State Space c-Reductions of Concurrent Systems in Rewriting Logic*

Alberto Lluch Lafuente[1], José Meseguer[2], and Andrea Vandin[1]

[1] IMT Institute for Advanced Studies Lucca, Italy
[2] University of Illinois in Urbana-Champaign, USA

Abstract. We present *c-reductions*, a simple, flexible and very general state space reduction technique that exploits an equivalence relation on states that is a bisimulation. Reduction is achieved by a *canonizer function*, which maps each state into a not necessarily unique canonical representative of its equivalence class. The approach contains symmetry reduction and name reuse and name abstraction as special cases, and exploits the expressiveness of rewriting logic and its realization in Maude to automate c-reductions and to seamlessly integrate model checking and the discharging of correctness proof obligations. The performance of the approach has been validated over a set of representative case studies.

1 Introduction

Taming state space explosion is one of the key challenges for effective model checking analysis. Bisimulation-based state space reductions are particularly attractive, because they *never generate spurious behaviors*. This is because temporal logic properties are preserved by bisimulations. Therefore, an LTL, CTL, or CTL* formula holds on a bisimilar reduced system iff it holds in the original system. A particular example are *symmetry reductions* which have been extensively studied [1] and are used in model checkers (from the seminal works on Murphi to extensions of Spin, Uppaal, Prism, etc.) and other verification tools such as SAT solvers or planners. Developing and applying such state space reduction techniques is still a challenging task: (i) automatic detection of system regularities like symmetries is not trivial and thus often delegated to the system designer; (ii) their exploitation is sometimes done by enriching the system description language (e.g. *scalarset* datatypes in [2, 3]), so that the user is required to learn new primitives; (iii) the implementation of state space reduction techniques has to be combined (both theoretically and practically) with the rest of the techniques and algorithms implemented in the model checker, and often this integration effort has to be repeated for every new version, improvement or technique; and (iv) checking correctness of the reductions is not easy and requires reasoning techniques (e.g. theorem proving) that may not be integrated in the model checking framework, or part of the user's skills. Indeed, problem

* Work supported by NSF Grant CCF 09-05584, AFOSR Grant FA8750-11-2-0084 and the EU Project ASCENS.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 430–446, 2012.

(iv) means that in order to correctly model check a formula in a reduced system it must be a correct reduction of the original system, which requires discharging *proof obligations*. The problem, however, is that most model checkers lack theorem proving support (within the same framework) for discharging such proof obligations, so that the checking task is usually left to the user and may never be done, decreasing the confidence that can be placed on the verification.

Research Questions. In addressing problems (i)–(iv) above, our work asks and provides answers to the following research questions: (1) Can symmetry reductions be generalized to reductions *requiring only that the bisimulation is an equivalence relation*? (2) Can model checking support for such bisimulation-based reductions be provided in a way that does *not* require any changes to the underlying model checker, yet with high performance? (3) Can the system description language be kept likewise unchanged? (4) Can the specifications of *reduced systems* be automatically generated from those of the original systems? (5) Can model checking and theorem proving be *seamlessly integrated* for such reductions, so that correctness proof obligations are explicitly generated and can be semi-automatically discharged by appropriate tools?

Our Contributions. We answer question (1) in the affirmative by proposing the notion of c-*reduction*, based on the idea of providing a *canonizer function* that computes a not-necessarily unique representative of the equivalence class of states defined by the bisimulation. This notion is quite flexible, since unique canonical representatives, although maximally space-efficient, can be time-inefficient. Furthermore, it is *fully general*: it subsumes various reduction techniques such as symmetry reduction, name reuse and name abstraction; and it can be applied to any Kripke structure. Questions (2) and (3) are answered in the affirmative: no such changes are needed (moreover, in [8] we report on performance experiments showing that c-reductions can achieve drastic state space reductions). Question (5) is answered by proposing rewriting logic [4] as an efficiently executable *logical framework* supported by a high-performance tool (Maude [5]) and having a formal tool environment where both LTL model checking and the discharging of correctness proof obligations for c-reductions are seamlessly integrated and partially automated. In fact, our answer to question (5) takes the form of a *formal methodology*, which breaks proofs of correctness into smaller, manageable proof subtasks. Many of the steps in our methodology apply to any c-reduction, but some of them are directly tailored to symmetry reductions. As we gain more experience, we plan to extend all steps of our methodology to arbitrary c-reductions. Question (4) is answered in our current prototype for a very wide class of concurrent systems, namely, *object-based concurrent systems*, and takes the form of a *theory transformation* that automatically maps the original system into the desired c-reduction of it.

We have evaluated our approach over a set of examples by considering the ease of defining reduction strategies, the effectiveness of the correctness checks, and the performance of the resulting reductions. Compared to previous work, we have observed performance gains in some cases (including previous implementations of symmetry reductions in Maude [6]), and a great flexibility in the

definition of reductions, which allows us to subsume a wide range of reductions including permutation and rotation symmetries, name reuse and name abstraction, which have interesting applications (e.g. implementation of the operational semantics of languages with dynamic features such as resource allocation). The usefulness of our proof methodology has also been evaluated through a case study. A preliminary version of our tool is available for download [7].

Synopsis. Sect. 2 offers the necessary background. Sect. 3 presents c-reductions in a generic way, focusing on Kripke structures. Sect. 4 describes the realization of c-reductions in rewriting logic, highlighting the theoretical results, and the reasoning and verification mechanisms and tools underlying our methodology for specifying and verifying c-reductions. Sect. 5 covers related work and conclusions.[1]

2 Preliminaries

We will use a simple running example of a banking system[2] of concurrent objects of the same class (accounts) having a natural number as attribute (their balance), and body-less messages (one dollar transfers) for them. The behavior of objects is governed by a simple rule: a message m for an object i can be consumed by object i to increment its balance by one. The system exhibits a clear symmetry: all objects are instances of the same class and have the same behaviour.

Systems like this (and of course more sophisticated ones) can be easily specificied as theories of rewriting logic [4], which can be specified as Maude [5] modules to be executed and analyzed within the Maude framework.

