Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 267–280
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Combining Techniques to Reduce State Space and Prove Strong Properties

Dilia E. Rodríguez [1]

Information Directorate
Air Force Research Laboratory
Rome, New York, USA

Abstract

An on-the-fly symmetry reduction technique that exploits the lexicographic order on metarepresentations of Maude terms, and a technique that uses auxiliary data to verify strong properties that are not directly expressible in propositional temporal logic are presented. Both are implemented by simple transformations of rewrite theories. They are applied in the verification of a strong-consistency property of a client-server protocol, a simplification of the Chain-Replication protocol.

Keywords: Verification, symmetry reduction, lexicographic order, formal object-oriented specifications.

1 Introduction

Verifying complex properties of concurrent systems is very challenging. There is the state-explosion problem, where the size of the state space increases exponentially with the number of components. Compounding this problem there is the requirement to express and verify strong, nontrivial properties. To ameliorate these problems we developed two techniques: one for on-the-fly symmetry reduction, and one that uses auxiliary data to support the verification of strong properties that are not directly expressible in propositional linear temporal logic (LTL). While the general approaches underlying these techniques are not new, some novel ways of implementing them exploit features of the rewriting logic [5] language Maude [2,3], offering some advantages: they are implemented by simple transformations of the specification being analyzed, and they can be easily combined.

Symmetry reduction is achieved by transforming the specification so that every state in a computation is a canonical representative of an equivalence class. This representative is obtained by exploiting the reflective nature of Maude, which provides

[1] Partially supported by a 2007 AFOSR/RI Minigrant.

1571-0661/Published by Elsevier B.V.
doi:10.1016/j.entcs.2009.05.024

a built-in total order on metarepresentations of terms, and permits the manipulation of these metarepresentations. The technique presented constructs the least element of the orbit of a state with respect to the lexicographic order on metarepresentations of terms. Since the technique transforms the specification, space reductions are obtained in execution, searches and model checking of the specification.

Section 2 introduces Maude preliminaries, in particular those related to object-based specifications. The specification of the simplication of the Chain-Replication protocol is presented in Section 3. Next, Section 4 discusses the on-the-fly symmetry reduction technique and its application to the protocol. Section 5 motivates and describes a technique that uses auxiliary data to verify the strong-consistency property this protocol should satisfy. Conclusions follow in Section 6.

2 Maude Preliminaries

Maude [3][2] is an executable language based on rewriting logic [5], a logic of concurrent change. In rewriting logic, a concurrent system is specified by a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory with the signature Σ specifying sorts (types) and operations; E, a set of equations on Σ-terms; and R, a set of labelled conditional rewrite rules, of the form $l : t \longrightarrow t'$ if *cond*. The equational theory describes the distributed structure of the system, while the conditional rewrite rules define its basic concurrent transitions.

A rewrite theory corresponds to a system module in Maude. For system modules that satisfy some admissibility requirements [2], rewriting with rules is performed modulo the equations of the module. This means that the state space is represented by equivalence classes of states, and only rewrite rules contribute to the size of the state space.

Maude supports object-based models, with a predefined module declaring sorts for the essential concepts, namely `Object`, `Msg`, and `Configuration`.

```
mod CONFIGURATION is sorts Object Msg Configuration .    ...
```

A configuration is a multiset of messages and objects. In particular, a single message or a single object is a configuration. Maude supports subsorts, so this can be expressed as follows:

```
subsort Object Msg < Configuration .
```

A configuration is described by a term of sort `Configuration` constructed with the following operators:

```
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration
        [ctor config assoc comm id: none] .
```

The first takes no arguments, and represents a configuration with neither objects nor messages. The second takes two arguments, which are juxtaposed (the _ is a placeholder, and there is no syntax between the arguments), and is declared with attributes of a multiset: it is associative, commutative, and has identity `none`.

A typical configuration has the form $O_1 \ldots O_n\ M_1 \ldots M_m$, where the O's represent objects and the M's messages. The most general form of a conditional rewrite

rule for an object-based model is of the form:

$$l : M_1 \dots M_m \; O_1 \dots O_n \to O'_{i_1} \dots O'_{i_k} \;\; Q_1 \dots Q_p \;\; M'_1 \dots M'_q \;\; \textit{if } C$$

This rule, labelled by l, represents transitions in which, if the condition C holds for the configuration on the left side of the rule, messages $M_1 \dots M_m$ are consumed; the states of some of the objects $O_1 \dots O_n$ change, becoming $O'_{i_1} \dots O'_{i_k}$, $k \leq n$, with the rest disappearing; and new objects $Q_1 \dots Q_p$ and messages $M'_1 \dots M'_q$ are created.

