Model Checking Strategy-Controlled Rewriting Systems

Rubén Rubio
Facultad de Informática, Universidad Complutense de Madrid, Spain
rubenrub@ucm.es

Narciso Martí-Oliet
Facultad de Informática, Universidad Complutense de Madrid, Spain
narciso@ucm.es

Isabel Pita
Facultad de Informática, Universidad Complutense de Madrid, Spain
ipandreu@ucm.es

Alberto Verdejo
Facultad de Informática, Universidad Complutense de Madrid, Spain
jalberto@ucm.es

Abstract

Strategies are widespread in Computer Science. In the domain of reduction and rewriting systems, strategies are studied as recipes to restrict and control reduction steps and rule applications, which are intimately local, in a derivation-global sense. This idea has been exploited by various tools and rewriting-based specification languages, where strategies are an additional specification layer. Systems so described need to be analyzed too. This article discusses model checking of systems controlled by strategies and presents a working strategy-aware LTL model checker for the Maude specification language, based on rewriting logic, and its strategy language.

2012 ACM Subject Classification Software and its engineering → Model checking; Theory of computation → Equational logic and rewriting; Theory of computation → Rewrite systems

Keywords and phrases Model checking, strategies, Maude, rewriting logic

Digital Object Identifier 10.4230/LIPIcs.FSCD.2019.34

Category System Description

Related Version An extended version of the paper is available at `http://maude.sip.ucm.es/strategies/tr0219.pdf` ([30]).

Funding Research partially supported by MCIU Spanish project TRACES (TIN2015-67522-C3-3-R). *Rubén Rubio*: Partially supported by MCIU grant FPU17/02319.

Acknowledgements We are grateful to the reviewers for their careful reading, and their detailed and very useful comments.

1 Introduction

Strategies are meaningful for artificial intelligence, games, semantics of programming languages, automated reasoning, etc. Although their purposes and formalizations differ, they all honor the Greek etymology of the word, meaning *the office of a general*, who is in charge of the overall planning of the operations. In the modeling of concurrent systems using rewriting techniques, strategies are a useful resource to capture the global behavior of the intended models. Since rewriting consists of a successive, non-deterministic and somehow unrelated application of rules anywhere within a term, strategies have been studied in deep [32], and different definitions have been proposed [8, 19]. Strategies as a first-class object have been exploited in tools like Stratego [10], Tom [4], and the specification languages ELAN [7], and Maude [13].

Model checking [11] is a consolidated formal method, which still evolves in multiple directions. Its classical setting is transition systems, where the notion of strategy is naturally defined. This paper studies the satisfaction of temporal properties by models controlled by strategies and how it can be checked, and applies the method to the Maude strategy language by implementing a strategy-aware model checker. It is built as an extension of the existing Maude LTL model checker [16] for systems specified in rewriting logic, already applied to various interesting systems [6, 23, 28].

Strategies and model checking together have already been addressed in the literature, but with a different approach and objectives. In the context of multiplayer games, several logics have been proposed to *reason* about player strategies like ATL* [1] and *strategy logic* [27]. However, strategies are not provided as input but quantified in the formula, and they are not represented explicitly. Other logics take past actions into account to condition its requirements like mCTL* [21]. In our case, strategies are part of the model specification while the property logic remains unaltered. As well, strategies should not be confused with heuristics to guide the search in the model-checker algorithms [14].

After reviewing the model-checking framework and strategies as defined in the literature, this paper discusses model checking linear temporal properties on strategy-controlled systems and a model transformation is proposed to match the classical setting and allow using the standard algorithms. Following a short introduction to rewriting logic [26], Maude [12], and its strategy language [15], we propose a small-step operational semantics from which model checking is defined according to the previous approach. The strategy-aware LTL model checker we have implemented is then described and illustrated by an example. The extended version of this document [30], the complete semantics of the Maude strategy language, and the model checker itself are all available at `http://maude.sip.ucm.es/strategies`.

2 Preliminaries

2.1 Model checking

Model checking [11] is a well-established formal method to ensure or refute the correctness of a model according to a temporal specification. In the classical setting, models are based on transition systems, formally described by Kripke structures [20] $\mathcal{K} = (S, \to, I, AP, \ell)$ where

- S is the set of states,
- $(\to) \subseteq S \times S$ is a serial binary relation on S,
- $I \subseteq S$ is a finite set of initial states,
- AP is a finite set of *atomic propositions*, and
- $\ell : S \to \mathcal{P}(AP)$ labels each state with the propositions that hold on it.