Definition 1 (rewrite theory). *A* rewrite theory $\mathcal{M}$ *is a tuple* $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ *where* Σ *is a signature, specifying the basic syntax (function symbols) and type infrastructure (sorts, kinds and subsorting) for terms, i.e. state descriptions;* E *is a set of (possibly conditional) equations, which induce equivalence classes of terms (and are used to specify functions), and (possibly conditional) membership predicates, which refine the typing information;* A *is a set of axioms which also induce equivalence classes of terms, i.e., equational axioms describing structural equivalences between terms, like associativity and commutativity;* R *is a set of (possibly conditional) non-equational rules, which specify the local concurrent transitions in a system whose states are* $E \cup A$*-equivalence classes of ground* Σ*-terms; and where* $\phi : \Sigma \to \mathcal{P}_{fin}(\mathbb{N})$ *is a* frozenness map*, assigning to each function symbol* f *of arity* n *a subset* $\phi(f) \subseteq \{1..n\}$ *of its frozen argument positions, i.e. positions under which rewriting with rules in* R *is forbidden.*

In our example we can define a theory (a Maude module) `BANK` whose signature Σ includes sorts for messages (`Message`), objects (`Object`), their identifiers

[1] Interested readers are referred to [8] which includes complementary material: the formal proofs ([8, Sect. A]), a performance evaluation with literature benchmark ([8, Sect. B]), and a full description of our case study ([8, Sect. C,D]).

[2] Indeed, it is a simplification of the model of a bank account system described in [5].

and attributes as natural numbers (`Nat`), configurations (`Configuration`) and states (`State`), and operators that allow us to represent an object `i` with attribute `x` as a term `< i | x >`, a message for object `i` as a term `credit(i)`, an empty configuration (of objects and messages) by `none`, and the multiset union of configurations by juxtaposition, obeying associativity and commutativity as axioms. The operator {_} wraps an entire configuration `c` as a state {`c`}. Rules `rl { < i | x > credit(i) c1 } => { < i | s(x) > c1}` , and `rl { < i | x > credit(i) } => { < i | s(x) > }` model the above described behavior of objects. Informally, one of these rules applies to states containing an object `< i | x >`, a message `credit(i)` for it, and (possibly) a subconfiguration `c1`[3]. If such a match is found, the state can be replaced by the term on the right-hand side of the rule (after applying the substitution of the match), resulting in a state without the message and where object `i` increments its balance with the successor operator `s`.

For the sake of simplicity, we assume that the system under study is described by a rewrite theory $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ whose rules are "topmost" for a designated kind `[State]` of states. We also assume that an operator {_} is used to enclose states so that all rules in R have that operator as their top operator in their left-hand sides. These assumptions are already quite general: they can cover, for example, object-based concurent systems. We further assume that $\mathcal{M}$ has *good executability properties*, i.e., that E is sufficient complete, (ground) confluent and terminating modulo A (that is, that the equational part correctly defines functions), and R is coherent with E modulo A [5] (that is, that applying equations to evaluate functions does not interfere with the application of the rules that specify system transitions). Moreover, unless we state the contrary, all extensions of $\mathcal{M}$ that we shall define will be required to be ground confluent, ground terminating, and sufficiently complete w.r.t. the same signature of constructors as $\mathcal{M}$. Fortunately, the standard Maude tools offer automatization support for checking such properties. Our running example satisfies all these conditions.

We consider the well-known semantic domain of Kripke structures for rewrite theories, suitable for state space exploration problems like model checking.

Definition 2 (Kripke structure). *A* Kripke structure K *is a tuple* $K = (S, \rightarrow, L, AP)$ *such that* S *is a set of states,* $\rightarrow \subseteq S \times S$ *is a transition relation between states, and* $L : S \rightarrow 2^{AP}$ *is a labelling function mapping states into sets of atomic propositions* AP *(i.e. observations on states).*

The Kripke semantics of a rewrite theory has `State`-sorted terms as states and one-step rewrites between `State`-sorted terms as transitions. The labelling function is defined by Boolean predicates specified equationally in the rewrite theory. As proved in [9], any computable Kripke structure, even an infinite-state one, can be obtained from an executable rewrite theory using only a finite signature Σ, and finite sets E of equations, A of axioms and R of rules.

[3] The second rule is needed since we treat the fact that `none` is an identity for union equationally rather than axiomatically.

Definition 3 (Kripke semantics of rewrite theories). *Let* $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ *be a rewrite theory with a designated state sort* `State`*, and a set* $AP \in \Sigma$ *of Boolean state predicates equationally defined in* E*. The Kripke structure associated to* $\mathcal{M}$ *is* $K_{\mathcal{M}} = (T_{State/E \cup A}, \rightarrow, L, AP)$ *such that* $T_{State/E \cup A}$ *are all* `State`*-sorted states,* $\rightarrow$ *is defined as* $\{[u] \rightarrow [v] \mid \mathcal{M} \vdash u \rightarrow^1_{R,State} v\}$ *(i.e. transitions are one-step rewrites between* $E \cup A$ *equivalence classes of* `State`*-terms in* $\mathcal{M}$*), and* L *is such that* $p \in L(s)$ *iff* $p(s) =_{E \cup A} true$*.*

We will consider bisimulation as the key semantic equivalence.

Definition 4 (bisimulation). *Let* $K = (S_K, \rightarrow_K, L_K, AP_K)$*,* $H = (S_H, \rightarrow_H, L_H, AP_H)$ *be two Kripke structures, and let* $\sim \subseteq S_K \times S_H$ *be a relation between* S_K *and* S_H*. We say that* $\sim$ *is a* bisimulation *between* K *and* H *iff for each two states* $s \in S_K$ *and* $s' \in S_H$ *such that* $s \sim s'$ *we have that: (i)* $L_K(s) = L_H(s')$*; (ii)* $s \rightarrow_K r$ *implies that there is a state* r' *s.t.* $s' \rightarrow_H r'$ *and* $r \sim r'$*; and (iii)* $s' \rightarrow_H r'$ *implies that there is a state* r *s.t.* $s \rightarrow_K r$ *and* $r \sim r'$*.*

The notion of bisimulation can be lifted to rewrite theories in the obvious way. We shall focus on bisimulations such that the relation $\sim$ is an equivalence relation, which includes the case of bisimulations induced by symmetries, i.e. when two states are bisimilar if they belong to the same class of symmetric states.

For instance, suppose that the initial state of our example is {`< 0 | 0 > < 1 | 0 > credit(0) credit(1)`}. We have then two possible transitions (given by the application of the rules governing the system), leading respectively to states: {`< 0 | 1 > < 1 | 0 > credit(1)`} and {`< 0 | 0 > < 1 | 1 > credit(0)`}. These two states are syntactically different but they are *symmetric*, i.e. equal up to the permutation of object identifiers.

Indeed, equivalence classes of symmetric states can be conveniently defined as the *orbits* of a group action (permutations in our example), which yield *symmetry reductions* as a special case of our approach. We hence recall here some basic notions about groups and group actions.