3 A Client-Server Protocol

The client-server protocol studied is a simplification of the Chain-Replication protocol developed by van Renesse and Schneider [6]. Their protocol has m servers and n clients, but from the perspective of a client there is a single server. The innovation of the protocol is in achieving fault tolerance and high throughput through the collective service provided by the servers, but the state-explosion problem is present in configurations with a single server and several clients. The simplified version of the Chain Replication protocol is used to demonstrate the state-space reduction techniques, and to define (and check in its limiting case) the property the Chain-Replication protocol should satisfy.

In this protocol the server stores an object, whose value a client may observe by making queries, or change by requesting updates. Informally, the property this protocol must satisfy is that any response to a query by a client must reflect prior updates.

An object-based model of this protocol has servers and clients as objects, and requests and replies as messages.

```
sorts Client Server .    subsorts Client Server < Object .
sorts Request Reply .    subsorts Request Reply < Msg .
```

A client is represented using the following operator:

```
op < client_ | request-count :_,  outstanding :_,  value :_>
   : NzNat Nat Bool Value -> Client [ctor] .
```

A nonzero natural serves to identify a client, and a request count is used to limit the number of requests a client can make, ensuring that the state space remains finite. Each client keeps the value of the object, as it has observed it through requests to the server. The protocol stipulates that a client may have at most one outstanding request. For the purposes of this study it is not useful to consider failures or messages lost, and so as long as there is an outstanding request the client may not issue another. Boolean attribute `outstanding` indicates whether the client has initiated a request for which it is expecting a reply.

A request instructs the server to perform an operation on the object: a query is a read operation, while an update is a write operation.

```
op c_ : NzNat -> Oid .
op query[_] : Oid -> Request [ctor] .
op update[_:_] : Oid Value -> Request [ctor] .
```

The action of a client sending a query is represented by the following conditional rule.

```
crl [send-query]
    < client N | request-count : K,   outstanding : false,  value : V >
 => < client N | request-count : s K, outstanding : true,   value : V >
    query[c N]    if K < lim .
```

Similarly, a client sending an update is represented by the following rule:

```
crl [send-update]
    < client N | request-count : K,   outstanding : false,  value : V >
 => < client N | request-count : s K, outstanding : true,   value : V >
    update[c N : val(N, s K)]    if K < lim .
```

Either request may be made only if there is no outstanding request, that is, if `outstanding` is false. If the request is a query, it is represented symbolically by `query[c N]`, which identifies the requesting client. If the request is an update, it must include the value the client is submitting. This is represented symbolically as `val(N, K)`, which indicates that this is the value client `N` submitted in its `K`-th request.

How requests or replies are transported from sender to receiver is not determined by the protocol, and so a term of sort `Msg` in the configuration represents a message that has been sent but not received.

The server receives and processes requests. It is represented using the following operator:

```
op < server | pending :_,  value :_> : RequestQueue Value -> Server
[ctor] .
```

It receives a request by removing it from the configuration and enqueueing it in the `pending` queue.

```
rl [get-request]
    < server | pending : Q,     value : V >  R
 => < server | pending : Q ; R, value : V > .
```

As the server processes a request of a client, it sends a reply confirming the operation. It replies to a query with the current value of the object; and to an update with the value the client had requested be assigned to the object, which is now the current value. So a reply has the following syntax.

```
op reply-to[_:_] : Oid Value -> Reply [ctor] .
```

where the first argument identifies the client to which it is addressed.

The act of the server processing a request and replying is represented by a single rule. Processing a query preserves the value of the object.

```
rl [respond-to-query]
    < server | value : V, pending : query[c N] ; Q >
 => < server | value : V, pending : Q >   reply-to[c N : V] .
```

Processing an update may change it.

```
rl [respond-to-update]
    < server | value : V', pending : update[c N : V] ; Q >
 => < server | value : V,  pending : Q >   reply-to[c N : V] .
```

A reply in the configuration represents a message in transit from the server to some client. A client receives a reply by the following rule.

```
rl  [get-reply] :
    reply-to[c N : V' ]
    < client N | request-count : K, outstanding : true, value : V >
 => < client N | request-count : K, outstanding : false, value : V'
```

```
> > .
```

The attribute `outstanding` becomes false, since receiving the reply concludes the operation.