In turn, the property is expressed by a formula φ in some temporal logic like CTL, CTL* or LTL, which describes the intended behavior in terms of the atomic properties $p, q, \ldots \in AP$ combined by different temporal operators. The model-checking problem, deciding whether the model satisfies the formula $\mathcal{K} \vDash \varphi$, is decidable for any of the previous logics, a decidable transition relation, and a finite S. However, for both CTL* and LTL, model checking is PSPACE-complete and the models of interest usually have a huge number of states. In various situations, the expectation of refuting correctness in reasonable time is good enough.

The actual model executions $\pi = (s_k)_{k=1}^\infty$ leave propositional traces $\ell(\pi) := (\ell(s_k))_{k=1}^\infty$, from which the satisfaction of the formula is decided.

$$\begin{array}{cccccc} \pi & s_1 \longrightarrow s_2 \longrightarrow \cdots \longrightarrow s_n \longrightarrow \cdots & \in S^\omega \\ \ell(\pi) & \{p\} \quad\; \emptyset \quad\;\; \cdots \quad \{p,q\} \quad \cdots & \in \mathcal{P}(AP)^\omega \end{array}$$

Linear-time properties can always be characterized by a satisfaction relation $\ell(\pi) \vDash \varphi$ on propositional traces and an implicit universal quantification over all model paths π. Differently, branching-time properties combine universal and existential requirements on any state of the derivation, so that the execution should be seen as a tree.

2.2 Strategies

In rewriting systems, rules typically represent local transitions that are often not enough to describe complex computations. These intricacies are usually expressed at a higher level, describing how rules should be applied. This is the task of strategies, whose study goes back to the λ-calculus, as fixed criteria for selecting the next redex to be reduced [5]. Later, strategies were allowed to be aware of the derivation history and to be *explicit* [2, §11.5], expressed as programs that control the application of rules. We are interested in the latter kind of strategies, for which different abstract descriptions and practical representations have been proposed and implemented [8, 19].

Strategies are properly defined in the context of *abstract reduction systems* (ARS). An ARS [2] $\mathcal{A} = (S, \to)$ is a set of states S endowed with a binary relation $\to$. An element $(s, s') \in (\to)$ or $s \to s'$ is called a *reduction step*, and a finite or infinite sequence of connected reduction steps $s_0 \to s_1 \to \cdots \to s_n$ is a *derivation*. We denote by $\Gamma_\mathcal{A}^\omega$ the set of infinite derivations of $\mathcal{A}$ seen as words in S^ω,

$$\Gamma_\mathcal{A}^\omega := \{s_0 s_1 \cdots s_n \cdots : s_k \in S \text{ and } s_k \to s_{k+1}, k \geq 0\},$$

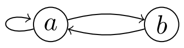
$\Gamma_\mathcal{A}^*$ is the set of finite derivations as words in S^*, and $\Gamma_\mathcal{A} := \Gamma_\mathcal{A}^\omega \cup \Gamma_\mathcal{A}^*$ the union of both in $S^\infty := S^\omega \cup S^*$. Considering both finite and infinite derivations is tedious, but we are interested in modeling computations and proofs as well as reactive systems behavior, for which they are respectively relevant. We say that $\mathcal{A}$ is *finite* if S is finite, and $\mathcal{A}$ is *finitary* if for any $s \in S$ the states s' such that $s \to s'$ are finitely many.

Several definitions of strategies are reviewed in [8], but two general formalizations are specially discussed:

1. *Abstract* or *extensional strategies* are subsets of derivations $E \subseteq \Gamma_\mathcal{A}$, that is, languages in S^∞ whose words w are $\mathcal{A}$-derivations with $w_k \to w_{k+1}$.
2. *Intensional strategies* are defined as partial functions $\lambda : S^* \to \mathcal{P}(S \cup \{\top\})$ that decide the possible next steps according to the past of the derivation. They must satisfy that for all $s, s' \in S$ and $v \in S^*$, $s' \in \lambda(vs)$ must imply $s \to s'$. The symbol $\top$ indicates that the derivation can stop there. In case $\lambda(w) = \emptyset$, the derivation cannot stop or continue, so it is discarded. Hence, these strategies can attempt rewriting paths that may eventually fail.