Definition 5 (group basics). *A* group *is a tuple* $G = (G, \bullet, e, (\_)^{-1})$ *where* G *is a set of elements,* $\bullet : G \times G \rightarrow G$ *is a binary associative operation,* $e \in G$ *is an identity (i.e.* $\forall f \in G. f \bullet e = e \bullet f = f$*), and* $(\_)^{-1}$ *is an inverse operator (i.e.* $\forall f \in G. f \bullet f^{-1} = f^{-1} \bullet f = e$*).*

Let G *be a group and* $H \subseteq G$ *be a subset of* G*. The group generated by* H *denoted* $\langle H \rangle$ *is defined as the closure of* H *under the inverse and product operators* $(\_)^{-1}$ *and* $\bullet$ *of* G*. In general* $\langle H \rangle$ *will be a subgroup of* G*, but if* $\langle H \rangle$ *coincides with* G*, then* H *is said to* generate G *and its elements are called* generators*.*

Let G *be a group and* A *be a set. An* action *of* G *on* A *is a monoid homomorphism* $[\![\cdot]\!] : G \rightarrow [A \rightarrow A]$*, that is,* $[\![f \bullet g]\!] = [\![f]\!] \circ [\![g]\!]$*, where* $f \circ g$ *denotes function composition in* $(A \rightarrow A)$*, and* $[\![e]\!] = id_A$*, with* id_A *the identity on* A*.*

Notable examples are permutation and rotation groups, which capture typical symmetries introduced by process replication in concurrent systems. Generators define groups in a concise manner, e.g. transpositions and single rotations for

permutation and rotation groups, respectively. The action of a group on the states of a Kripke structure implicitly defines an equivalence relation.

Definition 6 (equivalence induced by a group action). *Let S be a set of states, G be a group and $[\![\cdot]\!]$ be the action of G on S. Then the equivalence relation $\sim_G$ induced by G on S is defined by: $s \sim_G s' \Leftrightarrow \exists f \in G.[\![f]\!](s) = s'$.*

Group actions can be defined in rewriting logic with equations of the form `[[f]](t) = t'` where `f` denotes a group element (typically a generator) and `t`, `t'` are `State`-sorted terms. For instance, in our running example, the application of object identifier transpositions `i<->j` can be defined (by structural induction) with the equations:

```
 eq [teq1] : [[i<->j]]({c1}) = {[[i<->j]](c1)} .
 eq [teq2] : [[i<->j]](none) = none .
 eq [teq3] : [[i<->j]](c1 c2) = ([[i<->j]](c1)) ([[i<->j]](c2)) .
 eq [teq4] : [[i<->j]](< k | x >) = < [[i<->j]](k) | x > .
 eq [teq5] : [[i<->j]](credit(k)) = credit([[i<->j]](k) ) .
 eq [teq6] : [[i<->j]](i) = j .
ceq [teq7] : [[i<->j]](k) = k if (i != k) /\ (j != k) .
```

For example, the unconditional (`eq`) rule `teq 4` defines the application of a transposition `[[i<->j]]` to an `object (< k | x >)` as the object obtained by transposing its identifier. Equations `teq6` and `teq7` take care of transposing identifiers. A symmetric version of `teq6` is not needed since `_<->_` is commutative. Equation `teq7` is conditional (`ceq`): it applies when `teq6` is not applicable.

3 C-Reductions for Kripke Structures

We introduce the idea of *canonical reductions*, abbreviated c-reductions as a generic means to reduce a Kripke structure K by exploiting some equivalence relation $\sim$ on the states of K which is also a bisimulation on K (i.e. between K and itself). In Sect. 4 we will explain how c-reductions are specified, proved correct, and used for model checking in rewriting logic.

We start by defining canonizer functions, which are used to compute for a given state a (not necessarily unique) canonical representative of its equivalence class, modulo some equivalence relation which is also a bisimulation (e.g. a canonical permutation of the identifiers of processes with identical behavior).

Definition 7 (canonizer functions). *Let $K = (S, \rightarrow, L, AP)$ be a Kripke structure, and let $\sim \subseteq S \times S$ be a an equivalence relation which is a bisimulation on K. A function $\mathtt{c} : S \rightarrow S$ is a $\sim$-canonizer (resp. strong $\sim$-canonizer) iff for each $s \in S$ we have $s \sim \mathtt{c}(s)$ (resp. $s \sim \mathtt{c}(s)$, and $s \sim s' \rightarrow \mathtt{c}(s) = \mathtt{c}(s')$).*

Canonizer functions are used to compute smaller but semantically equivalent (i.e. bisimilar) Kripke structures by applying canonizers after each transition. Strong canonizers provide unique representatives for the equivalence classes of states and, hence, more drastic space reductions. That is, for two different but

equivalent states $s \sim s'$ they provide the same canonical representative (i.e. $\mathtt{c}(s) = \mathtt{c}(s')$). Typical examples of strong canonizers for equivalence classes are functions based on *enumeration* strategies [10] which generate the complete set of states of the equivalence class and then apply some function over it (e.g. based on a total ordering of the states). For instance, in our running example, an enumeration canonizer just generates all states that result from permuting (symmetric) processes in all possible ways and then selects one according to some total order (e.g. the lexicographic order of the description of states). In particular, for a state {`< 0 | 1 > < 1 | 0 > credit(1)`} the enumeration will produce its whole orbit: { {`< 0 | 1 > < 1 | 0 > credit(1)`}, {`< 0 | 0 > < 1 | 1 > credit(0)`}}. Then the canonizer would assign the least state of the set according to some total order, e.g. "identifier first, balance second" which would provide {`< 0 | 0 > < 1 | 1 > credit(0)`} as representative. Canonizers can be obtained in more efficient and smarter ways as shown in Sect. 4.4, e.g. with *local search strategies* [10] that repeatedly apply transpositions until the least state is reached. Instead, a non-strong (or weak) canonizer can provide different representatives for equivalent states. That is, it might be the case that $\mathtt{c}(s) \neq \mathtt{c}(s')$ even though $s \sim s'$. Weak canonizers provide weaker state space reductions, but they often enjoy advantages over strong canonizers: in some cases they are easier to be defined and analyzed, and their computation can be much more efficient in terms of runtime cost. Such heuristic canonizers can be found for instance in [11, 3], where the rough idea is to consider an ordering of the states that only depends on part of the state description. The resulting ordering relation is partial and the representative of a state is computed as one of the least states of the ordering.

The reduction of the state space is obtained by applying the canonizer to states after a transition. This is what we call a *c-reduction*.

Definition 8 (c-reduction of a Kripke structure). *Let $K = (S, \to, L, AP)$ be a Kripke structure, and let $\mathtt{c} : S \to S$ be a $\sim$-canonizer function for some equivalence relation $\sim \in S \times S$ which is a bisimulation on K. We call the Kripke structure $K/\mathtt{c} = (S, (\to; \mathtt{c}), L, AP)$ the* c-reduction *of K, where the composed transition relation $\to; \mathtt{c}$ is defined by $\to; \mathtt{c} = \{(s, \mathtt{c}(r)) \in S^2 \mid s \to r\}$.*

An important result is then that a c-reduction is bisimulation preserving.