Thus, the state of this system consists of one server, one or more clients, and possibly various requests and replies. This configuration is enclosed within delimeters as follows:

```
sort TConfiguration .
op {_} : Configuration -> TConfiguration [ctor] .
```

representing the state as a term of sort `TConfiguration`.

3.1 Experiments

A series of experiments shows the costs and effectiveness of the techniques developed in this study. As explained in Section 5, though the specification presented here describes the client-server protocol, it does not support the verification of its correctness. Let us call this specification minimal, and the one presented in Section 5 verifiable. To analyze a protocol using methods that explore the state space requires that the protocol be instantiated. Two parameters characterize an instantiation of the client-server protocol just described: `size`, the number of clients; and `lim`, the number of requests a client may make. The server and all clients are initialized with a special value. Experiments with the minimal specification appear in Section 4, and with the verifiable specification in Section 5. All experiments were performed on a 2.2 GHz Core 2 Duo laptop with 3.5 GB of RAM, running Linux.

The size of the state space was determined using the `search` command, which is part of the Maude system and allows one to explore the state space in a variety of ways (see [2]). Through arguments and various forms, it may return all states (for finite state spaces), or all states satisfying some property, or the first n states it finds, for a specified n. The result of the command includes the number of states examined in obtaining the result.

4 Symmetry Reduction

Many distributed systems include identical components. Thus, if one such component reaches a particular state in one of the possible behaviors of the system, an identical component would reach the same state in a similar behavior. This section first presents mathematical preliminaries for exploiting symmetry. (For a more complete presentation see [1][4].) Then it describes how to implement on-the-fly symmetry reduction for system with identical components.

4.1 Mathematical Preliminaries

A transition system is a pair $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ where A is a set of states and $\rightarrow_{\mathcal{A}} \subseteq A \times A$ is a binary relation called the transition relation. A permutation π on a finite set A is a function $\pi : A \rightarrow A$ that is one-to-one and onto. It is an automorphism on $\mathcal{A}$ if it is such that for all $a, a' \in A$, $a \rightarrow_{\mathcal{A}} a'$ if and only if $\pi a \rightarrow_{\mathcal{A}} \pi a'$. Given

an automorphism π on $\mathcal{A}$, $a_0, \ldots, a_n$ is a path in $\mathcal{A}$ if and only if $\pi a_0, \ldots, \pi a_n$ is a path in $\mathcal{A}$. Any set of automorphisms on $\mathcal{A}$ closed under composition and the inverse operation is a group. An automorphism group G on $\mathcal{A}$ induces a relation $\simeq_G$: $A \times A$ such that $a \simeq_G a'$ if and only if there exists an automorphism $\pi \in G$ such that $a = \pi a'$. A congruence on $\mathcal{A}$ is an equivalence relation $\approx$ where for all $a_1, a_2 \in A$ such that $a_1 \approx a_2$, if there exists $a_1' \in A$ such that $a_1 \rightarrow_{\mathcal{A}} a_1'$, then there is $a_2' \in A$ such that $a_1' \approx a_2'$ and $a_2 \rightarrow_{\mathcal{A}} a_2'$. The relation $\simeq_G$ is a congruence on $\mathcal{A}$.

Let $[a]$ denote the class of states equivalent to a. For any $a, a' \in A$, if $a \rightarrow_{\mathcal{A}} a'$, then for all $a_1 \in [a]$ there exists $a_1' \in [a']$ such that $a_1 \rightarrow_{\mathcal{A}} a_1'$. A quotient transition system $\mathcal{A}_G = (A_G, \rightarrow_{\mathcal{A}_G})$ of transition system $\mathcal{A}$ with respect to a permutation group G is defined by $A_G = \{[a] \mid a \in A\}$ and $\rightarrow_{\mathcal{A}_G} = \{[a] \rightarrow_{\mathcal{A}_G} [a'] \mid a \rightarrow_{\mathcal{A}} a'\}$. Given an equivalence class $[a]$ in $\mathcal{A}_G$, some $*a \in [a]$ may be chosen to represent $[a]$, and a quotient representative system $\mathcal{A}_{G^*} = (A_{G^*}, \rightarrow_{\mathcal{A}_{G^*}})$ may be defined by $A_{G^*} = \{*a \mid [a] \in A_G\}$ and $\rightarrow_{A_{G^*}} = \{*a \rightarrow_{A_{G^*}} *a' \mid [a] \rightarrow_{\mathcal{A}_G} [a']\}$. Then a is reachable from a_0 in $\mathcal{A}$ if and only if $*a$ is reachable from $*a_0$ in $\mathcal{A}_{G^*}$.

