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 counterexamples.* 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. In particular, *symmetry reductions* are bisimulation-based reductions of this kind which have been extensively studied (see, e.g., [1]). However, despite its effectiveness, symmetry reduction is not yet established as a standard feature of verification tools as is the case for other reduction techniques. Some exceptions exist (e.g. [2, 3]) but, for example, in SPIN [4], symmetry reduction is not a built-in feature, while other techniques are. There are several reasons for this: (i) automatic system detection is hard and rarely present (e.g. [3, 5]) and thus very often delegated to the system designer; (ii) their exploitation is sometimes done by enriching the system description language (e.g. *scalarset* datatypes in [2, 6]), 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

[⋆] This is unpublished work. A preliminary version of this work was informally presented in a *work-in-progress* session of a workshop (WRLA 2012) but the version presented there will of course *not* be published in the Springer LNCS workshop proceedings.

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 (iv) applies also to other reduction techniques and means that correctly model checking a formula in a reduced system requires discharging *proof obligations* ensuring that the reduced system is a correct reduction of the original one. The problem, however, is that most model checkers lack theorem proving support for discharging such proof obligation, so 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 being reduced? (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, and we report on performance experiments showing that c-reductions can achieve drastic state space reductions. Question (5) is answered by proposing rewriting logic [7] as an efficiently executable *logical framework* supported by a high-performance tool (Maude [8]) 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 an ample 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 [9]), and a great flexibility in the definition of reductions, which allow 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 case studies. A preliminary version of our tool is available for download [10].

Synopsis. §2 offers the necessary background. §3 presents `c`-reductions in a generic way, focusing on Kripke structures. §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. §5 covers related work and conclusions.[3]

2 Preliminaries

We will use a simple running example of a banking system[4] of concurrent objects of the same class (accounts) having a natural 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 [7], which can be specified as Maude [8] 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* $\phi : \Sigma \to \mathcal{P}_{\mathit{fin}}(\mathbb{N})$ *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 and

[3] For the convenience of reviewers we include an appendix with complementary material that may help understanding and assessing our work: the formal proofs (§A), a performance evaluation with literature benchmark (§B), and our case study (§C, §D).

[4] Indeed, it is a simplification of the model of a bank account system described in [8].

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 operation $\{\_\}$ 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 a (possibly empty) subconfiguration `c1`. 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 [8] (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 denumerable 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 [11], 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.

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 $K_{\mathcal{M}} = (T_{\mathtt{State}/E\cup A}, \rightarrow, L, AP)$ such that $T_{\mathtt{State}/E\cup A}$ are all `State`-sorted states, $\rightarrow$ is defined as $\{[u] \rightarrow [v] \mid \mathcal{M} \vdash u \rightarrow^{1}_{R,\mathtt{State}} v\}$ (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 finite and 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 A. Then the equivalence relation $\sim_G$ induced by G on A 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 trasposing 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 §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 [5] 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 example, 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. Strong canonizers can be obtained in more efficient and smarter ways as shown in §4.4, e.g. with *local search strategies* [5] 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 sometimes 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 [12, 6], 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 $K = (S, \rightarrow, L, AP)$ be a Kripke structure, and let $\mathtt{c} : S \rightarrow S$ 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 $K/\mathtt{c} = (S, (\rightarrow; \mathtt{c}), L, AP)$ the* c-reduction *of K, where the composed transition relation $\rightarrow; \mathtt{c}$ is defined by: $\rightarrow; \mathtt{c} = \{(s, \mathtt{c}(r)) \in S^2 \mid s \rightarrow r\}$.*

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

Theorem 1 ($\sim$-preservation). *Let $K = (S, \rightarrow, L, AP)$ 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 Maude's LTL Model Checker [13], Invariant Analyzer [14], Inductive Theorem Prover [15] and Church Rosser and Coherence Checker [16].

$$\begin{tikzcd}
 & \mathcal{M}.E \arrow[dl, Rrightarrow] \arrow[d, Rrightarrow] \arrow[drr, Rrightarrow] & & (i) \arrow[d, dotted] & \\
\mathcal{M}.R \arrow[ddd, Rrightarrow] \arrow[dddr, dashed, bend right=60, "(v)"'] & \mathcal{M}.AP \arrow[dddl, Rrightarrow] \arrow[ddd, Rrightarrow] & (iii)\,(ii) \arrow[l, dotted] \arrow[ll, dotted, bend right] \arrow[dr, dotted] & \mathcal{M}+G \arrow[d, Rrightarrow] \arrow[ddd, Rrightarrow, bend left] & \\
 & & & \mathcal{M}/G & (iv) \arrow[ul, dotted] \arrow[ddl, dotted] \\
 & & & & \\
\mathcal{M} \arrow[r, dotted, leftrightarrow, "(vi)"] & \mathcal{M}/\mathsf{c} & & \mathcal{M}+\mathsf{c} \arrow[ll, Rrightarrow] &
\end{tikzcd}$$

Fig. 1. Modules and steps.

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 § 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 of 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 (§4.1) which ensures $\sim_G$ to be an equivalence relation; (ii) verify that $\sim$ preserves the state predicates AP (§4.2) by analyzing their invariance under an auxiliary theory $\mathcal{M}/G$ that models group actions; (iii) verify that $\sim$ is a bisimulation (§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 (§4.4); (v) build the c-reduction $\mathcal{M}/\mathsf{c}$ of $\mathcal{M}$ (§4.5), and (vi) use $\mathcal{M}/\mathsf{c}$ for model checking analysis purposes. Our methology then ensures that an LTL 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 $\mathcal{M}$.

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 [10] 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 [10]; several of them are mentioned, together with detailed performance experiments and comparisons with other tools and methods in §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 §E, consist on: (a) "uncurrying" the desired group action function $[\![\cdot]\!]_{\_} : G \longrightarrow (\texttt{State} \to \texttt{State})$ as a function $[\![\cdot]\!]_{\_} : G \times \texttt{State} \longrightarrow \texttt{State}$; (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 $[\![\cdot]\!]_{\_} : H \times \texttt{State} \to \texttt{State}$, which extends uniquely to a monoid action $[\![\cdot]\!]_{\_} : H^* \times \texttt{State} \to \texttt{State}$ satisfying for each $u \in \texttt{State}$, $g \in H$, $w \in H^*$ the recursive equations: $[\![\epsilon]\!]u = u$ and $[\![wg]\!]u = [\![w]\!]([\![g]\!]u)$. Then it is easy to prove (see Appendix E) that the only possible group action $[\![\cdot]\!]_{\_} : G \times \texttt{State} \to \texttt{State}$ extending $[\![\cdot]\!]_{\_} : H \times \texttt{State} \to \texttt{State}$ 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 $[\![\cdot]\!]_{\_} : G \times \texttt{State} \to \texttt{State}$ can be replaced by the simpler monoid action $[\![\cdot]\!]_{\_} : H^* \times \texttt{State} \to \texttt{State}$.

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 $\mathit{List}[X]$ to the instance $\mathit{List}[H]$.

Definition 9 (group pre-action specification). *Let $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ be the rewrite theory under study with designated* ***`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 $\mathit{List}[H]$ 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* ***`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.[5]

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 ∼ 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 $R_{\mathcal{M}/G} = \{$`{t} => [[g]]({t})`$) \mid \texttt{g} \in H\}$.*

I.e, we replace the rules of $\mathcal{M}$ by rules that move from a state u to another 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.

[5] For the full proof see §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: $u \sim_G v \Leftrightarrow u \to^*_{R_{\mathcal{M}/G}} v$. 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 [17] 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 defined in 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*[6]*, we can prove* `p({`ϑ`({t''})}) = true`.

Proposition 2 is very useful in practice, since we can use the Invariant Analyzer [14, 17] (InvA) to automate a good part of the effort of proving stability, leaving the remaining proof obligations for Maude's inductive theorem prover [15]. 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 $\mathcal{M}$,[7] 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 $\{\theta(\mathtt{t'''})\} \to_{\mathcal{M}} \{w\}$ for which we can prove: $\{\theta(\mathtt{t'})\} \to^*_{\mathcal{M}/G} \{w\}$.

$$\begin{tikzcd}
\{\theta(\mathtt{t})\} \arrow[r, "\mathcal{M}"'] \arrow[d, "\mathcal{M}/G"'] & \{\theta(\mathtt{t'})\} \arrow[d, dotted, "\mathcal{M}/G"', "*"] \\
\{\theta(\mathtt{t'''})\} \arrow[r, dotted, "\mathcal{M}"'] & \{w\}
\end{tikzcd}$$

[6] Mappings of variables into non-necessarily ground terms such that $\vartheta(\mathtt{t'}) =_A \vartheta(\mathtt{t})$.