Theorem 1 ($\sim$-preservation). *Let $K = (S, \to, L, AP)$ be a Kripke structure, let $\sim$ be an equivalence relation on S that is a bisimulation on K, and let $\mathtt{c}$ be a $\sim$-canonizer function. Then $\sim$ is a bisimulation relation between K and $K/\mathtt{c}$.*

4 Correct c-Reductions in Rewriting Logic

We now describe a methodology for specifying, proving correct, and analyzing c-reductions in rewriting logic. In this methodology, correctness proofs and model checking verification are supported by tools in the Maude formal environment such as the Maude LTL Model Checker [12], Invariant Analyzer [13], Inductive Theorem Prover [14] and Church Rosser and Coherence Checker [15].

We assume that there is some regularity in $\mathcal{M}$ that we try to exploit by defining an equivalence (bisimulation) relation $\sim$ on states to ease the analysis of $\mathcal{M}$. We also assume that the specification $\mathcal{M}$ satisfies the assumptions in Sect. 2 and is conveniently structured (see Fig. 1) into a core equational part ($\mathcal{M}.E$), and its extension with state predicate functions that define the atomic propositions ($\mathcal{M}.AP$) and behavioral rules ($\mathcal{M}.R$). Such modular structure is very natural and easy to achieve, and facilitates our methodology. Fig. 1 schematizes our methodology by identifying the main theories (or modules), their incremental construction via extensions (triple arrows) or refactoring (dashed arrows), and the modules involved in each step (dotted arrows). In particular, our methodology consists in the following steps: (i) specify and verify the equivalence relation $\sim$; when the equivalence $\sim_G$ is induced by a group G, specify the group action that induces $\sim_G$ in a module $\mathcal{M} + G$ and verify that it is indeed a group action (Sect. 4.1) which ensures $\sim_G$ to be an equivalence relation; (ii) verify that $\sim$ preserves the state predicates AP (Sect. 4.2) by analyzing their invariance under an auxiliary theory $\mathcal{M}/G$ that models group actions; (iii) verify that $\sim$ is a bisimulation (Sect. 4.3) by checking a coherence-like property between the rules of $\mathcal{M}.R$ and those of $\mathcal{M}/G$; (iv) define a canonizer c in a module $\mathcal{M} + \mathsf{c}$ and show it to be a $\sim$-canonizer (Sect. 4.4); (v) build the c-reduction $\mathcal{M}/\mathsf{c}$ of $\mathcal{M}$ (Sect. 4.5), and (vi) use $\mathcal{M}/\mathsf{c}$ for model checking analysis purposes. Our methology then ensures that any CTL* property φ holds on $\mathcal{M}/\mathsf{c}$ *if and only if* it holds on $\mathcal{M}$, since $\mathcal{M}/\mathsf{c}$ *has been proved to be a correct* c-reduction of $\mathcal{M}$, and therefore *bisimilar* to $\mathcal{M}$.

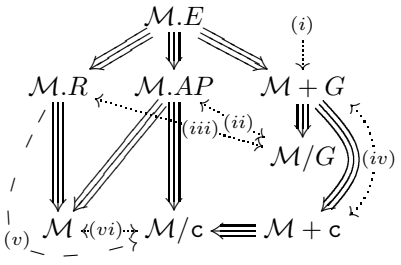


Fig. 1. Modules and steps

Some of the above steps are independent or apply at different levels of abstraction, so that they act as building blocks to be re-used as needed. For instance, verifying a c-reduction strategy does not require performing all the verification steps if it is based on a state equivalence that has been already proven to be correct. In practice, bisimulation relations and their canonizers need not be defined and proven correct for every system, as there will be classes of systems for which they can be specified once and for all. In such cases, one can define c-reductions as theory transformations for wide classes of examples corresponding, for instance, to certain permutation groups, or to other useful equivalence relations besides the symmetry reduction case. In Maude this can be done by exploiting reflection, so that the c-reduction is automatized as a function at the metalevel, possibly after checking some proof obligations. Our current prototype [7] applies some generic c-reductions to any object-based module.

Even though in some of the steps of our methodology we focus on c-reductions based on group actions, the c-reduction technique, in particular steps (v-vi), is more general and allows arbitrary canonizers. We focus on group actions to illustrate the ideas and the semi-automatic correctness checks (steps (i)-(iv))

with a simple example. More substantial examples can be found in [7]; several of them are mentioned, together with detailed performance experiments and comparisons with other tools and methods in [8, Sect. B].

4.1 Specifying and Verifying Group Actions

We give a simple method to equationally specify group actions and verify their correctness *in terms of a set H of generators only, without having to explicitly define the group G* generated by H. The key ideas, explained in detail in [8, Sect. E], consist on: (a) "uncurrying" the desired group action function $[\![\cdot]\!]_{\_} : G \longrightarrow (\mathtt{State} \to \mathtt{State})$ as a function $[\![\cdot]\!]_{\_} : G \times \mathtt{State} \longrightarrow \mathtt{State}$; (b) choosing a subset $H \subseteq G$ that generates G as a monoid; and (c) specifying the inverse $i(g) = g^{-1}$ of each generator $g \in H$ as a product of generators by a function $i : H \to H^*$, where H^* is the free monoid on the alphabet H.

The trick is that, after equationally specifying steps (b) and (c), *G needs not be explicitly defined*: it is enough to specify the action of the generators by a function $[\![\cdot]\!]_{\_} : H \times \mathtt{State} \to \mathtt{State}$, which extends uniquely to a monoid action $[\![\cdot]\!]_{\_} : H^* \times \mathtt{State} \to \mathtt{State}$ satisfying for each $u \in \mathtt{State}$, $g \in H$, $w \in H^*$ the recursive equations: $[\![\epsilon]\!]u = u$ and $[\![wg]\!]u = [\![w]\!]([\![g]\!]u)$. Then it is easy to prove (see [8, Sect. E]) that the only possible group action $[\![\cdot]\!]_{\_} : G \times \mathtt{State} \to \mathtt{State}$ extending $[\![\cdot]\!]_{\_} : H \times \mathtt{State} \to \mathtt{State}$ exists if and only if the following equalities hold for each generator g and each state u: $[\![g]\!]([\![g^{-1}]\!](u)) = [\![g^{-1}]\!]([\![g]\!](u)) = u$. Then G needs not be explicitly specified, because we can safely replace G by the group $H^*/i = H^*/\{g \cdot i(g) = \epsilon \mid g \in H\}$, so that $\sim_G = \sim_{H^*/i}$, and the group action $[\![\cdot]\!]_{\_} : G \times \mathtt{State} \to \mathtt{State}$ can be replaced by the simpler monoid action $[\![\cdot]\!]_{\_} : H^* \times \mathtt{State} \to \mathtt{State}$.