4.2 States, Indexed Objects and Automorphisms

In the client-server protocol clients are specified uniformly as indexed objects, and indices are further used to represent values symbolically. Thus, the states of the system may be expressed as functions on indices. More generally, a state with n identical components that are identified by indices in a finite set I may be described by a function $s : I^n \rightarrow A$, where A is a set of states. A permutation $\pi_I : I \rightarrow I$ induces a permutation $\pi : A \rightarrow A$ defined by $\pi\, s(i_1, \ldots, i_n) = s(\pi_I i_1, \ldots, \pi_I i_n)$. The question now is which permutations on states induced by permutations on indices are automorphisms.

To determine this consider the effect of permuting indices on each of the rules of the specification. A close examination of all the rules of the client-server protocol shows that whether a rule is enabled is independent of the values of indices. Thus, for the client-server protocol all permutations on indices induce permutations on states that are automorphisms.

4.3 Lexicographic Order and Symmetry Reduction

To analyze a system using state-space exploration methods the parameters of the specification must be instantiated. In particular, a system with identical components must be instantiated with a fixed number of such components, using indices to differentiate among them. As seen above, the set of all permutations of these indices induces a group $\mathcal{G}$ of permutations on states that are automorphisms. The equivalence class with respect to $\mathcal{G}$ for state s, called the orbit of s, is $[s] = \{\pi s \mid \pi \in \mathcal{G}\}$, and some $*s \in [s]$ is selected as its representative. Here we consider indices and indexed states that are lexicographically ordered, and in the next section we describe an algorithm to construct the least element of a lexicographically ordered orbit of states.

Let $\mathcal{I}$ be a non-empty finite set $\{i_1, \ldots, i_n\}$, and let $\preceq_{\mathcal{I}}$ be a relation on $\mathcal{I}$, such

that $i_1 \prec_{\mathcal{I}} \cdots \prec_{\mathcal{I}} i_n$. A lexicographic order on sequences of elements of $\mathcal{I}$ is defined as

$$(x_1, \ldots, x_m) \prec_{lex} (y_1, \ldots, y_m) \iff \exists k \geq 1. \forall l < k. x_l \sim_{\mathcal{I}} y_l \wedge x_k \prec_{\mathcal{I}} y_k$$
$$(x_1, \ldots, x_m) \sim_{lex} (y_1, \ldots, y_m) \iff \forall k \geq 1. x_k \sim_{\mathcal{I}} y_k.$$

Proposition 4.1 *For any permutation π on $\{i_1, \ldots, i_n\}$,*

$$(i_1, \ldots, i_n) \preceq_{lex} (\pi i_1, \ldots, \pi i_n).$$

Terms also can be lexicographically ordered. Let $(\mathcal{F}, \preceq_{\mathcal{F}})$ be a preorder of function symbols, with constants c and d, and symbols f and g with arities $m \geq 1, n \geq 1$, respectively. Furthermore, let $s, t, s_1, \ldots, s_m, t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$, the set of ground terms built with symbols in $\mathcal{F}$. Lexicographic equivalence is defined as

$$c \sim_{lex} d \iff c \sim_{\mathcal{F}} d$$
$$f(s_1, \ldots, s_m) \sim_{lex} g(t_1, \ldots, t_n) \iff m = n \wedge f \sim_{\mathcal{F}} g \wedge \forall 1 \leq i \leq m. s_i \sim_{lex} t_i.$$

The strict part of the lexicographic relation is defined as follows:

$$i.\ c \prec_{lex} d \iff c \prec_{\mathcal{F}} d$$
$$ii.\ f(s_1, \ldots, s_m) \prec_{lex} c \iff f \prec_{\mathcal{F}} c$$
$$iii.\ c \prec_{lex} f(s_1, \ldots, s_m) \iff c \overset{\prec}{\sim}_{\mathcal{F}} f$$
$$iv.\ f(s_1, \ldots, s_m) \prec_{lex} g(t_1, \ldots, t_n)$$
$$\iff f \prec_{\mathcal{F}} g \vee [f \sim_{\mathcal{F}} g \wedge (s_1, \ldots, s_m) \prec_{lex} (t_1, \ldots, t_n)]$$
$$v.\ () \prec_{lex} (s_1, \ldots, s_m)$$
$$vi.\ (s_1, \ldots, s_m) \not\prec_{lex} ()$$
$$vii.\ (s_1, \ldots, s_m) \prec_{lex} (t_1, \ldots, t_n)$$
$$\iff s_1 \prec_{lex} t_1 \vee [s_1 \sim_{lex} t_1 \wedge (s_2, \ldots, s_m) \prec_{lex} (t_2, \ldots, t_n)]$$