[7] The case of conditional rules in $\mathcal{M}$ is analogous, using *conditional* citical 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 a possibly infinite set of ground A-unifiers, but 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 succeed in proving 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 Muade's 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 Maude search. 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 $\texttt{c} : \texttt{State} \rightarrow \texttt{State}$ in a protecting extension $\mathcal{M}+\texttt{c}$ 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 reductions 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 $\texttt{c} : \texttt{State} \rightarrow \texttt{State}$, and $\phi_{\texttt{c}}$ extends ϕ by making all functions in $\Sigma_{\texttt{c}}$ frozen.*[8]

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

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 [5] for symmetries based on a group G with generators $H \subseteq G$, the equations $E_{\texttt{c}}$ 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 [10]. 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 [5] where canonizers are defined as `c({t}) = min{[[f]]({t})` $\mid$ `f` $\in G\}$. Again, preservation of $\sim_G$ by c follows from the very definition of `c`. Indeed, for all states u, $\texttt{c}(u)$ 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 $\mathcal{M} + \texttt{c}$ a* `c`*-extension of $\mathcal{M}$ such that the equations of $E_{\texttt{c}}$ 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 $\mathcal{M} + \texttt{c}$: termination, (ground) confluence and sufficient completeness, plus $\mathcal{M}$ protected in $\mathcal{M} + \texttt{c}$, 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 $\texttt{c}(u) \neq \texttt{c}(v)$. 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 $\texttt{c}(s) = \texttt{c}([\![g]\!](s))$. It is easy to see that if this holds, an inductive argument allows us to conclude $\texttt{c}(s) = \texttt{c}([\![f]\!](s))$ 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 $\mathtt{c}$-reductions

The next step is defining a $\mathtt{c}$-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* $\mathtt{c}$*-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 ($\mathtt{c}$-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* $\mathtt{c}$*-extension of* $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ *for which* $\mathtt{c}$ *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* $\mathtt{c}$-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 [10] for the class of object-based rewrite theories. Our transformation exploits Maude's reflective features: it is defined by a function that manipulates the meta-representation of the input theory to be $\mathtt{c}$-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 $\mathtt{c}$-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 Maude's 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 Maude's rechability analyzer we can check that the state space of the $\mathtt{c}$-reduced system is actually 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 $\mathtt{c}$-reduced one has only 70.

This is just a simple example: our actual performance experiments in Appendix B include examples taken from the literature where the applied $\mathtt{c}$-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 [5] and ProB [3]. 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 [3].

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 such approach 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 based on canonizer functions that can be exploited 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 verification steps §4.1-§4.5 which include checking that the equivalence relation is a bisimulation and that reduction strategies preserve such equivalence relation. Even if not presented here for space reasons we have performed a set of experimental results (see §B) were we have observed a comparable performance with respect to mature tools such as SPIN and performance gains with respect to a previous implementations of symmetry reductions in Maude [9].

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 [12, 6] 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 [9], 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 [10].

References

1. Wahl, T., Donaldson, A.F.: Replication and abstraction: Symmetry in automated formal verification. Symmetry **2** (2010) 799–847
2. Hendriks, M., Behrmann, G., Larsen, K.G., Niebert, P., Vaandrager, F.W.: Adding symmetry reduction to uppaal. In: FORMATS. LNCS 2791, Springer (2003)
3. Spermann, C., Leuschel, M.: ProB gets nauty: Effective symmetry reduction for B and Z models. In: TASE, IEEE Computer Society (2008) 15–22
4. Holzmann, G.: The SPIN model checker: primer and reference manual. Addison-Wesley Professional (2003)
5. Donaldson, A.F., Miller, A.: A computational group theoretic symmetry reduction package for the SPIN model checker. In: AMAST. (2006) 374–380
6. Bosnacki, D., Dams, D., Holenderski, L.: Symmetric SPIN. International Journal on Software Tools for Technology Transfer **4** (2002) 92–106
7. Meseguer, J.: Conditional rewriting logic as a united model of concurrency. Theoretical Computer Science **96** (1992) 73–155
8. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude. Volume 4350 of LNCS. Springer (2007)
9. Rodríguez, D.E.: Combining techniques to reduce state space and prove strong properties. In: WRLA. Volume 238(3) of ENTCS. (2009) 267 – 280
10. C-Reducer, `http://sysma.lab.imtlucca.it/tools/c-reducer`.
11. Meseguer, J., Palomino, M., Martí-Oliet, N.: Algebraic simulations. Journal of Logic and Algebraic Programming **79** (2010) 103–143
12. Bosnacki, D., Dams, D., Holenderski, L.: A heuristic for symmetry reductions with scalarsets. In: FME, Springer (2001)
13. Eker, S., Meseguer, J., Sridharanarayanan, A.: The maude LTL model checker and its implementation. In: SPIN. Volume 2648 of LNCS., Springer (2003)
14. The Maude Invariant Analyzer Tool (InvA), `camilorocha.info/software/inva`.
15. Clavel, M., Palomino, M., Riesco, A.: Introducing the ITP tool: a tutorial. Journal of Universal Computer Science **12** (2006) 1618–1650
16. Durán, F., Meseguer, J.: A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications. In Ölveczky, P.C., ed.: WRLA. Volume 6381 of LNCS., Springer (2010) 69–85
17. Rocha, C., Meseguer, J.: Proving safety properties of rewrite theories. In Corradini, A., et al., eds.: CALCO. Volume 6859 of LNCS. Springer (2011) 314–328
18. Turner, E., Butler, M.J., Leuschel, M.: A refinement-based correctness proof of symmetry reduced model checking. In: ASM. LNCS 5977, Springer (2010)
19. Farzan, A., Meseguer, J.: Partial order reduction for rewriting semantics of programming languages. In: WRLA. Volume 176 of ENTCS. (2007) 61–78
20. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theoretical Computer Science **403** (2008) 239–264

A Proofs

This section documents the proofs for the main formal results of the paper. The proofs of those results that are less immediate or more inherent to our approach are described in detail, while the proofs of the rest of the results are sketched.

In some of the proofs, we sometimes use $\rightarrow_{\mathcal{M}}$ as an abbreviation of $\rightarrow_{R_{\mathcal{M}}}$ (as we did throughout the paper) when referring to the derivation of rewrite steps using the rules R of a rewrite theory $\mathcal{M}$. The latter version is preferred in proofs that involve derivations of terms in $\mathcal{M}$ obtained using equations.

We start with Theorem 1 which ensures c-reduced Kripke structures to be bisimulation preserving for c being a $\sim$-canonizer and $\sim$ being an equivalence relation and a bisimulation.

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

Proof. Let s and s' be two arbitrary states of S such that $s \sim s'$. Condition (i) of Def. 4 holds trivially by the definition of $\sim$.

Condition (ii) is also easy to see. Indeed if we have $s \rightarrow r$, we know that s' can simulate the transition $s \rightarrow r$ by a transition $s' \rightarrow r'$ since $\sim$ is a bisimulation on K. But then we have $r \sim \mathsf{c}(r')$ (since c preserves $\sim$, and $\sim$ is an equivalence relation an hence symmetric and transitive).

Condition (iii) is also easy to show (c.f. beside figure (ii)). Indeed, a transition $s' \rightarrow r' \rightarrow_{\mathsf{c}} \mathsf{c}(r)$, can be simulated by some transition $s \rightarrow r$ such that $r \sim r'$ since $\sim$ is a bisimulation on K. But, as in the above case, we also have that $r' \sim \mathsf{c}(r)$ (since c preserves $\sim$ and $\sim$ is an equivalence relation and hence transitive). □

$$\begin{tikzcd}
s \arrow[rr] \arrow[d, squiggly, no head] & & r \arrow[d, squiggly, no head] \\
s' \arrow[r] & r' \arrow[r, "\mathsf{c}"'] \arrow[ur, squiggly, no head] & \mathsf{c}(r')
\end{tikzcd}$$

(i)

$$\begin{tikzcd}
s' \arrow[r] \arrow[d, squiggly, no head] & r' \arrow[r, "c"'] \arrow[dr, squiggly, no head] & c(r') \arrow[d, squiggly, no head] \\
s \arrow[rr] & & r
\end{tikzcd}$$

(ii)

Next we sketch the proof of the proposition that identifies the properties to be checked on the implementation of a group action in order to ensure it to be correct.

