected for a given variable, within given predicate constraints. This is possible because of the CSP law, $P \mathrel{\Box} STOP = P$, which, in this case, ensures a process $\Box v : t \bullet b \mathbin{\&} P(v)$ chooses a value of v from t which satisfies b whenever possible. Note that a similar law does not hold for internal choice and hence this approach is not possible with CSP-OZ due to its non-blocking semantics.

Although replacing set comprehension in this way does not decrease the state-space of a specification, a decrease in average model compilation time is observed. This time is most often the largest time factor in the FDR model checking process, outweighing the actual checking time by orders of magnitude, and hence is worthwhile to reduce.

A consequence of the avoidance of set construction is that decomposition of class aspects into labelled sets is no longer relevant. Instead, a class is encoded as a *OZSemantics*-like process which begins by choosing an initial value for the state. A choice is made for each declared constant and variable in this process definition, and axiomatic definition and state schema constraints are included in-line (conjoined with the logical operator *and*) as in the following:

```
CreditCard =
    -- Axiomatic schema type declarations
    [] limit : Nats @
    -- State schema type declarations
    [] balance : Ints @
        -- Axiomatic schema constraints
        member(limit, {1000, 2000, 5000}) and
        -- State schema constraints
        balance + limit >= 0 and
        -- Init schema constraints
        balance == 0 &
            CreditCardBehaviour(limit, balance)
```

The process *CreditCardBehaviour*(*limit*, *balance*) performs the same role as the *OZ_PART* process of the initial encoding – an operation and its input, output and post-state values are selected, and the corresponding event performed. Each operation is separated into its own 'branch', unfolding the replicated choice on the operation set *Ops* of *OZ_PART*. Operation events can be included in-line because of this unfolding. *enable* clauses are always true and hence are excluded entirely.

The following demonstrates the approach:

```
CreditCardBehaviour(limit, balance) =
    (-- withdraw operation
        -- Input type declarations
        [] amount_in : Nats @
        -- Post-state typing constraints
        [] balance' : Ints @
            -- State schema constraints
            (balance' + limit >= 0) and
            -- withdraw schema constraints
            (amount_in <= balance + limit and
            balance' == balance - amount_in) &
                withdraw.amount_in ->
                CreditCardBehaviour(limit, balance')
    )
[]
    (-- deposit operation
        -- Input type declarations
        [] amount_in : Nats @
        -- Post-state typing constraints
        [] balance' : Ints @
            -- State schema constraints
            (balance' + limit >= 0) and
            -- deposit schema constraints
            (balance' == balance + amount_in) &
                deposit.amount_in ->
                CreditCardBehaviour(limit, balance')
    )
[]
    (-- withdrawAvail operation
        -- Post-state typing constraints
        [] balance' : Ints @
        -- Output type declarations
        [] amount_out : Nats @
            -- State schema constraints
            (balance' + limit >= 0) and
            -- withdrawAvail schema constraints
            (amount_out == balance + limit and
            balance' == -limit) &
                withdrawAvail.amount_out ->
                CreditCardBehaviour(limit, balance')
    )
```

A consequence of not decomposing the class aspects to labelled sets is that explicit inclusion of state aspects – notably state schema constraints – is required in each operation schema encoding. Axiomatic constants need not be explicitly included – their values are fixed in the top-level process, and passed to the class behaviour process. Their definitions are hence made available throughout the class behaviour process.

The above process draws upon the same channel event definitions as the initial CSP-OZ based encoding, and so with inclusion of these definitions, the alternate encoding is complete. The following section compares the time efficiency of this encoding against the initial encoding.

4 Case Studies

Before a comparison of the time efficiency of these encodings can be made, a key limitation in any model checking methodology must be addressed - the *state explosion problem* [4]. This problem occurs when large sets are utilised in classes to be model-checked – as the size of the set increases linearly, the model state-space often increases in a polynomial or even exponential fashion, causing the 'explosion' of states for which this problem is named. This reduces the feasibility of model checking classes utilising these large sets, as model checking time and consumption of processing resources increases in a similar fashion. The restriction of large sets to the minimum deemed necessary to properly 'exercise' the specification is frequently necessary before model checking is feasible.