When lexicographically ordering the elements of the orbit of an indexed term, the terms being compared differ only at corresponding positions of index subterms. Only case i and simpler versions of cases iv and vii apply.

$$iv'.\ f(s_1, \ldots, s_n) \prec_{lex} f(t_1, \ldots, t_n)$$
$$\iff (s_1, \ldots, s_n) \prec_{lex} (t_1, \ldots, t_n)$$
$$vii'.\ (s_1, \ldots, s_n) \prec_{lex} (t_1, \ldots, t_n)$$
$$\iff s_1 \prec_{lex} t_1 \vee [s_1 \sim_{lex} t_1 \wedge (s_2, \ldots, s_n) \prec_{lex} (t_2, \ldots, t_n)]$$

In comparing terms root positions are considered before subterms, and subterms to the left are considered before subterms to the right. Thus, determining whether two terms are lexicographically ordered requires that the terms be traversed in preorder until the order can be determined.

Now consider an indexed term t with indices in $\{i_1, \ldots, i_n\}$. Let $(j_1, \ldots, j_n)$ be the tuple of distinct indices in the order they are encountered in a preorder traversal of t. Then from the Proposition 4.1, and the definition of $\preceq_{lex}$ on terms it follows that the permutation $\pi = \{j_k \mapsto i_k \,|\, 1 \leq k \leq n\}$ obtains the least element of the orbit of t. The next section develops this idea into an algorithm to compute the least element of the orbit of a lexicographically ordered orbit a term.

4.4 An Algorithm for Symmetry Reduction

The straightforward computation of a representative of a state indexed over $\{i_1, \ldots, i_n\}$ requires the application of $n!$ permutations to the state. The algorithm presented here exploits the lexicographic order on terms to obtain the least element of the orbit of a state with respect to this order more efficiently. Rewriting logic and Maude are reflective. Every Maude term can be metarepresented as an element of a data type `Term`, and Maude provides functions to convert between representations at different reflection levels. The algorithm described here computes the least element of the orbit of a `Term`.

We introduce sorts `Permutation` and `Permutation?` to represent permutations and sets of permutations, respectively. A predicate `is-index` defines some index-terms, and a permutation is a bijection on these terms.

```
sort IndexPair .
op (_|->_) : Term Term -> IndexPair [ctor] .

sort Permutation .
subsort IndexPair < Permutation .

op emptyPermutation : -> Permutation .
op __ : Permutation Permutation -> Permutation
        [ctor assoc comm id: emptyPermutation] .

op _(_) : Permutation Term -> Term .

op is-index : Term -> Bool .
```

The application of a permutation to constants and indices is defined as usual.

The sort `Permutation?` implicitly represents sets of permutations, or alternatively represents partially defined permutations.

```
sort Permutation? .
subsort Permutation < Permutation? .

op <_‘,_‘,_> : Permutation Nat NzNat -> Permutation? [ctor] .
op _(_) : Permutation? Term -> Term .

eq < P, N, N > = P .

ceq < P, M, N >(I) = P(I) if is-index(I) .
```

The term `< P, m, n >`, where `m < n`, represents a partially defined permutation of a set of (metarepresentations of) indices $\{i_1, \ldots, i_n\}$, or alternatively the set of all permutations that extend `P`. Thus, the set of all permutations on $\{i_1, \ldots, i_n\}$ is represented by `< emptyPermutation, 0, n >`.

We simultaneously construct and apply the permutation that obtains the least element of the lexicographically ordered orbit of a `Term`.

```
var  Gs : TermList .   var  Q : Permutation? .

op [_‘,_] : Permutation? TermList -> PermTerms [ctor] .
op {_‘,_} : Permutation? TermList -> PermTerms [ctor] .
op p_ : PermTerms -> Permutation? .
op t_ : PermTerms -> TermList .

eq p{ Q, Gs } = Q .    eq t{ Q, Gs } = Gs .
eq p[ Q, Gs ] = Q .    eq t[ Q, Gs ] = Gs .
```

We want to define an algorithm that evaluates $[\langle emptyPermutation, 0, n\rangle, T]$ to $\{\pi, \pi(T)\}$, where $\pi = \{j_k \mapsto i_k \,|\, 1 \leq k \leq n\}$ and $(j_1, \ldots, j_n)$ are the n distinct indices in the order they are encountered in a preorder traversal of T.