Proposition 1 (correctness checks 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 $[\![\cdot]\!]_{\_} : H \times \mathtt{State} \rightarrow \mathtt{State}$ 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) $(\forall g : H, u : \mathtt{State})\ [\![g^{-1}]\!]([\![g]\!](u)) = u$.*

Proof. The proof is rather simple and relies on the notion of *implicit group action* (see Definitions 14 and 15 in §E). □

What follows is the proof for Proposition 2 which allow us to use the InvA tool for checking the invariance of propositions under the equivalence relation $\sim$ under study. Recall that the requirements for using this result are: (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.

Proposition 2 (proposition invariance by stability). *Let $\mathcal{M}/G$ the auxiliary rewrite theory defined in 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*[9]*, we can prove* `p({`ϑ`({t''}}) = true`*.*

Proof. Recall that what we need to prove is that for any two equivalent states $u \sim_G v$, and each proposition $p \in AP$ we have $p \in L(u) \Leftrightarrow p \in L(v)$.

By the construction of $\mathcal{M}/G$ we have that $u \sim_G v$ is equivalent to $u \rightarrow^*_{\mathcal{M}/G} v$, i.e. $\sim_G$-equivalence amounts to reachability in $\mathcal{M}/G$. In addition, our implementation of the the labeling function L for mapping states into sets atomic propositions realizes $p \in L(u)$ with a boolean predicate $\mathtt{p}(u)$. Hence, the invariance of AP under $\sim_G$ can be clearly recasted as follows: for any two states u, v such that $u \rightarrow^*_{R_{\mathcal{M}/G}} v$, and each proposition $p \in AP$ we have $p(u) = p(v)$.

Now, this is exactly the problem of checking invariance of boolean predicates studied [17], where it has been shown that it suffices to check for each rule `{t'} => {t''}` $\in R_{\mathcal{M}/G}$, each equation `p({t}) = true` in $\mathcal{M}.AP$, and each A-unifier ϑ between *t'* and *t* ϑ between `t'` and `t`, whether `p({`ϑ`([[g]]({t''})}) = true`.
□

We now prove Proposition 3 which allows us to reduce the problem of showing that an equivalence relation $\sim_G$ is a bisimilation to a problem of joinability of 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 $\{\theta(\mathtt{t'})\} \rightarrow^*_{\mathcal{M}/G} \{w\}$ and $\{\theta(\mathtt{t'''})\} \rightarrow_{\mathcal{M}} \{w\}$.*

Proof. Recall that what we need to prove is that for any two bisimilar states $u \sim u'$, any transition $u \rightarrow_{R_{\mathcal{M}}} v$ from u to v can be simulated by a transition $u' \rightarrow_{R_{\mathcal{M}}} v'$ from u' to a state v' bisimilar to v ($v \sim v'$), and vice vera. Graphically,

[9] Mappings of variables into non-necessarily ground terms such that $\vartheta(\mathtt{t'}) =_A \vartheta(\mathtt{t})$.

$$\begin{tikzcd} u \arrow[r, "R_{\mathcal{M}}"'] \arrow[d, phantom, "\wr"] & v \arrow[d, phantom, "\wr"] \\ u' \arrow[r, dotted, "R_{\mathcal{M}}"'] & v' \end{tikzcd}$$

where dotted arrows are the ones whose existence must be shown.

Now, it should be clear that by the construction of $\mathcal{M}/G$ we have that $s \sim s'$ is equivalent to $s \rightarrow^*_{\mathcal{M}/G} s'$, i.e. bisimilarity amounts to reachability in $\mathcal{M}/G$. So the property we have to show can be recasted as showing the existence of state v' and the dotted arrows in the diagram below

$$\begin{tikzcd} u \arrow[r, "R_{\mathcal{M}}"'] \arrow[d, "R_{\mathcal{M}/G}"', "*"] & v \arrow[d, dotted, "R_{\mathcal{M}/G}"', "*"] \\ u' \arrow[r, dotted, "R_{\mathcal{M}}"'] & v' \end{tikzcd}$$

for all possible states u, u' and v satisfying the relations of the diagram.

This proof can be easily simplified by the observation that $\rightarrow^*_{\mathcal{M}/G}$ is the closure of $\rightarrow_{\mathcal{M}/G}$ so that any of the previous diagrams can be decomposed as follows

$$\begin{tikzcd} u \arrow[r, "R"'] \arrow[d, "R_{\mathcal{M}/G}"'] & v \arrow[d, dotted, "R_{\mathcal{M}/G}"'] \\ u_1 \arrow[r, dotted, "R"'] \arrow[d, "R_{\mathcal{M}/G}"'] & v_1 \arrow[d, dotted, "R_{\mathcal{M}/G}"'] \\ u_2 \arrow[r, dotted, "R"'] \arrow[d, "R_{\mathcal{M}/G}"'] & v_2 \arrow[d, dotted, "R_{\mathcal{M}/G}"'] \\ \ldots \arrow[r, dotted, "R"'] \arrow[d, "R_{\mathcal{M}/G}"'] & \ldots \arrow[d, dotted, "R_{\mathcal{M}/G}"'] \\ u' \arrow[r, dotted, "R"'] & v' \end{tikzcd}$$

By induction we can show that we can reduce the problem to solving the above simple squares. But we know that in our setting each transition $u \rightarrow_{R_{\mathcal{M}}} v$ is just an instance $\{\theta_1(\texttt{t})\} \rightarrow_{R_{\mathcal{M}}} \{\theta_1(\texttt{t'})\}$ of a rule `t => t'` of $\mathcal{M}$, given by a variable assignment θ_1 such that $u = \theta_1(\texttt{t})$. Similarly, each transition $u \rightarrow_{\mathcal{M}/G} u_1$ is just an instance $\{\theta'_1(\texttt{t''})\} \rightarrow_{R_{\mathcal{M}}} \{\theta_1(\texttt{t'''})\}$ of a rule `t'' => t'''` of $\mathcal{M}/G$. The interesting case is now when $\theta_1(\texttt{t})$ is equal to $\theta'_1(\texttt{t''})$ (say with a unifier $\theta = \theta_1 \cup \theta'_1$), in which case we have reduced the proof to solving the desired joinability problem, i.e. the existence of the dotted arrows in the below diagram

$$\begin{tikzcd} \{\theta(\texttt{t})\} \arrow[r, "R_{\mathcal{M}}"'] \arrow[d, "R_{\mathcal{M}/G}"'] & \{\theta(\texttt{t'})\} \arrow[d, dotted, "R_{\mathcal{M}/G}"', "*"] \\ \{\theta(\texttt{t'''})\} \arrow[r, dotted, "R_{\mathcal{M}}"'] & \{w\} \end{tikzcd}$$

□

Next we show Proposition 4 which allows us to check the above joinability problem by resorting to Maude's unification and reachability capabilities.