The following definition captures (a) and (b), where we assume that H has been equationally specified by a new sort H, and then H^* has been specified by instantiating a parameterized module $List[X]$ to the instance $List[H]$.

Definition 9 (group pre-action specification). *Let $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ be the rewrite theory under study with designated $\mathtt{State}$ sort. A* group pre-action *on $\mathcal{M}$ is an equational theory $\mathcal{M} + G = (\Sigma \cup \Sigma_G, E \cup E_G \cup A, \emptyset, \phi)$ which is a protecting extension of the equational part of $\mathcal{M}$, $\mathcal{M}.E$, where Σ_G and E_G extend the equational theory $\mathcal{M}.E$ with a sort H, a sort H^* of lists of elements in H (i.e. the module $List[H]$ is protected in $\mathcal{M}+G$), a function $[\![\cdot]\!]_{\_} : H \times \mathtt{State} \to \mathtt{State}$ recursively extended to a monoid action $[\![\cdot]\!]_{\_} : H^* \times \mathtt{State} \to \mathtt{State}$ as explained above, and a function $i : H \to H^*$.*

The *proof obligations* that need to be verified to show that a group pre-action is a group action are as follows:

Proposition 1 (correctness criteria for group actions). *Let $\mathcal{M} + G = (\Sigma \cup \Sigma_G, E \cup E_G \cup A)$ be a group pre-action on $\mathcal{M}$. Then in the inital algebra of $\mathcal{M}+G$ the function $[\![\cdot]\!]_{\_} : H \times \mathtt{State} \to \mathtt{State}$ uniquely extends to a group action of H^*/i on $\mathtt{State}$ if and only if the following two equations hold inductively in such an initial algebra: (i) $(\forall g : H, u : \mathtt{State})\ [\![g]\!]([\![g^{-1}]\!](u)) = u$, and (ii) $(\forall g : H, u : \mathtt{State})\ [\![g^{-1}]\!]([\![g]\!](u)) = u$.*

Using the above implicit definition method and checking the correctness criteria in the above proposition one can equationally define group actions and prove their correctness by inductive equational reasoning. In particular, this can be done for any group action of interest, defining symmetries between states, including the full and rotation symmetries that have been identified and thoroughly studied in the past. Note that sometimes (e.g. transpositions) $i(g) = g$, so that i needs not be defined explicitly, because it is the identity function. The action function $[\![\cdot]\!]_{\_} : H \times \texttt{State} \to \texttt{State}$ can be very easily specified in Maude, by topmost equations relating two `State`-sorted terms of the form `[[g]]({t}) = {t'}`, for (patterns of) elements $\texttt{g} \in H$.

Inductively showing that the equations (i) and (ii) in Proposition 1 are satisfied can usually be done easily by structural induction on the algebraic structure of states. For instance, to check that in our running example full symmetries yield a group action, all we have to do is to prove the equality $[\![i \leftrightarrow j]\!]([\![i \leftrightarrow j]\!](\{t\})) = \{t\}$, i.e. that applying the same transposition of i and j (denoted $i \leftrightarrow j$) twice amounts to applying the identity. This proof can be done by structural induction on `State`-sorted terms. For instance, to show that the property holds in the general case (i.e. `[[i<->j]] ([[i<->j]] ({c1 c2})) = {c1 c2}`), we apply the equations implementing the group action (namely `teq1`, `teq3`) to obtain `{[[i<->j]] ([[i<->j]] (c1)) [[i<->j]] ([[i<->j]] (c2))} = {c1 c2}` and conclude the proof by applying induction.[4]

Once proved that $\mathcal{M} + G$ correctly specifies a group action, we can conclude that the induced relation on states $\sim_G$ is actually an equivalence relation. In our example, we have an equivalence relation induced by object permutations.

4.2 Checking That $\sim$ Preserves Atomic Predicates

To prove that the equivalence $\sim_G$ induced by the action of group G preserves the atomic propositions AP we proceed as follows. First, we define a rewrite theory $\mathcal{M}/G$ for the sole purpose of analysis. The theory $\mathcal{M}/G$ is a protecting extension of $\mathcal{M} + G$ that introduces some rewrite rules to "move" inside orbits.

Definition 10. *Let $\mathcal{M}+G = (\Sigma \cup \Sigma_G, E \cup E_G \cup A, \emptyset, \phi)$ be the theory specifying the action of a group G with generator $H \subseteq G$ on the states of a theory $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$. Then, the theory $\mathcal{M}/G$ is defined as $\mathcal{M}/G = (\Sigma \cup \Sigma_G, E \cup E_G \cup A, R_{\mathcal{M}/G}, \phi)$, where $R_{\mathcal{M}/G} = \{\{\texttt{t}\} \Rightarrow [\![\texttt{g}]\!](\{\texttt{t}\}) \mid \texttt{g} \in H\}$.*

In words, we replace the rules of $\mathcal{M}$ by rules that move from a state u to a state v obtained by applying a generator to u. If H is infinite, $R_{\mathcal{M}/G}$ is also infinite. However, in practice we can often find a finitary reformulation of $R_{\mathcal{M}/G}$, because $R_{\mathcal{M}/G}$ can often be expressed very concisely using patterns for the elements in H. For instance, the module `BANK/PERMUTATION` of our running example contains just two rules: `rl { < i | x > < j | y > } => { [[i<->j]] ( < i | x > < j | y >) }` and `rl { < i | x > < j | y > c1 } => { [[i<->j]] ( < i | x > < j | y > c1) }` to model transitions transposing two arbitrary objects.

[4] For the full proof see [8, Sect. C].

It is easy to see (by the properties of the generators of a group) that two states are reachable in $\mathcal{M}/G$ if and only if they are in the same orbit, i.e. that for any two states u, v we have the equivalence: $u \sim_G v \Leftrightarrow u \rightarrow^*_{R_{\mathcal{M}/G}} v$. Therefore, proving that a predicate $p \in AP$ is preserved by $\sim_G$, i.e. that for each pair of states $u, v \in T_{\mathtt{State}_{E/A}}$ $u \sim_G v$ implies $p(u) = p(v)$, is equivalent to proving that p is *stable* under $\mathcal{M}/G$, i.e. $\mathcal{M}/G \models (p \Rightarrow \Box p)$, where $\Box$ denotes the *always* operator of LTL.