As new indices are encountered the permutation is further defined

```
op _<+>_ : Permutation? Term -> Permutation? .

eq  < P, M, N > <+> I
  = < ( P (I |-> n-term(M + 1)) ), M + 1, N > .

ceq [Q, I] = {Q, Q(I)}  if is-index(I) and-then I in Q .

ceq [Q, I] = {Q <+> I, (Q <+> I)(I)}
if is-index(I) and-then not(I in Q) .
```

The term `n-term(`m`)` is the metarepresentation of the m^{th} index in the chain of indices $i_1 \prec_{\mathcal{I}} \cdots \prec_{\mathcal{I}} i_n$. Note that if $[Q, I]$ evaluates to $\{Q', J\}$, and $Q(K)$ is defined for some index K, then $Q'(K) = Q(K)$. Furthermore, note that as a `Permutation? Q` is extended by the operator `_<+>_` index `I` is assigned an index that is greater than any element of the range of `Q`.

Constants remain unchanged by a `Permutation?`, and vice versa.

```
eq [ Q, C ] = { Q, C } .
```

Furthermore, the general definition may have to be specialized for specifications in which the same term that represents an index in some context is not an index in another. In the specification of the client-server protocol nonzero naturals are used to identify the clients, but naturals are used as counters to symbolically represent values. So a term that is a counter should be unchanged by permutations.

```
op special : Qid -> Bool .
eq special(R)
  = (R == 'val) or-else
    (R == '<‘client_|‘request-count‘:_‘,
                    outstanding‘:_‘,value‘:_>) .

eq [ Q, 'val [I, K] ]
  = { p[ Q, I ], 'val[ t[ Q, I ], K ] } .
```

Here the `Permutation? Q` is applied to the index `I`, but not to the counter `K`. The `Permutation? Q` might be further defined as it encounters `I`. As noted above, if $Q(J)$ is defined for some index J, then $p[\,Q, I\,](J) = Q(J)$, and so $p[\,Q,'\,val[I, K]\,](J) = Q(J)$.

In a preorder traversal of the `Term` representing a client object, that is, `'<‘client_|‘request-count‘:_‘,outstanding‘:_‘,value‘:_>[I, K, B,V]`, the term `I` is visited before the rest of the terms in the list, in particular, the term `V`. Since as the term is traversed the partial permutation may be further defined, the `Permutation?` resulting from the application to the index I is applied to the value V.

parameters		*no reduction*			*symmetry reduction*					
size	*lim*	*states*	*time*	*mem*	*states*	% Δ	*time*	% Δ	*mem*	% Δ
2	5	38,029	.8 s	28MB	19,295	-49	8.5 s	+962	40MB	+42
3	2	72,063	1.8 s	52MB	13,280	-82	10.9 s	+730	34MB	-35
3	3	952,747	28.5 s	621MB	174,428	-81	161 s	+565	301MB	-52
3	4	*aborted*			1,126,845	NA	19 m	NA	1.9GB	NA
4	2	*aborted*			356,379	NA	9 m	NA	718MB	NA

Table 1
Experiments with the minimal specification.

```
eq [ Q, '<'client_|'request-count':_',outstanding':_',value':_>
                [I, K, B, V] ]
    = { p[ p[ Q, I ], V ],
       '<'client_|'request-count':_',outstanding':_',value':_>
            [ t[ Q, I ], K, B, t[ p[ Q, I ], V ] ]  } .
```

For a general `Term` the definition is as follows:

```
ceq [ Q, R [ Ts ] ]
  = { p[ Q, Ts ], R [ t[ Q, Ts ] ] }
if not special(R) .

eq [ Q, (T, Ts) ]
 = { p[ p[ Q, T ], Ts ], ( t[ Q, T ], t[ p[ Q, T ], Ts ]) } .
```

The approach taken here to symmetry reduction is to modify a specification $\mathcal{R} = (\Sigma, E, R)$ to $\mathcal{R}' = (\Sigma U \Sigma', EUE', R')$. Σ' and E' include the signature and equations used to describe the above algorithm. In addition, at the object level "markers" are introduced to detect when a transition has occurred.

```
sort Marker .
subsort Marker < Msg .

ops ? ! : -> Marker .
```