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* $\{\texttt{t}\}$ `=>` $\{\texttt{t'}\}$ *in $\mathcal{M}$, rule* $\{\texttt{t''}\}$ `=>` $\{\texttt{t'''}\}$ *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 $\{\vartheta(\texttt{t'})\} \rightarrow^*_{\mathcal{M}/G} \{w\}$.*

Proof. By Proposition 3 showing bisimimulation amounts to showing the joinability of the critical pairs that arise when considering rules of $\mathcal{M}$ and rules of $\mathcal{M}/G$ as illustrated in the diagram (1) below

$$\begin{tikzcd}
\{\theta(\texttt{t})\} \arrow[r, "R_{\mathcal{M}}"'] \arrow[d, "R_{\mathcal{M}/G}"'] & \{\theta(\texttt{t'})\} \arrow[d, dotted, "R_{\mathcal{M}/G}"', "*"] \\
\{\theta(\texttt{t'''})\} \arrow[r, dotted, "R_{\mathcal{M}}"'] & \{w\}
\end{tikzcd}$$

(1)

$$\begin{tikzcd}
\{\vartheta(\texttt{t})\} \arrow[rr, "R_{\mathcal{M}}"'] \arrow[dd, "R_{\mathcal{M}/G}"'] & & \{\vartheta(\texttt{t'})\} \arrow[dl, dotted, "R^*_{\mathcal{M}/G}"] \arrow[ddl, dotted, "R^*_{\mathcal{M}/G}"] \arrow[dddl, dotted, "R^*_{\mathcal{M}/G}"] \\
& \{w_1\} & \\
\{\vartheta(\texttt{t'''})\} \arrow[ur, dotted, "R_{\mathcal{M}}"'] \arrow[r, dotted, "R_{\mathcal{M}}"'] \arrow[dr, dotted, "R_{\mathcal{M}}"'] & \{w_i\} & \\
& \{w_n\} &
\end{tikzcd}$$

(2)

What we will show is that such joinability is given whenever we perform the reachability checks enumerated in the proposition we are proving, graphically depicted in the diagram (2) above.

The proof is rather simple. Any ground unifier θ in diagram (1) is just an instance of one of the most general unifiers ϑ in diagram (2), say with a substitution of variables by ground terms η (i.e. $\theta = \eta \circ \vartheta$). If the reachability check is successful for ϑ, we know that there is a term $\{w_i\}$ such $\{\vartheta(\texttt{t'''})\} \rightarrow_{R_{\mathcal{M}}} \{w_i\}$ and $\{\vartheta(\texttt{t'})\} \rightarrow^*_{R_{\mathcal{M}/G}} \{w_i\}$. Now, the former rewrites can be grounded. That is, any state $\{w_i\}$ can be instantiated as the ground term (i.e. a state) $\{\eta(w_i)\}$, and idem for the rewrites $\{\eta(\vartheta(\texttt{t'''}))\} \rightarrow_{R_{\mathcal{M}}} \{\eta(w_{i,j})\}$ and $\{\eta(\theta(\texttt{t'}))\} \rightarrow^*_{R_{\mathcal{M}/G}} \{\eta(w_i)\}$. In other words, $\{\eta(w_i)\}$ is the state $\{w\}$ that joins the pairs for the ground unifier θ. □

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 $\mathcal{M} + \mathsf{c}$ a c-extension of $\mathcal{M}$ such that the equations of E_{c} defining c are in* group application form*. Then, c is a $\sim_G$-canonizer.*

The proof is trivial. As we informally explain in §4.4 we can just exploit the group application form of the equations defining the canonizer c to devise an inductive proof. □

Before proving Theorem 2 we show a helpful lemma regarding the coherence of a c-reduced theory.

Lemma 1 (coherence of $\mathcal{M}/\mathsf{c}$). *Let $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ be ground confluent, terminating, sufficiently complete, and with R ground coherent with E modulo A, c be a canonizer function, and $\mathcal{M}/\mathsf{c} = (\Sigma \cup \Sigma_{\mathsf{c}}, E \cup E_{\mathsf{c}} \cup A, R_{\mathsf{c}}, \phi_{\mathsf{c}})$ be the c-reduction of $\mathcal{M}$. Then all rules in R_{c} are ground coherent with $E \cup E_{\mathsf{c}}$ modulo A.*

$$\begin{tikzcd}
u \arrow[r, "1", "R_{\mathcal{M}}"'] \arrow[dd, "E/A!"'] & v \arrow[rr, "\mathsf{c}"'] \arrow[dr, "E/A"', "*"] & & \mathsf{c}(v) \arrow[dr, "E/A"', "*"] & \\
& & w & & \mathsf{c}(w) \\
u' \arrow[r, "1", "R_{\mathcal{M}}"'] & v' \arrow[rr, "\mathsf{c}"'] \arrow[ur, "E/A"', "*"] & & \mathsf{c}(v') \arrow[ur, "E/A"', "*"] &
\end{tikzcd}$$

Proof. Since $\mathcal{M}$ is topmost and all operators in Σ_{c} are frozen (see Def. 11), any ground $\Sigma \cup \Sigma_{\mathsf{c}}$-term u of sort `State` such that $\mathcal{M} \vdash u \rightarrow^1_R v$ must be a Σ term. Therefore, its $E \cup E_{\mathsf{c}}/A$-canonical form u' is exactly its E/A-canonical form. By the ground coherence of R with E modulo A we obtain the inner pentagon in the diagram on the right which trivially yields the outer pentagon, proving $E \cup E_{\mathsf{c}} \vdash \mathsf{c}(v) = \mathsf{c}(v')$ as desired. □

Last, we can prove Theorem 2.

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

Proof. The proof immediatedly follows from the fact that $\mathcal{M}/\mathsf{c}$ has the same equational part as $\mathcal{M}/\mathsf{c}$ and from Lemma 1. □

			SymmSpin				c-reductions			
			weak		strong		weak		strong	
Experiment	n	if	sf	tf	sf	tf	sf	tf	sf	tf
Peterson	2	-0.50	-0.48	+0.00	-0.49	+0.00	-0.45	+0.00	-0.45	+0.00
	3	-0.83	-0.75	-0.79	-0.82	-0.79	-0.77	+0.00	-0.83	+1.50
	4	-0.96	-0.91	-0.77	-0.95	-0.84	-0.93	-0.75	-0.95	+0.50
DBM	7	-0.99	-0.99	-0.99	-0.99	-0.80	-0.98	-0.87	-0.99	+2.00
	8	-0.99	-0.99	-0.99	-0.99	-0.48	-0.99	-0.95	-0.99	-0.65

Table 1. SymmSpin vs c-reductions in Maude (the values for SymmSpin are derived from [12]; only comparison of reduction factors is meaningful).

B Performance experiments

We present here a selection of the experiments we have carried out, with the main purpose of validating the effectiveness of the implementation of the c-reduction approach in Maude.

Our main hypothesis to be checked is that the relative performance gain (in terms of runtime and state space reductions) is comparable to the one obtained by state-of-the-art model checkers. The second hypothesis is that c-reductions are more efficient than the previous approach to symmetry reductions in Maude described in [9]. Finally, we enrich the experiments where we combine various c-reductions not supported by the previously mentioned tools. The Maude implementation of the benchmark examples used here are included in the release of our prototype [10].

Comparison with SymmSPIN We have chosen SymmSPIN [12, 6] as a representative model checker with which to compare our approach. SymmSPIN extends the SPIN [4] model checker with support for symmetry reductions. It is worth to remark that we do not perform absolute comparison as we aim at checking the usefulness of our approach (experiments with SymmSPIN cannot be reproduced since the tool is no more available for download).

We have implemented in Maude two of the benchmark models tested in [12, 6], namely Peterson's mutual exclusion protocol[10], and a database management system.[11] Both examples exhibit a full symmetry due to the presence of families of replicated concurrent processes with identical behavior. In both cases we manually translated the Promela specification into Maude specifications, in the most faithful manner we could.

We have considered various c-reduction strategies. For the sake of simplicity, we consider only the two best strategies of SymmSPIN (i.e. *pc-segmented* and *pc-sorted*) and the two best of our own strategies our implementation (partly inspired by the ones proposed in [12, 6]), for which regards time and state space

[10] Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc. (1996)

[11] Valmari, A.: Stubborn sets for reduced state generation. In: Proceedings on Advances in Petri nets 1990, Springer (1991)

				Reflection-based			c-reductions					
Params		Not reduced					weak			strong		
n	m	states	if	states	sf	tf	states	sf	tf	states	sf	tf
2	5	38,029	-0.50	19,295	-0.49	+9.62	21,630	-0.43	+1.14	19,025	-0.50	+0.46
3	2	72,063	-0.83	13,280	-0.82	+7.30	29,534	-0.59	+1.01	12,235	-0.83	+0.34
3	3	952,747	-0.83	174,428	-0.81	+5.65	307,532	-0.68	-0.50	160,121	-0.83	+0.30

Table 2. Reflection-based symmetry reduction vs. c-reductions in Maude.

reduction. We call them *strong* and *weak* as they are actually strong and non-strong canonizers.