To prove stability we need only to focus on the *positive* equations defining when p holds, which we assume are of the form `p({t}) = true`, or `p({t}) = true if cond`, with `cond` a condition. In our example, the predicate `some-message` characterizing states in which there is at least one message around for some existing object is defined by the equations `eq some-message( < i | x > credit(i)) = true` and `eq some-message( < i | x > credit(i) c1 ) = true`.

Under the assumptions that: (i) the constructors of $\mathcal{M}.E$ are *free modulo the axioms* A, and (ii) the terms `t` in predicate equations `p({t}) = true`, and the left-hand sides of rules in $\mathcal{M}/G$ are constructor terms, we can use the results in [16] to reduce proving $\mathcal{M}/G \models (p \Rightarrow \Box p)$ to the following proof obligations:

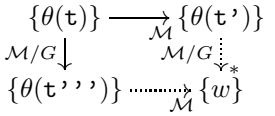
Proposition 2 (predicate preservation through stability). *Let $\mathcal{M}/G$ the auxiliary rewrite theory of Definition 10 and $\mathcal{M}$ satisfy assumptions (i)–(ii) above. and let* `p` *be an atomic proposition defined in $\mathcal{M}.AP$ by positive equations of the form described above. Then,* `p` *is preserved by $\sim_G$ iff for each rule* `{t'} => {t''}` $\in R_{\mathcal{M}/G}$*, each equation* `p({t}) = true` *in $\mathcal{M}.AP$, and each A-unifier*[5]*, we can prove* `p({`ϑ`({t''})}) = true`.

Proposition 2 is very useful in practice, since we can use the Invariant Analyzer [13, 16] (InvA) to automate a good part of the effort of proving stability, leaving the remaining proof obligations for the Maude inductive theorem prover [14]. For example, the above mentioned proposition can be shown to be invariant under object permutations by InvA in a fully automatic way.

4.3 Checking That $\sim$ Is a Bisimulation

Once the state relation $\sim$ we want to exploit has been shown to preserve the atomic propositions of interest, we have to check that $\sim$ is a bisimulation.

In the case of an equivalence relation $\sim_G$ induced by a group G, proving that $\sim_G$ is a bisimulation amounts to showing joinability of suitable "critical pairs" between the state transition rules `{t} => {t'}` in the rule set $\mathcal{M}$,[6] and the rules `{t''} => {t'''}` of $\mathcal{M}/G$.



Indeed, bisimulation is ensured if we prove that for all ground A-unifiers θ between `t` and `t''` and each corresponding critical pair denoted with ordinary arrows in the diagram on the right, there is a rule R giving us a one-step rewrite $\{\theta(\mathtt{t'''})\} \rightarrow_{\mathcal{M}} \{w\}$ for which we can prove: $\{\theta(\mathtt{t'})\} \rightarrow^*_{\mathcal{M}/G} \{w\}$.

[5] Mappings of variables into non-necessarily ground terms such that $\vartheta(\mathtt{t'}) =_A \vartheta(\mathtt{t})$.
[6] The case of conditional rules in $\mathcal{M}$ is analogous, using *conditional* critical pairs.

Proposition 3 (correctness of bisimulation by joinability). *Let $\mathcal{M}$ be the rewrite theory under study, with an action of the group G. Then $\sim_G$ is a bisimulation between $\mathcal{M}$ and itself iff for all rules* {t} => {t'} *in $R_{\mathcal{M}}$, all rules* {t''} => {t'''} *in $\mathcal{M}/G$, and all ground A-unifiers θ between* t *and* t'' *there is a state $\{w\}$ such that $\{\theta(\texttt{t'})\} \rightarrow^*_{\mathcal{M}/G} \{w\}$ and $\{\theta(\texttt{t'''})\} \rightarrow_{\mathcal{M}} \{w\}$.*

The above proposition requires considering the set of all ground A-unifiers which may be infinite. Fortunately, we can instead use A-unifiers with variables and, in particular, the *most general* ones. Since each ground A-unifier is an instance of a most general one, if we can prove the conditions in Proposition 3 for the finite set of most general A-unifiers, then we have proved bisimilarity. However, using the most general A-unifiers may not always automatically prove bisimilarity: some inductive joinability proof obligations may still be left.

That is, the use of most general A-unifiers yields the following *sound* and easy to automate proof method. First, we use the Maude A-unification command to find most general A-unifiers ϑ between {t} and {t''}, respectively the left-hand-sides of each rule {t} => {t'} of $\mathcal{M}$, and each rule {t''} => {t'''} of $\mathcal{M}/G$ (after a renaming of variables to ensure that they have no variables in common). Second, for each such A-unifier ϑ we can use the Maude search command to determine all possible 1-step rewrites $\{\vartheta(\texttt{t'''})\} \rightarrow_{\mathcal{M}} \{w\}$. Note that for each ground instance s of $\{\vartheta(\texttt{t'''})\}$ the obtained rewrite steps correspond to some of the possible transitions outgoing from state s. Last, we can use the search command again to check if at least one of such obtained terms $\{w\}$ can also be reached from $\{\vartheta(\texttt{t'})\}$ in $\mathcal{M}/G$. For example, applying this method to our running example yields six unifiers in the first step, each requiring one reachability check that is efficiently solved by the search command of Maude. Proposition 4 summarizes the method.

Proposition 4 (soundness of the bisimulation check). *Let $\mathcal{M}$ be the rewrite theory under study and $\sim_G$ an equivalence on states induced by the action of a group G. Then $\sim_G$ is a bisimulation between $\mathcal{M}$ and itself if for each rule* {t} => {t'} *in $\mathcal{M}$, rule* {t''} => {t'''} *in $\mathcal{M}/G$, and most general A-unifier ϑ between* t *and* t'', *there is one state $\{w\}$ with $\{\vartheta(\texttt{t'''})\} \rightarrow_{\mathcal{M}} \{w\}$ for which we can show $\{\vartheta(\texttt{t'})\} \rightarrow^*_{\mathcal{M}/G} \{w\}$.*

4.4 Defining and Verifying Canonizer Functions

The next step is to define canonizer functions c : State → State in a protecting extension $\mathcal{M}$+c of the rewrite theory $\mathcal{M}$ under study. Note that in order to define c we may need to define some auxiliary functions (e.g. the ordering relations used in symmetry reduction to determine orbit representatives).