Each rule $l \to r$ in R is replaced by a rule `?` $l \to r$ `!`. Before any transition may be enabled, the least element of the orbit of the final state of the last transition is constructed.

```
var C : Configuration .

eq  { ! C }
  = { ? downTerm(t [ < emptyPermutation, 0, size >,
                     upTerm(C)], error) } .
```

Table 1 shows the results of experiments to study the effectiveness of this symmetry-reduction technique. Using the `search` command to look for a state that would exceed the limit on the number of requests a client may make forced the exploration of the entire state space, and provided the total number of states of an instantiation. The results show that for smaller state spaces, the cost in time and memory for the state-space reduction is too high. For the next instantiations (of size 3 and limits 2 and 3) the state-space reductions are accompanied by substantial reductions in memory used, though at the expense of much larger execution times. Finally, for the largest instantiations (size 3, lim 4, and size 4, lim 2) the exploration of the state space is possible only with the symmetry-reduction technique.

5 Strong Consistency

The protocol described in Section 3 is a simplification of the Chain-Replication [6], but it should satisfy the same property as the original protocol. As stated in [6], the protocol guarantees strong consistency, which requires that query and update operations be executed in some sequential order, and that the effects of update operations be reflected in the results returned by subsequent query operations. The specification presented in Section 3 does not permit the verification of this property. It is a minimal specification, in which the state is as simple as can be to describe the protocol. Verification of the strong-consistency property requires a state with more information. This section transforms the specification of Section 3 into one that supports the verification of the strong-consistency property.

This property requires that the response of the server to a client reflect the update operations that have been performed. In the absence of failures, the most basic requirement is that when a server responds to a client, the client eventually receives the response. This can be expressed in linear temporal logic as a formula of the form $\Box(\phi \rightarrow (\Diamond\psi))$. This means that for any path, whenever a state satisfies property ϕ (server sends response) there will be some future state in the path that will satisfy property ψ (client receives response). The correctness property, however, requires a response with the correct information. It might be expressed as a formula of the following form: $\forall X.\forall i.\Box(\phi_i(X) \rightarrow (\Diamond\psi_i(X)))$. Here X is a value the server assigns to the object, and i identifies a client. The predicate $\phi_i(X)$ states that the server replies to client i with value X; while predicate $\psi_i(X)$ states that client i receives X in a reply. No such binding of the variable X, however, is expressible in linear temporal logic.

In fact, the property the client-server protocol should satisfy in all states is concerned not only with the eventual value the client will receive, but also with the value it currently has. Thus, the form of the property is more complex than the one described above, and remains not directly expressible in linear temporal logic. So we transform the specification of Section 3 to be able to verify this property.

It is a property about agreement between the server and each client. The server is the keeper of the value of the object; while a client may request update and query operations. These are not instantaneous, so we define what it means for a client and server to agree on the value of the object.

This protocol allows a client to have at most one outstanding request for an operation. A client initiates the operation by sending a request, marked by the attribute `outstanding` becoming true. The server eventually receives it, processes it, and replies to the client. When the client receives the response its `outstanding` attribute becomes false, and the operation is completed.

With the reception of the reply the client updates its value of the object. This should result in the client agreeing with the value the server had when it processed the last request by this client. This is the condition that should hold whenever the `outstanding` attribute has value false.

While a request is outstanding there are three stages. The first begins when

the request is sent (with `send-query` or `send-update`). The request becomes part of the configuration. The second begins when the `get-request` rule removes this request from the configuration and enqueues it in the `pending` attribute of the server. During these two stages the agreement should still be that the client should have the same value of the object as the server had when the server processed the last request by this client.

The last stage begins when the server processes the request (with the `respond-to-query` or `respond-to-update` rule), and sends the reply. This reply to the client becomes part of the configuration. During this stage the client should have the value the server had when it processed the previous to last request by this client, or if this is the first request by this client, the client should have its initial value of the object.

To be able to determine whether the required agreement holds at all times the specification will have auxiliary data.

```
sort AuxData .    subsort AuxData < Msg .
op [_](_,_) : Oid Value Value -> AuxData .
```

For each client, the values the server had when it processed the last and previous to last requests are kept: `[c I](P, L)`.