Table 1 presents the results for instances of the models with increasing number of components (n). We offer only results for those instances for which it has been possible to generate the unreduced state spaces, so to compare the relative gain of the reductions in terms of relative state reduction factor (sf) and relative time reduction factor (tf). A relative state (resp. time) reduction factor of k indicates that, if the non-reduced exploration involved n states (resp. seconds) then the reduced exploration involved $m = n + (k \times n)$ states (resp. seconds). Put it otherwise, k is calculated as $k = (m/n) - 1$. Clearly, values below 0 indicate a reduction, while numbers above 0 an increase. The table also includes the "ideal" relative reduction factor (if), which in the case of full symmetries is $(1/n!) - 1$ where n is the size of the permutations (i.e. the number of replicated processes in the examples), since the size of each orbit is at most $n!$. So for an exhaustive exploration that requires s states and t seconds an ideal factor k means that we can expect the reduction to require at the best the exploration of $n + (k \times s)$ states in $t + (k \times t)$ seconds (where k is obviously negative). Clearly, "sf" and "tf" are always greater than "if". We highlight cells corresponding to the best results in each category (state space and run-time gain) for each model instance. It is worth to remark that the results for the SymmSPIN tool have been derived from [12]. Reproducing them was not possible since SymmSPIN is not available for download. Fortunately, we are not interested in absolute measures but in the relative gain of the reduction.

The two approaches provide state space reductions near to the ideal gain. The two strategies based on `weak` canonizers provide very similar outcomes, while the ones based on `strong` canonizers reduce similarly. SymmSPIN is more time-efficient, which is not a surprise, since the reduction algorithms are implemented in a procedural language (C) and efficiently compiled, while our implementation is based on a declarative language (Maude) running over an intepreter (the Maude engine).

Comparison with reflection-based symmetry reduction in Maude Our second set of experiments aims at checking whether c-reductions offers better performances than the symmetry reduction implementation in Maude described in [9]. Very briefly, the main idea of [9] is to select the canonical representative of a state on the basis of the lexicographical order of the meta-representation of the state, which is achieved by exploiting Maude's reflection capabilities.

The comparison is performed over the Chain-Replication protocol[12] used in [9]. As in the previous case, the replication of identical processes yields a full symmetry. Table 2 presents our results in the same format as Table 1 with the only exception that the model is instantiated with two parameters: the number n of replicated components and the number m of queries they perform. The table shows that the reductions of the reflection-based approach of [9] are bounded by the ones obtained by our strategies. That is: our weak strategy (resp. strong) provides a better (resp. worse) time reduction factor, while our strong (resp. weak) strategy provides a better (resp. worse) space reduction factor. In particular our weak strategy offers worse space reductions, while the strong one offers better space reductions. More interestingly, our reduction strategies introduce much less time overhead, differing often by an order of magnitude. This is not a surprise, since resorting to Maude's meta-level involves a considerable overhead.

Canonizers based on the lexicographical order of meta-representations are also implemented in our prototype [10] where similar results can be observed.

Exploiting permutations, rotations, reuse and abstraction Our last set of experiments regards the joint application of a number of c-reductions of different nature, not supported by the previously mentioned tools. As a test case we have considered a message-passing solution to the Dining Philosophers problem along the lines of the case study used in an approach[13] where newly generated messages (representing forks) receive fresh identities (as the original purpose that approach was to reason about individual messages). There are a couple of regularities that can be exploited in the form of c-reductions and that happen to yield bisimulations (for an empty set of atomic predicates): the rotational symmetry of philosophers, the full symmetry of messages, the reuse of message identifiers and their abstraction. Of course, the situation is different when one considers state predicates that involve the identity of philosophers or messages. However, our goal here is to validate the effectiveness of the mentioned c-reductions.

Table 3 reports the results. The table presents the size of the state space and the time (in ms) to generate it, for instances of the model with increasing number of philosophers. The table considers the state spaces generated in the following cases: reuse of message identifiers (NR), reuse together with (rotational and full) symmetry reduction (NR+RS+FS), abstraction of name identifiers (NA), and abstraction of name identifiers together with (rotational) symmetry reduction (NA+RS). The sizes of the unreduced state spaces are not shown since they are infinite (due to the creation of messages with fresh identifiers).

[12] van Renesse, R., Schneider, F.B.: Chain replication for supporting high throughput and availability. In: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6, USENIX Association (2004) 77

[13] Distefano, D., Rensink, A., Katoen, J.P.: Model checking birth and death. In Baeza-Yates, R.A., et al., eds.: TCS. Volume 223., Kluwer (2002)

	NR		NR+RS+FS		NA		NA+RS	
n	states	time	states	time	states	time	states	time
2	21	0	10	0	18	0	10	0
3	115	8	27	12	76	0	27	8
4	801	100	86	60	322	20	86	48
5	6,251	1,456	275	320	1,364	124	275	248
6	54,869	20,765	982	1,732	5,778	740	982	1,412
7	541,731	463,080	3,499	11,828	24,476	6,624	3499	8,124
8	O.T.	O.T.	13,016	49,651	103,682	29,329	13,006	35,594
9	O.T.	O.T.	48,828	247,987	439,204	192,072	48,819	186,329

Table 3. c-reductions for the dining philosophers.

The first clear advantage is that reuse of message identifiers yields finite state spaces (since the number of messages in each state is bounded by n). Besides this, we see how combining various c-reductions results in better and more efficient reductions. To be noticed is the fact that name reusing alone ran out of time (more than 5 hours) for models instantiated with more than 7 philosophers, while combining it with symmetry reductions (for messages and philosophers) allows us to manage larger instances. In particular, the best reductions are obtained with the combination of name abstraction and rotational symmetry (NA+RS), while name abstraction alone (NA) offers the fastest explorations from 2 to 8 philosophers, and is outperformed by the combination NA+RS for greater instances.

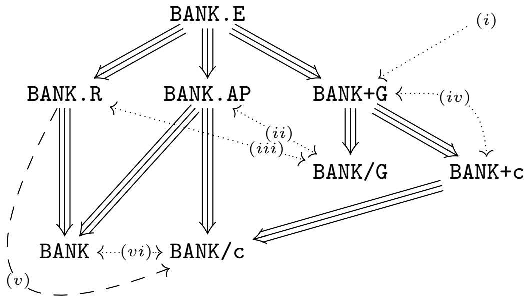

Fig. 2. Modularisation of the specification and verification steps in the running example.

C Validation

This section provides a detailed description of the application of methodology to the running example. The full specification of this example can be found in §D. The modular structure of the specification is illustrated in Fig. 2 and follows the one of Fig. 1.

Step (i): Proving that `BANK+PERMUTATION` correctly specifies the action of permutations. Recall that in order to show that `BANK+PERMUTATION` is a group action we have to show that the following equation holds

```
[[ i <-> j ]] ([[ i <-> j ]] ({c1})) = {c1}
```

where `i`, `j` are arbitrary natural numbers and `c1` is an arbitrary configuration.

We proceed by structural induction on `c1`. The base cases are trivial. For example, consider the case when `c1` is `none`. Then what we have to show is the equation

```
[[ i <-> j ]] ([[ i <-> j ]] ({none})) = {none}
```

but applying equations `teq1` and `teq2` we can reduce the left hand side of the equation to obtain.

```
{[[ i <-> j ]] (none)} = {none}
```

and applying again the same equation concludes the proof for this case. The rest of the base cases are solved similarly.

Consider, now the general case where we have to show

```
[[ i <-> j ]] ([[ i <-> j ]] ({c1 c2})) = {c1 c2}
```

Applying `teq1` and `teq3` (twice) to the left hand side of the above equation we obtain

```
{[[ i <-> j ]] ([[ i <-> j ]] (c1)) [[ i <-> j ]] ([[ i <-> j ]] (c2))}
= {c1 c2}
```

Now, the application of the inductive hypothesis concludes the proof.

As a byproduct, we can safely add the equation

```
eq [teq9] : [[ i <-> j ]] ([[ i <-> j ]] (c1)) = c1 .
```

to module BANK+PERMUTATION which turns out to be convenient for theorem proving purposes. In particular, equation teq9 will be exploited by some of the tools we use in the below verification steps.