In the case of the *CreditCard* class, the use of integer and natural number sets cause a state explosion problem, which may be overcome with modification of these types. The expedient of defining upper and lower bounds (*MinInt* and *MaxInt*) for the number sets used allows this specification to be model checked, but has an impact on completeness of verification in the form of the *boundary problem.* This problem occurs when an operation that would otherwise successfully assign a variable a value is not permitted to do so by this typing constraint or 'boundary'. This issue is negligible where values encountered are expected to be within these boundaries, but some further modification of the class is often necessary.

Modification of the constant values in the definition of the axiomatic constant *limit* is also required for the *CreditCard* class, as model checking a class utilising the range of all natural numbers up to these constants is not feasible. Hence, these constants are modified from 1000, 2000, and 5000 to 1, 2 and 5 respectively. The range of integer values of -8 to 8 is deemed to be a suitable minimum range to exercise the full behaviour of the *CreditCard* class with this modified *limit* constant. Hence *MaxInt*, *MinInt*, and the type definitions for the integers ('*Ints*') and the naturals ('*Nats*') are declared as follows:

$nametype\ MaxInt = 8$
$nametype\ MinInt = -8$

$nametype\ Ints = \{MinInt..MaxInt\}$
$nametype\ Nats = \{0..MaxInt\}$

With these modifications, the two encodings of the *CreditCard* class may be feasibly model checked.

	CSP-OZ encoding	**Initial encoding**	**'Efficient' encoding**
Integer range	Time/States	Time/States	Time/States
-8..8	3s/109	3s/36	1s/36
-10..10	6s/131	5s/42	3s/42
-12..12	9s/153	7s/48	4s/48
-14..14	13s/175	10s/54	7s/54
-16..16	26s/197	14s/60	10s/60
-18..18	36s/219	19s/66	14s/66
-20..20	45s/241	25s/72	19s/72
-50..50	550s/571	299s/162	**372s/162**

Table 3: Model checking times (in seconds) and number of states for the *CreditCard* class for increasing upper and lower bounds of the integer type.

4.1 Approach comparison

To examine the behaviour of both encodings under varied state spaces, trials using a series of differing upper and lower bounds on the types *Ints* and *Nats* were conducted. The model checking time (comprising the time for compilation and performing a simple deadlock check) and state space of each encoding were compared to a baseline of a CSP-OZ encoding of the class. These encoding trials were tested on a Sun Sparc Ultra-80 machine with four 450MHz Ultra Sparc II processors and 2GB of RAM, and running the Solaris version of FDR 2.77. The results can be found in Table 3.

The table shows a significant reduction in the state space for the initial and 'efficient' encodings for all integer bounds tested. This reflects the underlying difference in the semantics of classes adopted between these approaches and the CSP-OZ encoding. The initial and 'efficient' encodings are also generally more time efficient with the latter encoding being slightly better than the former. An anomaly occurs, however, at the high end of the trials (as indicated by the time / state pair in bold face in Table 3) where the initial encoding proves more time efficient.

Experiments on encodings of other classes showed similar anomalies. In particular, anomalies appeared earlier in the trials when the types used to define the state variables and constants were larger than those of communicated values. This suggested that the encoding of state in the initial encoding is more efficient than that in the 'efficient' encoding and that the latter's improved efficiency came from its encoding of operations alone. Working from this hypothesis, a new encoding unifying the others was developed.

4.2 A unified encoding

In the unified encoding, the state definitions are encoded in a similar fashion to in the initial encoding. To enable the values of the axiomatic constants to be fixed, as in the 'efficient' encoding, separate set constructors are used for axiomatic constants and state variables. This is demonstrated for the *CreditCard* class below:

```
CreditCardAxiom = {limit |
    limit <- Nats,
    member(limit, {1, 2, 5})}

CreditCardState(limit) = {balance |
    balance <- Ints,
    balance + limit >= 0}

CreditCardInit(limit) = {balance |
    balance <- CreditCardState(limit),
    balance == 0}
```

The approach then more closely follows that of the ‘efficient’ encoding except that choices of constants and state variables are made using the above sets as in the following:

```
CreditCard =
    [] limit : CreditCardAxiom @
    [] balance : CreditCardInit(limit) @
        CreditCardBehaviour(limit, balance)
```

where *CreditCardBehaviour* is:

```
CreditCardBehaviour(limit, balance) =
    (-- withdraw operation
        -- Input type declarations
        [] amount_in : Nats @
        -- Post-state typing constraints
        [] balance' : CreditCardState(limit) @
            -- withdraw schema constraints
            (amount_in <= balance + limit and
            balance' == balance - amount_in) &
                withdraw.amount_in ->
                CreditCardBehaviour(limit, balance')
    )
[]
```