Extensional strategies represent an abstract selection of ARS executions as a whole, while the more constructive intensional strategies determine the next reduction in each step. Unlike in [8], we have considered unlabeled transition systems to simplify the presentation. Since classical model checking only considers properties on the states, labels can be easily added without repercussions. Moreover, the definition of intensional strategies has been modified to include the $\top$ symbol. Otherwise, derivations could stop at any step, which is inconvenient in practice. These definitions fall into the class of *history aware strategies* of [32], and intensional strategies are similar to *non-deterministic strategies* in games, except that these may select the next player action instead of the next state.

▶ **Example 1.** Consider the ARS $(\{a, b\}, \{(a, a), (a, b), (b, a)\})$



A strategy allowing at most one stay in b is described extensionally as $\{a\}^*\{b, \varepsilon\}(\{a\}^* \cup \{a\}^\omega)$, and intensionally as $\lambda(v) = \{a, \top\}$ if v contains a b, and $\lambda(v) = \{a, b, \top\}$ otherwise.

An intensional strategy induces an extensional one

$$E(\lambda) := \{w \in \Gamma_{\mathcal{A}} : w_i \in \lambda(w_0 \cdots w_{i-1}) \wedge (w \in S^\omega \vee \top \in \lambda(w))\}.$$

But the converse is not true, intensional strategies are less expressive than extensional ones. In the ARS of Example 1, the intensional strategy for $\{a\}^*$ has to be defined by $\lambda(v) = \{a, \top\}$ for any $v \in \{a\}^*$, since another a can always be added. And thus, the word a^ω would be included in $E(\lambda)$ by definition, so that $E(\lambda) \neq \{a\}^*$. Intuitively, intensional strategies decide on-the-fly while constructing the derivation, so they cannot decide on properties on the *infinity*. Languages recognized by automata with non-trivial acceptance conditions are examples of strategies that are necessarily extensional, but the more realistic devices or computations we are interested in modeling are very likely to be intensional. In any case, the extensional definition is simpler and will be useful.

Formally, intensional strategies are characterized as closed sets [8], which contain all words $w \in S^\omega$ whose finite prefixes are all prefixes of derivations within the strategy[1]. When discussing model checking, we will find an alternative characterization.

3 Strategy-aware model checking

Given a Kripke structure $\mathcal{K} = (S, \to, I, AP, \ell)$ and a strategy $E \subseteq \Gamma_{\mathcal{K}} := \Gamma_{(S,\to)} \cap IS^\infty$ starting from the initial states of $\mathcal{K}$, we want to give sense to model checking against a linear temporal formula φ and define the satisfaction relation $(\mathcal{K}, E) \models \varphi$. First, since temporal formulas are properly defined on infinite executions, we assimilate finite traces to infinite ones by repeating its last state forever. This is standard and fits with the idea of a finite machine that remains in its final state once stopped.

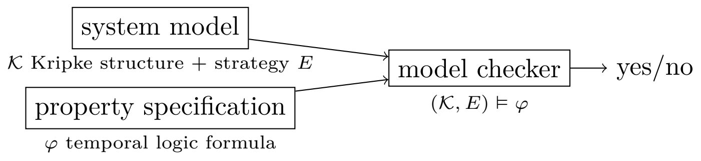

system model
$\mathcal{K}$ Kripke structure + strategy E
model checker
yes/no
$(\mathcal{K}, E) \vDash \varphi$
property specification
φ temporal logic formula

Figure 1 Model-checking procedure sketch.

Model checking a system controlled by a strategy against a linear-time property has an unavoidable and clear definition. As pointed in Section 2, the satisfaction of a linear property follows from a satisfaction relation $\rho \vDash \varphi$ on propositional traces $\rho \in \mathcal{P}(AP)^\omega$. Then,

▶ **Definition 2.** *Let φ be a linear formula, $(\mathcal{K}, E) \models \varphi$ if $\ell(\pi) \models \varphi$ for all $\pi \in E$.*

The set of allowed propositional traces $P = \{\, \rho \in \mathcal{P}(AP)^\omega : \rho \models \varphi \,\}$ completely defines the property, and the model-checking problem is the language containment problem $\ell(E) \subseteq P$, which is decidable and PSPACE-complete as long as $\ell(E)$ and P are ω-regular, like in the

[1] In fact, X^ω can be given a *prefix* topology (and even a distance) such that *closed* can be understood in the topological sense and the points so described are limits.