Step (ii): Checking invariance of predicates under object permuations. In order to check the invariance of the atomic proposition (state observations) under object transpositions we rely on the InvA tool as explained in §4.2.

Consider for example the atomic proposition some-message which identifies states in which there is at least one message for an existing object. Such proposition is defined by the following equations in module BANK.APP.[14]

```
fmod BANK.APP is
  ...
  eq [some-message-eq1] : some-message({ < i | x > credit(i)    }) = true .
  eq [some-message-eq2] : some-message({ < i | x > credit(i) c1 }) = true .
  ...
endfm
```

Moreover, consider the rules (implemented in BANK/PERMUTATION) that define transitions between equivalent states by transposing objects.

```
mod BANK/PERMUTATION is
   ...
  rl [transposition1] :
     {                < i | x > < j | y >   }
  => { [[ i <-> j ]] ( < i | x > < j | y > ) } .

  rl [transposition2] :
     {                < i | x > < j | y > c1  }
  => { [[ i <-> j ]] ( < i | x > < j | y > c1) } .

endm
```

Now, in order to show the invariance of some-message under object permutations we can use InvA to perform what is called a *stability* check of the proposition some-message under the rules of BANK/PERMUTATION.

[14] The predicates p do not need to be completely specified: the cases in which they evaluate to false can be neglected, and hence we use a module $\mathcal{M}.APP$ (which contains the partial definition of the propositions) of which $\mathcal{M}.AP$ is a protecting extension (which contains a complete definition of the propositions). In particular, $\mathcal{M}.AP$ can be obtained from $\mathcal{M}.APP$ just by adding equations of the form p({t}) = false [owise]. This is in general convenient, but in the case of InvA is also necessary since it does not support the owise keyword.

```
Maude> select INVA .
Maude> loop init .
rewrites: 976 in 2ms cpu (2ms real) (406836 rewrites/second)
    Invariant Analyzer 1.1 - February 8th 2011
    (with Church-Rosser Checker 31)
Maude> (analyze-stable some-message(s:State) in BANK.APP BANK/PERMUTATION .)
rewrites: 15357 in 23ms cpu (24ms real) (649015 rewrites/second)
Checking BANK/PERMUTATION ||- some-message => O some-message ...
Proof obligations generated:  1
Proof obligations discharged: 1
Success!
```

The check is successful: all proof obligations are automatically discharged and we can conclude the desired invariance.

We can proceed similarly with the rest of the atomic propositions. For instance, consider `two-dollars` which identifies states in which at least one object has at least two dollars.

```
fmod BANK.APP is
  ...
  eq [two-dollars-eq1] : two-dollars({ < i | s(s(x)) >    }) = true .
  eq [two-dollars-eq2] : two-dollars({ < i | s(s(x)) > c1 }) = true .
  ...
endfm
```

We can use again InvA to show the invariance of `two-dollars` under object permutations:

```
Maude> select INVA .
Maude> loop init .
rewrites: 976 in 2ms cpu (2ms real) (406836 rewrites/second)
    Invariant Analyzer 1.1 - February 8th 2011
    (with Church-Rosser Checker 31)
Maude> (analyze-stable two-dollars(s:State) in BANK.APP
          BANK/PERMUTATION .)
rewrites: 15571 in 16ms cpu (19ms real) (918643 rewrites/second)
Checking BANK/PERMUTATIONS ||- two-dollars => O two-dollars ...
Proof obligations generated:  2
Proof obligations discharged: 2
Success!
```

The check is again successful: all proof obligations are automatically discharged and we can conclude the desired invariance.

Step (iii): Checking that object permutations yield a bisimulation. The next step is to check wether object transpositions induce a bisimulation by applying the methodology described in §4.3.

First, we compute the most general unifiers between the left-hand sides of the rules in `BANK.R` and the rules `BANK/PERMUTATION` by using the `unify` command. We focus in particular on the rule `credit2` of `BANK.R`

```
mod BANK.R is
  ...
  rl [credit2] :
         { < i | x    > credit(i)  c1 }
      => { < i | s(x) >            c1 } .
  ...
endm
```

and the rule `transposition2` of `BANK/PERMUTATION`

```
mod BANK/PERMUTATION is
  ...
  rl [transposition2] :
     {                 < i | x > < j | y > c1  }
  => { [[ i <-> j ]] ( < i | x > < j | y > c1) } .
  ...
endm
```

The desired most general unifiers are then computed as follows:

```
Maude> unify { < i:Nat | x:Nat    > credit(i:Nat)  c1:Configuration }
          =? { < j:Nat | y:Nat > < k:Nat | z:Nat > c2:Configuration  } .
unify in BANK/C : {< i | x > credit(i) c1}
          =? {< j:Nat | y:Nat > < k:Nat | z:Nat > c2:Configuration} .
Decision time: 1ms cpu (5ms real)

Solution 1
c1 --> #7:Configuration < #3:Nat | #4:Nat > < #5:Nat | #6:Nat >
i --> #1:Nat
x --> #2:Nat
c2:Configuration --> #7:Configuration credit(#1:Nat) < #1:Nat | #2:Nat >
j:Nat --> #3:Nat
y:Nat --> #4:Nat
k:Nat --> #5:Nat
z:Nat --> #6:Nat

...

Solution 6
c1 --> < #3:Nat | #4:Nat >
i --> #1:Nat
x --> #2:Nat
c2:Configuration --> credit(#1:Nat)
j:Nat --> #1:Nat
y:Nat --> #2:Nat
k:Nat --> #3:Nat
z:Nat --> #4:Nat
```

We obtain thus 6 unifiers (we present only the first and last ones for brevity). Let us consider the first one (`Solution 1`). What we have to do now is to consider the right-hand-side of `transposition2` after the application of the unifier `Solution 1`, which is

```
{ < n5:Nat | n4:Nat > < n3:Nat | n6:Nat > [[ n5:Nat <-> n3:Nat ]]
  ( c7:Configuration credit(n1:Nat) < n1:Nat | n2:Nat > )}
```

and to compute all state terms reachable from it in one rewrite step with the rules of BANK.R.[15]

```
Maude> search in BANK.BISIMULATION : { < n5:Nat | n4:Nat > < n3:Nat | n6:Nat
> [[ n5:Nat <-> n3:Nat ]] ( c7:Configuration credit(n1:Nat) < n1:Nat | n2:Nat
>  ) } =>1 S:State .
search in BANK.R : {< n5:Nat | n4:Nat > < n3:Nat | n6:Nat >
[[n5:Nat <-> n3:Nat]]c7:Configuration credit(n1:Nat) < n1:Nat | n2:Nat >} =>1
S:State .

Solution 1 (state 1)
states: 2  rewrites: 5 in 0ms cpu (0ms real) (113636 rewrites/second)
S:State --> {< n5:Nat | n4:Nat > < n3:Nat | n6:Nat > < [[n5:Nat <-> n3:Nat]]n1:Nat
| s(n2:Nat) > [[n5:Nat <-> n3:Nat]]c7:Configuration}

No more solutions.
states: 2  rewrites: 5 in 0ms cpu (0ms real) (40322 rewrites/seconds
```

Only one state is reached (`state 1`). What is left to do is to check whether that state is reachable (in BANK/PERMUTATION) from the state that results from applying the unifier `Solution 1` to the right hand side of rule `credit2`.

```
Maude> search [1] in BANK/PERMUTATION : { < n1 | s(n2) > c7 < n3 | n4 > < n5
| n6 > } =>* {< n5 | n4 > < n3 | n6 > < [[n5 <-> n3]] n1 | s(n2) > [[n5 <->
n3]] (c7)} .
search in BANK/PERMUTATION : {< n1 | s(n2) > c7 < n3 | n4 > < n5 | n6 >} =>*
{< n3 | n6 > < n5 | n4 > < [[n5 <-> n3]]n1 | s(n2) > [[n5 <-> n3]]c7} .

Solution 1 (state 6)
states: 7  rewrites: 54 in 0ms cpu (4ms real) (82442 rewrites/second)
empty substitution
```

The search is successful, and so are the rest of the needed checks. Therefore, we can conclude that object permutations yield a bisimulation.