Definition 11 (c-extension of a rewrite theory). *Let $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ be the rewrite theory under study. A* c*-extension of $\mathcal{M}$ is a protecting extension of $\mathcal{M}$ of the form $\mathcal{M} + \texttt{c} = (\Sigma \cup \Sigma_{\texttt{c}}, E \cup E_{\texttt{c}} \cup A, R, \phi_{\texttt{c}})$ where $\texttt{c} \in \Sigma_{\texttt{c}}$ with* c : State → State, *and $\phi_{\texttt{c}}$ extends ϕ by making all functions in $\Sigma_{\texttt{c}}$ frozen.*[7]

[7] Imposing frozenness on the operators of $\Sigma_{\texttt{c}}$ is needed for the result of [8, Lemma 1].

Many candidate canonizers may exist for a given bisimulation $\sim$, each leading to different results in terms of the size of the reduced state space and computational performance. In any case, all canonizer functions must preserve $\sim$, i.e. they must be $\sim$-canonizers. This may require some theorem proving but it can be relatively easy to check in most cases, since we can use the equations E_c and show that each one preserves $\sim$.

For example, in the case of *local* reduction strategies [10] for symmetries based on a group G with generators $H \subseteq G$, the equations $E_{\mathtt{c}}$ defining `c` are of the form `c({t}) = c([[g]]({t})) if [[g]]({t})<{t}` with $g \in H$, `_<_` defining an ordering relation on states, plus an equation `c({t})={t} [owise]` to deal with the case when none of the previous equations is applicable, that is, when there is no way to transform a state into a *smaller* equivalent one by applying a generator (or inverse of a generator). Since such equations define `c` in terms of group actions or of the identity function when all conditions fail, preservation of the equivalence $\sim_G$ induced by G is immediate by the very definition of `c`.

Examples of local search strategies are implemented in our prototype tool [7]. In our running example, we can define such a canonizer as follows:

```
ceq   c( {            < i | x > < j | y > c1   } )
    = c( { [[i<->j]]( < i | x > < j | y > c1 ) } )
    if i < j /\ x < y .
eq c({c1}) = {c1} [ owise ] .
```

A very similar situation is that of *enumeration* strategies [10], where canonizers are defined as `c({t}) = min{[[f]]({t}) | f` $\in G\}$. Again, preservation of $\sim_G$ by c follows from the very definition of `c`. Indeed, for all states u, `c`(u) will be necessarily of the form $[\![g_1 \bullet g_2 \bullet \cdots \bullet g_n]\!](u)$, with each g_i being a generator. We call the equation format described above *group application form.*

Proposition 5 (group application $\sim_G$ canonizers). *Let $\mathcal{M}$ be the rewrite theory under study, $\sim_G$ the state equivalence induced by a group action, $\mathcal{M}/G$ as in Definition 10, and $\mathcal{M} + \mathtt{c}$ a `c`-extension of $\mathcal{M}$ such that the equations of $E_{\mathtt{c}}$ defining `c` are in* group application form. *Then, `c` is a $\sim_G$-canonizer.*

In practice, when specifying $\sim_G$-canonizers in the above form, all we have to check are the executability properties of the equations of $\mathcal{M} + \mathtt{c}$: termination, (ground) confluence and sufficient completeness, plus $\mathcal{M}$ protected in $\mathcal{M} + \mathtt{c}$, for which we can use the standard Maude tools.

Note that proving ground confluence of `c` is not sufficient to show that `c` is a *strong* canonizer. It may still be the case that for some two states u, v such that $u \sim v$ we have that `c`$(u) \neq$ `c`(v). For example, in the case of equivalences $\sim_G$ induced by a group G generated by $H \subseteq G$ as a monoid, to prove that `c` is a strong canonizer we also need to show that for all group elements in $g \in H$ and states s we have `c`$(s) =$ `c`$([\![g]\!](s))$. It is easy to see that if this holds, an inductive argument allows us to conclude `c`$(s) =$ `c`$([\![f]\!](s))$ for all $f \in G$ and hence for any two equivalent states $s \sim_G s' = [\![f]\!](s)$. Of course, there are cases in which no check is needed. For instance, it is well-known that enumeration strategies yield strong canonizers, while local strategies are not strong in general.

4.5 Defining c-Reductions

The next step is defining a c-reduction of $\mathcal{M}$ as a rewrite theory $\mathcal{M}/\mathtt{c}$. This is very useful, since then *no changes to a model checker are needed to support c-reductions*: we just model check $\mathcal{M}/\mathtt{c}$. We show that $\mathcal{M}/\mathtt{c}$ can be easily obtained by applying a theory transformation $\mathcal{M} \mapsto \mathcal{M}/\mathtt{c}$ defined as follows.

Definition 12 (c-reduction of a rewrite theory). *Let* $\mathcal{M} + \mathtt{c} = (\Sigma \cup \Sigma_\mathtt{c}, E \cup E_\mathtt{c} \cup A, R, \phi_\mathtt{c})$ *be a* c*-extension of* $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ *for which* c *is a* $\sim$*-canonizer of an equivalence bisimulation* $\sim$*. We then call* $\mathcal{M}/\mathtt{c} = (\Sigma \cup \Sigma_\mathtt{c}, E \cup E_\mathtt{c} \cup A, R_\mathtt{c}, \phi_\mathtt{c})$ *a* c-reduction *of* $\mathcal{M}$*, where* $R_\mathtt{c} = \{\mathtt{t => c(t') if cond} \mid (\mathtt{t => t' if cond}) \in R\}$.

$\mathcal{M}/\mathtt{c}$ is very much like $\mathcal{M}$, except that each rule `t => t' if cond` in R, is transformed into a rule `t => c(t') if cond`, i.e. into a rule where the canonizer function `c` is applied to the right hand side to ensure that canonization is performed after each system transition. For our running example we obtain, e.g. a rule: `rl { < i | x > credit(i) c1 } => c({ < i | s(x) > c1 }).`

This transformation is supported by our prototype [7] for the class of object-based rewrite theories. Our transformation exploits Maude reflective features: it is defined by a function that manipulates the meta-representation of the input theory to be c-reduced.

For some rules in $\mathcal{M}/\mathtt{c}$ it may be more efficient not to apply the canonizer after each step. For instance, if we know that the corresponding rule in $\mathcal{M}$ will always result in a canonical state we can save the time of applying the canonizer.

It is trivial to show that $\mathcal{M}/\mathtt{c}$ is a c-reduction by construction, and in particular that $\mathcal{K}_\mathcal{M}/\mathtt{c} = \mathcal{K}_{\mathcal{M}/\mathtt{c}}$. It can also be shown that it has good executability properties. By the properties required for $E_\mathtt{c}$, it inherits all the properties of the equational part of $\mathcal{M}$, namely sufficient completeness, confluence and termination modulo A. Moreover, it can be shown that $\mathcal{M}/\mathtt{c}$ is coherent modulo A.