The only other change to the original specification is to the rules that process the requested operations:

```
rl  [respond-to-query] :
    [c N](P, V') < server | value : V, pending : query[c N ] ; Q >
 => [c N](V', V) < server | value : V, pending : Q  >  reply-to[c N : V] .

rl  [respond-to-update] :
    [c N](P, V') < server | value : V',pending : update[c N : V] ; Q > >
 => [c N](V', V) < server | value : V, pending : Q >  reply-to[c N : V ] .
```

which now must update the auxiliary data to reflect the value the server had when it processed the last and previous to last operations requested by this particular client.

So in all states a client should have one of the last two values the server had when processing a request by this client. During the third phase of an outstanding request by client `I` the configuration (i.e. state) includes `reply-to[c I : V]` as well as the auxiliary datum `[c I](P, V)`. In any state during this stage client `I` should have value `P`. Otherwise, when there is no reply for client `I`, it should have the last value in `[c I](P, V)`, that is, `V`.

The search command can be used to verify that this property holds for all clients in all states. Simply search for any state that satisfies the negation of the required property. client client? The following search command seeks states that violate the agreement between server and client that was described above.

```
search { init(size) } =>*
{ < client I:NzNat |
      request-count : K:Nat, outstanding : B:Bool, value : V':Value >
  [c I:NzNat]( P:Value, V:Value )  C:Configuration }
such that
   ( (reply-to[c I:NzNat : V:Value] in C:Configuration)
     and (V':Value =/= P:Value) )
or ( (not (reply-to[c I:NzNat : V:Value] in C:Configuration) )
     and (V':Value =/= V:Value) ) .
```

If no such state is found then the instantiation of the protocol that was subjected to this search satisfies the strong-consistency property.

parameters		*no reduction*			*symmetry reduction*					
size	*lim*	*states*	*time*	*mem*	*states*	*% Δ*	*time*	*% Δ*	*mem*	*% Δ*
2	5	109,409	4.2 s	75MB	55721	-49	38 s	+804	101MB	+35
3	2	101,649	5.2 s	76MB	19,153	- 81	22 s	+323	47MB	-38
3	3	3,253,621	254 s	2.3GB	598,593	-82	864 s	+240	1.1GB	-52
3	4	*aborted*			*aborted*					
4	2	*aborted*			671,262	NA	23m	NA	1.5GB	NA

Table 2
Experiments with the verifiable specification.

Instantiations with one server and two clients, and with one server and three clients, were found to be strongly consistent. Table 2 shows the metrics of the experiments. They show a high cost in execution time for the computation of the canonical representatives.

Table 2 shows the results of experiments with the verifiable specification. As with the experiments with the minimal specification, the technique is not useful for smaller state spaces. Then for larger state spaces the reductions in the state space are accompanied by significant memory reductions, but at a significantly increased execution time. Finally, the largest of the instantiations can be verified only with the application of the symmetry-reduction technique.

6 Conclusion

A general on-the-fly symmetry reduction technique was presented. It exploits the lexicographic order on metarepresentations of terms to construct a representative of the orbit of a state by a single traversal of the metarepresentation of the state. The technique is implemented by a simple transformation of a specification. This means that state-space reductions, and concomitant time and memory reductions, are effected in executing, searching and model checking a Maude specification.

A technique that uses auxiliary data to allow the verification of strong and complex properties that are not directly expressible in propositional linear temporal logic. The ability to verify these properties comes at the price of a larger state space. The combination of both techniques allowed the verification of a strong-consistency property for several instantiations of a simplification of the Chain-Replication protocol.

Acknowledgement

My thanks to José Meseguer for discussions on symmetry reduction, and in particular for his idea of using the total order on metarepresentations of terms to select a representative for an orbit of states.

References

[1] Clarke, E. M., O. Grumberg, D. Peled, “Model Checking,” MIT Press, Cambridge, MA, 2000.

[2] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, "All About Maude — A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic," Springer Verlag, 2007.

[3] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada, *Maude: specification and programming in rewriting logic*, Theoretical Computer Science, **285** (2002), 187–243.

[4] Ip, C. N., and D. L. Dill, *Better verification through symmetry*, Formal Methods in System Design. **9(1–2)** (1996), 41–75.

[5] Meseguer, J., *Conditional rewriting logic as a unified model of concurrency*, Theoretical Computer Science, **96(1)** (1992), 73–155.

[6] van Renesse, R., and F. Schneider, *Replication for Supporting High Throughput and Availability*, Proc. of the Sixth Symposium on Operating Systems Design and Implementation, 2004.