Step (v): Building the c-reduction BANK/C. Building the c-reduction for our running example is then quite easy. All we need to do is to define the module BANK/C as a protecting extension of BANK+C with the rules of BANK.R refactored as defined in Def. 12. The result is basically as follows:

[15] We actually use a simple auxiliary module BANK.PERMUTATION that imports both BANK.R and BANK+PERMUTATION since we need to rewrite terms containing symbols from BANK+PERMUTATION.

```
mod BANK/C is

  protecting BANK+C .
  ...
  rl [credit1] :
              { < i | x    > credit(i) }
      => c({ < i | s(x) >            }) .

  rl [credit2] :
              { < i | x    > credit(i) c1 }
      => c({ < i | s(x) >            c1 }) .

endm
```

Step (vi): Exploring the reduced state space. We can now use the module `BANK/C` for state space exploration analysis. For example, we can explore the state space of our running example using the `search` command of Maude, starting from a state with 4 empty accounts and 4 messages:

```
Maude> search in BANK/C : {init(4,4)} =>* s:State .
search in BANK/C : {init(4, 4)} =>* s:State .
...
states: 70  rewrites: 14333 in 26ms cpu (26ms real) (536615 rewrites/second)
```

which yields a state space with 70 states (against the 625 of the original state space).

More pragmatically, we can use Maude's LTL model checker to successfully verify some temporal properies. For instance, we can verify property $\Diamond\Box\neg$`some-message` to determine that *"eventually there will be no more messages forever"* as follows:

```
Maude> red modelCheck({init(4,4)}, <>[]~ some-message) .
reduce in MUTEX-CHECK : modelCheck({init(4, 4)}, <> []~ some-message) .
rewrites: 14485 in 17ms cpu (19ms real) (841906 rewrites/second)
result Bool: true
```

obtaining a positive result.

D Specification of the running example

This section documents the complete specification of our running example.

```
---
--- This is a simple example based on the BANK-ACCOUNT model
--- which can be found in the Maude Manual (ch. 8)
--- The example has been adapted for the sake of
--- illustrating the c-reduction technique
---

---
--- Natural numbers implementation
--- to ensure compatibility with verification tools
---
fmod NATURALS is

  sort Nat .

  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  ops 1 2 3 4 5 6 7 8 9 10 : -> Nat .

  eq 1 = s(0) .
  eq 2 = s(1) .
  eq 3 = s(2) .
  eq 4 = s(3) .
  eq 5 = s(4) .
  eq 6 = s(5) .
  eq 7 = s(6) .
  eq 8 = s(7) .
  eq 9 = s(8) .
  eq 10 = s(9) .

  op _ != _ : Nat Nat -> Bool .

  vars n m : Nat .

  eq 0 != 0 = false .
  eq 0 != s(n) = true .
  eq s(n) != 0 = true .
  eq s(n) != s(m) = ( n != m) .

  op _ < _ : Nat Nat -> Bool .

  eq n < 0 = false .
  eq 0 < s(n) = true .
  eq s(n) < s(m) = ( n < m ) .


endfm

---
--- The main module defining the signature
--- (simplified CONFIGURATION-like notation)
---
fmod BANK.E is

  protecting NATURALS .

  sorts Object Message Configuration State .
  subsort Message Object < Configuration .

  op <_|_> : Nat Nat -> Object [ctor] .
  op credit : Nat -> Message [ctor] .
  op __ : Configuration Configuration -> Configuration [assoc comm] .
  op none : -> Configuration [ctor] .
  op {_} : Configuration -> State [ctor frozen] .

  vars c1 : Configuration .

  --- Identity of object yuxtaposition
  eq c1 none = c1 .

  --- A generator of initial state
  op init : Nat Nat -> Configuration .
  op credits : Nat Nat -> Configuration .

  vars n m : Nat .

  eq init(0,m) = none .
  eq init(s(n),m) = < n | 0 > credits(m,n) init(n,m) .

  eq credits(0,n) = none .
  eq credits(s(m),n) = credit(n) credits(m,n) .

endfm


---
--- The behavioural rules of the example
---
mod BANK.R is

  protecting BANK.E .

  vars i x : Nat .
  vars c1 : Configuration .

  --- A simple rule for crediting an account
  rl [credit1] :
         { < i | x    > credit(i)  }
      => { < i | s(x) >            } .

  rl [credit2] :
         { < i | x    > credit(i)  c1 }
      => { < i | s(x) >            c1 } .


endm


---
--- Implementation of object permutations
---
fmod BANK+PERMUTATION is

  protecting BANK.E .

  vars i j k x y : Nat .
  vars obj1 : Object .
  vars msg1 : Message .
  vars c1 c2 : Configuration .

  sort Transposition .

  op _<->_ : Nat Nat -> Transposition [ctor comm] .

  op [[_]] _ : Transposition State -> State [frozen] .
  op [[_]] _ : Transposition Configuration -> Configuration [frozen] .
  op [[_]] _ : Transposition Nat -> Nat .

  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) .

  --- This is useful for verification purposes
  --- and is anyhow something one has to prove to hold
  --- for the implementation of transpositions to be correct
  eq [teq8] : [[ i <-> i ]] (c1) = c1 .
  eq [teq9] : [[ i <-> j ]] ([[ i <-> j ]] (c1)) = c1 .
  eq [teq10] : [[ i <-> i ]] (k) = k .
  eq [teq11] : [[ i <-> j ]] ([[ i <-> j ]] (k)) = k .


endfm

---
--- Module that (partially) defines the atomic propositions
--- (state observations)
---
fmod BANK.APP is

  protecting BANK+PERMUTATION .

  vars i j x y : Nat .
  vars c1 : Configuration .

  ops empty-account two-dollars some-message : State -> [Bool] [frozen] .

  --- It is important to note that this module partially defines
  --- the propositions (only true cases) since it will be only used
  --- for checking their stability under permutations with InvA
  --- and InvA does not need a total definition
  --- (and cannot handle the [owise] directive)

  --- Some account has no money
  eq [empty-account-eq1] : empty-account({ < i | 0 >    }) = true .
  eq [empty-account-eq1] : empty-account({ < i | 0 > c1 }) = true .

  --- Some account has at least 2 dollars
  eq [two-dollars-eq1] : two-dollars({ < i | s(s(x)) >    }) = true .
  eq [two-dollars-eq2] : two-dollars({ < i | s(s(x)) > c1 }) = true .

  --- There is a message for some object
  eq [some-message-eq1] : some-message({ < i | x > credit(i)    }) = true .
  eq [some-message-eq2] : some-message({ < i | x > credit(i) c1 }) = true .

endfm

---
--- Module that totally defines the atomic propositions
--- (state observations)
---
fmod BANK.AP is

  protecting BANK.APP .

  vars c1 : Configuration .

  --- Some account has no money
  eq [empty-account-eq2] : empty-account({c1}) = false [owise] .

  --- Some account has at least 2 dollars
  eq [two-dollars-eq1] : two-dollars({c1}) = false [owise] .

  --- There is a message for some object
  eq [some-message-eq1] : some-message({c1}) = false [owise] .

endfm


---
--- The full system
---
mod BANK is

  protecting BANK.AP .
  protecting BANK.R .

endm


---
--- Permutations as rules for proving AP-preservation
---
mod BANK/PERMUTATION is

  protecting BANK+PERMUTATION .

  vars i j k x y : Nat .
  vars c1 : Configuration .

  rl [transposition1] :
     {                 < i | x > < j | y >   }
  => { [[ i <-> j ]] ( < i | x > < j | y > ) } .

  rl [transposition2] :
     {                 < i | x > < j | y > c1  }
  => { [[ i <-> j ]] ( < i | x > < j | y > c1) } .

  --- Auxiliary constructors useful for verification purposes
  ops n1 n2 n3 n4 n5 n6 n7 n8 : -> Nat .
  ops c3 c4 c5 c6 c7 c8 : -> Configuration .

endm

---
--- Auxiliary module for checking bisimulation
---
mod BANK.BISIMULATION is

  protecting BANK.R .
  protecting BANK+PERMUTATION .

endm


---
--- The c-extension of BANK that defines the c-canonizer
--- for object permutations
---
mod BANK+C is

  protecting BANK+PERMUTATION .

  op c : [State] -> [State] .

  vars i j x y : Nat .
  vars c1 : Configuration .

  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 ] .

endm


---
--- The c-reduction of BANK-RULES
---
mod BANK/C is

  protecting BANK.AP .
  protecting BANK+C .

  vars i x : Nat .
  vars c1 : Configuration .

  --- A simple rule for crediting an account
  rl [credit1] :
           { < i | x    > credit(i) }
      => c({ < i | s(x) >           }) .

  rl [credit2] :
           { < i | x    > credit(i) c1 }
      => c({ < i | s(x) >           c1 }) .

endm

load model-checker.maude .

---
--- Module with predicates for the LTL model checker
---
mod BANK-PREDS is

    protecting BANK/C * (sort State to c-State , sort Nat to c-Nat ) .
    including SATISFACTION .

    subsort c-State < State .

    op some-message : -> Prop .

    vars c1 : Configuration .

    eq {c1} |= some-message = some-message({c1}) .

endm

---
--- Module for model checking the c-reduction
---
mod MUTEX-CHECK is

    protecting BANK-PREDS .
    including MODEL-CHECKER .
    including LTL-SIMPLIFIER .

endm
```