Theorem 2 (executability of $\mathcal{M}/\mathtt{c}$). *Let* $\mathcal{M}$ *be the rewrite theory under study, and let* $\mathcal{M}/\mathtt{c}$ *be as in Definition 12. If* $\mathcal{M} + \mathtt{c}$ *has good executability properties, then* $\mathcal{M}/\mathtt{c}$ *also has good executability properties.*

The above theorem means that we can use $\mathcal{M}/\mathtt{c}$ for model checking analysis. For example, in our running example we can use the Maude LTL model checker to successfully verify the property $\Diamond\Box\neg$`some-message` (*"eventually there will be no more messages forever"*) efficiently. If we explore the whole state space of our running example using the Maude reachability analyzer we can check that the state space of the c-reduced system is drastically smaller than that of the original system. For instance, if we choose an initial state with 4 empty accounts with 4 messages for each, the original state space has 625 states, while the c-reduced one has only 70.

This is just a simple example: the performance experiments reported in [8, Sect. B] include examples taken from the literature where the applied c-reductions provide drastic gains and allow analyzing systems whose original state spaces is too large to be effectively analyzed.

5 Related Work and Conclusions

Related Work. We briefly comment on some interesting related approaches besides the ones already mentioned. A complementary line of research focuses on automatic symmetry detection, proposed for some model checkers, e.g. SPIN [10] and ProB [17]. Our approach does not forbid (though does not yet provide) automatically detected symmetries but focuses on user-definable ones, providing a methodology to check their correctness, with the main advantage being that we rely on tools and techniques used to perform the verification of the system itself. A related work is reported in [18] where formal methods are used to prove the soundness of the reduction techniques of [17].

Interesting are as well other state space reduction techniques, in particular those already proposed in the setting of rewriting logic and Maude, such as partial order reduction [19], and equational abstraction [20]. The closest one is [20], where abstractions are defined equationally. The main difference with our approach is in the kind of behavioral equivalence considered: equational abstractions yield simulations while we focus on bisimulations. With respect to [19] our approach is orthogonal and we are hence investigating how to combine them to improve the efficiency of rewriting-logic based interpreters of programming languages, in particular those with primitives for dynamic memory allocation.

Conclusions. We have presented *c-reductions*, a general bisimulation-based reduction technique that exploits canonizer functions whenever a bisimulation is an equivalence relation. The main differentiating features with respect to other state space reduction techniques are: (i) no changes to the underlying model checker are required, and reductions are defined using the original system description language; (ii) model checking and correctness proofs for the reduction are seamlessly integrated and supported by tools; (iii) semi-automation: both for applying the reduction and for checking their correctness; and (iv) generality: it subsumes in a uniform way symmetry reduction as well as other kinds of reductions (e.g. name reuse and name abstraction).

We have presented the basic concepts, described some typical classes of reductions, and illustrated how they can be analyzed. Our methodology performs a series of incremental steps Sect. 4.1–4.5, which include checking that the equivalence relation is a bisimulation and that the reduction strategies preserve such equivalence relation. Even if not presented here, we have performed a set of experimental results (see [8, Sect. B]) were we have observed a comparable performance with respect to symmetry reduction extensions of mature tools such as SPIN and performance gains with respect to previous implementations of symmetry reduction in Maude [6].

The flexibility of our approach has allowed us to define a wide range of reductions. Beyond the classical permutation and rotation symmetries, we have considered some simple cases of name reuse and name abstraction, which are crucial to deal with the infinite state spaces of systems with dynamic allocation of resources. Indeed, compared to the approach presented in [3, 11] we are able to treat a wider class of systems, where identifiers of symmetric objects can appear

as pointers in attributes of other objects, and with wider classes of symmetries such as rotational ones. Similar remarks can be made about [6], with respect to which we offer a wider class of reduction strategies and better performance.

Even though we have emphasized reductions based on group actions, the c-reduction approach is more general and accepts any possible canonizer function. Correctness proof methods fully covering the general case should be developed in future work. A preliminary version of our tool is publicly available [7].

References

1. Wahl, T., Donaldson, A.F.: Replication and abstraction: Symmetry in automated formal verification. Symmetry 2, 799–847 (2010)
2. Hendriks, M., Behrmann, G., Larsen, K., Niebert, P., Vaandrager, F.: Adding Symmetry Reduction to Uppaal. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 46–59. Springer, Heidelberg (2004)
3. Bosnacki, D., Dams, D., Holenderski, L.: Symmetric SPIN. International Journal on Software Tools for Technology Transfer 4, 92–106 (2002)
4. Meseguer, J.: Conditional rewriting logic as a united model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Rodríguez, D.E.: Combining techniques to reduce state space and prove strong properties. In: WRLA. ENTCS, vol. 238(3), pp. 267–280 (2009)
7. C-Reducer, `http://sysma.lab.imtlucca.it/tools/c-reducer`
8. Lluch Lafuente, A., Meseguer, J., Vandin, A.: State space c-reductions of concurrent systems in rewriting logic (2012), Full version, `eprints.imtlucca.it/1012/`
9. Meseguer, J., Palomino, M., Martí-Oliet, N.: Algebraic simulations. Journal of Logic and Algebraic Programming 79, 103–143 (2010)
10. Donaldson, A.F., Miller, A.: A Computational Group Theoretic Symmetry Reduction Package for the Spin Model Checker. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 374–380. Springer, Heidelberg (2006)
11. Bošnački, D., Dams, D., Holenderski, L.: A Heuristic for Symmetry Reductions with Scalarsets. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 518–533. Springer, Heidelberg (2001)
12. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL Model Checker and Its Implementation. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 230–234. Springer, Heidelberg (2003)
13. The Maude Invariant Analyzer Tool (InvA), `http://camilorocha.info/software/inva`
14. Clavel, M., Palomino, M., Riesco, A.: Introducing the ITP tool: a tutorial. Journal of Universal Computer Science 12, 1618–1650 (2006)
15. Durán, F., Meseguer, J.: A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 69–85. Springer, Heidelberg (2010)
16. Rocha, C., Meseguer, J.: Proving Safety Properties of Rewrite Theories. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 314–328. Springer, Heidelberg (2011)
17. Spermann, C., Leuschel, M.: ProB gets nauty: Effective symmetry reduction for B and Z models. In: TASE, pp. 15–22. IEEE Computer Society (2008)

18. Turner, E., Butler, M., Leuschel, M.: A Refinement-Based Correctness Proof of Symmetry Reduced Model Checking. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 231–244. Springer, Heidelberg (2010)
19. Farzan, A., Meseguer, J.: Partial order reduction for rewriting semantics of programming languages. In: WRLA. ENTCS, vol. 176(4), pp. 61–78 (2007)
20. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theoretical Computer Science 403, 239–264 (2008)