Integer range	**Initial encoding** Time/States	**'Efficient' encoding** Time/States	**Unified encoding** Time/States
-8..8	3s/36	1s/36	1s/36
-10..10	5s/42	3s/42	2s/42
-12..12	7s/48	4s/48	3s/48
-14..14	10s/54	7s/54	4s/54
-16..16	14s/60	10s/60	6s/60
-18..18	19s/66	14s/66	8s/66
-20..20	25s/72	19s/72	11s/72
-50..50	299s/162	372s/162	201s/162

Table 4: Model checking times (in seconds) and number of states for the *CreditCard* class for increasing upper and lower bounds of the integer type.

```
    (-- deposit operation
        -- Input type declarations
        [] amount_in : Nats @
        -- Post-state typing constraints
        [] balance' : CreditCardState(limit) @
            -- deposit schema constraints
            (balance' == balance + amount_in) &
                deposit.amount_in ->
                CreditCardBehaviour(limit, balance')
    )
[]
    (-- withdrawAvail operation
        -- Post-state typing constraints
        [] balance' : CreditCardState(limit) @
        -- Output type declarations
        [] amount_out : Nats @
            -- withdrawAvail schema constraints
            (amount_out == balance + limit and
            balance' == -limit) &
                withdrawAvail.amount_out ->
                CreditCardBehaviour(limit, balance')
    )
```

The results of running the experiments with the unified encoding are compared with those of the other trial encodings in Table 4.

As can be seen from these results, not only has the anomaly been overcome for the integer bounds of -50..50, but additional time efficiency has been obtained for all integer bounds tested. Experiments on other classes have revealed similar results.

5 Conclusion

This paper has investigated several encodings of Object-Z classes in the input notation of the CSP model checker FDR. The efficiencies of the representations, in terms of number of states and time required for compilation and model checking, were compared by running a number of experiments. While this has lead to some insights into how to efficiently encode such classes, the experimental approach adopted in this work is seen only as a first step in developing a more sophisticated scheme for translating Object-Z to FDR.

A more thorough investigation of the algorithms underlying FDR is necessary to justify any general claims of efficiency of the approaches developed. The results of this paper, however, provide hints as to which aspects of these algorithms need be investigated, and hence provide a basis for future work in this direction. In addition, other aspects of Object-Z, most notably notions of inheritance and object instantiation, need to be incorporated into the translation scheme. The feasibility of doing this is currently being investigated.

Acknowledgements

The authors would like to thank Kirsten Winter for her interesting and insightful discussions while collaborating on this work.

References

[1] G. Barrett. Model checking in practice: The T9000 virtual channel processor. *IEEE Transactions on Software Engineering*, 21(2):69–78, 1995.

[2] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 1998.

[3] G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *6th International Conference for Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 331–346. Springer-Verlag, 2000.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[5] R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z.* MacMillan Press Limited, 2000.

[6] FDR2 user manual. `http://www.formal.demon.co.uk/fdr2manual/index.htm`, 1999.

[7] C. Fischer. CSP-OZ - a combination of CSP and Object-Z. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 423–438. Chapman & Hall, 1997.

[8] C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *1st International Conference on Integrated Formal Methods (IFM'99)*, pages 315–334. Springer-Verlag, 1999.

[9] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[10] K.L. McMillan and J. Schwalbe. Formal verification of the Gigamax cache consistency protocol. In N. Suzuki, editor, *Shared Memory Multiprocessing.* MIT Press, 1992.

[11] A. Mota and A. Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. *Science of Computer Programming*, 40:59–96, 2001.

[12] A.W. Roscoe. Model checking CSP. In A.W. Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 353–378. Prentice Hall, 1994.

[13] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[14] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP.* PhD thesis, Programming Research Group, Oxford University, 1998.

[15] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[16] G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME'97)*, volume 1313 of *LNCS*, pages 62–81. Springer-Verlag, 1997.

[17] G. Smith. *The Object-Z Specification Language.* Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[18] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems—an integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2001.

[19] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.