E Algebraic Specifications of Group Actions

In this section we describe a simple way to specify a group action in an algebraic way and how to verify that an action is indeed a group action.

Monoid and group actions. We start defining the action of a monoid on a set of A.

Definition 13 (monoid action). *Given a monoid* $M = (M, \bullet, e)$ *and a set* A *a* monoid action *of* M *on* A *is a monoid homomorphism* $\llbracket\cdot\rrbracket : M \to (A \to A)$*, that is (i)* $\llbracket e\rrbracket = id_A$ *(with* id_A *the identity on* A*), and (ii)* $\llbracket g \bullet g'\rrbracket = \llbracket g\rrbracket \circ \llbracket g'\rrbracket$*, where* $\_ \circ \_$ *denotes function composition in* $(A \to A)$*, i.e. given* $f, f' \in (A \to A)$*,* $a \in A$*,* $(f \circ f')(a) = f(f'(a)$*.*

A monoid action can be equivalently described in a uncurried form as a function $\llbracket\cdot\rrbracket\_ : M \times A \to A$ defined by the equality $\llbracket g\rrbracket a = \llbracket g\rrbracket(a)$. In this case $\llbracket\cdot\rrbracket\_$ is a monoid homomorphism iff (i) $\llbracket e\rrbracket a = a$, and (ii) $\llbracket g \bullet g'\rrbracket a = \llbracket g\rrbracket(\llbracket g'\rrbracket a)$.

Now, we can define the action of a group $G = (G, \bullet, e, (\_)^{-1})$ on a set A simply as a monoid action $\llbracket\cdot\rrbracket : G \to (A \to A)$ (as we did in Definition 5).

It is trivial to prove that if G is a group, M is a monoid, and $f : G \to M$ is a monoid homomorphism, then the image of G by f, $f[G]$ is a group. Therefore, for any group action $\llbracket\cdot\rrbracket : G \to (A \to A)$ we have $\llbracket G\rrbracket \subseteq Bij(A)$ as a subgroup inclusion, where $Bij(A)$ is the set of all bijective functions $f : A \to A$.

Implicit specification of group actions. A subset $H \subseteq G$ is a (monoid) generator (see Definition 5) of a group G if the unique monoid homomorphism

$$H^* \overset{\overline{j}}{\twoheadrightarrow} G : g_1 \dots g_n \mapsto g_1 \bullet \cdots \bullet g_n$$

induced by the inclusion $j : H \hookrightarrow G$ is surjective, where $(H^*, \_\_, \epsilon)$ denotes the free monoid of strings on H, with string concatenation as juxtaposition $\_\_$ and the empty string ϵ.

Note also that we can always define a function $i : H \to H^*$ such that $\overline{j}(i(g)) = g^{-1}$ since by the surjectivity of $\overline{j}$, for each $g \in H$ there must exist a (not necessarily unique) $w \in H^*$ such that $g^{-1} = \overline{j}(w)$.

Note that if $G \to (A \to A)$ is a group action and $j : H \hookrightarrow G$ is a set of monoid generators, then the composed homomorphism

$$H^* \overset{\overline{j}}{\twoheadrightarrow} G \xrightarrow{\llbracket\cdot\rrbracket} (A \to A)$$

is a monoid action.

furthermoreF, that the quotient monoid $H^*/i = H^*/\{gi(g) = \epsilon \mid g \in H\} \cup \{i(g)g = \epsilon \mid g \in H\}$ is a group, since each i-equivalence class $[g_1, \dots, g_n]$ has an inverse element $[i(g_1), \dots, i(g_n)]$, so that we have a further factorization

$$H^* \twoheadrightarrow H^*/i \twoheadrightarrow G \overset{\llbracket\cdot\rrbracket}{\twoheadrightarrow} \llbracket G\rrbracket \hookrightarrow Bij(A) \hookrightarrow (A \to A)$$

The particular choice of G in the above diagram does not matter much: G can be any group which is a quotient of H^*/i and has $\llbracket G \rrbracket$ as a quotient. In particular we could choose $G = A/i^*$ or $G = \llbracket G \rrbracket$.

This suggests defining a group action *implicitly* as follows.

Definition 14 (implicit group action). *Given a set of generators H, and a function $i : H \to H^*$, an* implicit *group action of H on a set A is a function $\llbracket\cdot\rrbracket : H \to (A \to A)$ such that for each $g \in H$ we have (inv$_1$) $\llbracket g \rrbracket \circ \llbracket i(g) \rrbracket = id_A$, and (inv$_2$) $\llbracket i(g) \rrbracket \circ \llbracket g \rrbracket = id_A$, where $\llbracket\cdot\rrbracket : H^* \to (A \to A)$ is the unique homomorphic extension of $\llbracket\cdot\rrbracket$.*

It can be shown that an implicit group action is indeed a group action.

Proposition 6. *Let H be a set of generators, and $\llbracket\cdot\rrbracket : H \to (A \to A)$ be an implicit group action on a set A. Then $\llbracket\cdot\rrbracket$ is the action on A by the group G generated by H.*

Proof. First, note that $\llbracket\cdot\rrbracket : H \to (A \to A)$ has a unique homomorphic extension to a monoid action (overloaded as $\llbracket\cdot\rrbracket$) $\llbracket\cdot\rrbracket : H^* \to (A \to A)$

Since $\llbracket\cdot\rrbracket$ is an implicit group action it satisfies the conditions (inv$_1$) and (inv$_2$) and it is trivial to check that these conditions mean that we have a factorization of homomorphims:

$$\begin{tikzcd}
H^* \arrow[rr, "\llbracket\cdot\rrbracket"] \arrow[d, two heads] & & (A \to A) \\
H^*/i \arrow[r, two heads, "\llbracket\cdot\rrbracket"] & \llbracket H^* \rrbracket \arrow[r, hook] & Bij(A) \arrow[u, hook]
\end{tikzcd}$$

Therefore, we can view the implicit group action as either (i) an action of the group H^*/i, (ii) an action of the group $\llbracket H^* \rrbracket$, or (iii) an action of any group G such that we have a surjective homomorphism $H^*/i \twoheadrightarrow G \twoheadrightarrow \llbracket H^* \rrbracket$. □

Note that we can always specify an implicit group action in uncurried form.

Definition 15 (implicit group action (uncurried)). *Given a set H of generators and a function $i : H \to H^*$, an* implicit group action *on a set A is a function $\llbracket\cdot\rrbracket_- : H \times A \to A$ such that for each $g \in H$ and $a \in A$ we have (inv'$_1$) $\llbracket g \rrbracket(\llbracket i(g) \rrbracket\rrbracket a) = a$, and (inv'$_2$) $\llbracket g \rrbracket(\llbracket i(g) \rrbracket\rrbracket a) = a$ where $\llbracket\cdot\rrbracket_- : H^* \times A \to A$ is the unique function extending $\llbracket\cdot\rrbracket_- : H \times A \to A$ such that for each $a \in A$, $g \in H$, $w \in H^*$ it holds $\llbracket \epsilon \rrbracket a = a$ and $\llbracket wg \rrbracket a = \llbracket w \rrbracket(\llbracket g \rrbracket a)$.*