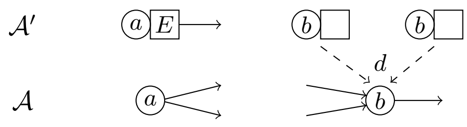


Figure 2 Strategy representation in an ARS (Definition 3).

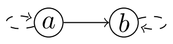
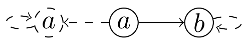
LTL case [29]. Although this is a clear starting point and it would provide an actual algorithm if strategies were given as Büchi automata, this is not usually the case since they are rather expressed in some intensional or syntactical form like the Maude strategy language described in Section 4.1. Thus, to effectively decide model checking with standard algorithms, we propose to encode the model controlled by strategies as another abstract reduction system. Any intensional strategy can be so encoded, but the actual procedure will depend on the strategy representation. In Section 4.2, we do it for the Maude strategy language by means of a small-step operational semantics.

▶ **Definition 3.** *Given an ARS $\mathcal{A} = (S, \to)$ and a strategy $E \subseteq \Gamma_\mathcal{A}$, the pair $(\mathcal{A}, E)$ is represented in the ARS $\mathcal{A}' = (X, R)$ with descent function $d : X \to S$ if*

$$d(\Gamma_{\mathcal{A}'} \cap J\, X^\infty \cap (X^\omega \cup X^* F)) = E$$

for some sets $J, F \subseteq X$ of initial and final states of the strategy.

The representation $\mathcal{A}'$ simulates the system $\mathcal{A}$ constrained by E, in the sense that the executions of $\mathcal{A}'$ are the traces selected by the strategy. Intuitively, the representation states include something else to guarantee that the strategy is respected, which can be stripped with the descent function d. The initial states for the strategy execution are the subset J, since other states in X may represent ongoing strategy executions. The final set F is only required to distinguish admitted finite traces from incomplete ones. Still, we do not want them for model checking: the trace extension of the first paragraph can be implemented in the above representation by adding self-loops in F. However, this cannot be done without care. For example, observe the following encoding for the strategy $\{a, ab\}$, where a and b are final.

The extended language is $\{a^\omega, ab^\omega\}$ according to our criterion. However, a loop in a will allow spurious derivations like aab^ω. This can be solved by adding an extra copy of the final states, like in the right figure. Now, we assume that the strategies are over infinite words and define the concept of model checking.

▶ **Proposition 4.** *Let $\mathcal{K} = (S, \to, I, AP, \ell)$ be a Kripke structure, $E \subseteq \Gamma_\mathcal{K}$ a strategy, and (X, R) a representation of $(\mathcal{A}, E)$ with descent function d and initial states J, $(\mathcal{K}, E) \models \varphi$ iff $\mathcal{K}' \models \varphi$ where $\mathcal{K}' = (X, R, J, AP, \ell \circ d)$.*

That is, model checking $\mathcal{K}$ controlled by E is model checking its representation $\mathcal{K}'$. The reason is that the propositional traces of any such $\mathcal{K}'$ are exactly those of E. This method can be applied to any intensional strategy, but model checking will only be decidable if the ARS representation is finite.

▶ **Proposition 5.** *A strategy $E \subseteq \Gamma_\mathcal{A}$ on a finitary ARS $\mathcal{A}$ can be represented in a finitary ARS iff E is intensional, i.e. there is an intensional strategy λ such that $E = E(\lambda)$. In that case, it can be represented in a finite ARS iff E is ω-regular.*

We can draw from this proposition that a strategy on a finite ARS must be intensional and ω-regular to match the classical model-checking framework and apply its algorithms, because otherwise it cannot be represented in a finite Kripke structure.

Notice that the discussion in this section is restricted to linear-time properties. The representation of Definition 3 is not adequate for branching-time properties, since branches need not be preserved while descending with d. A suitable model-checking definition for logics like CTL* passes by model checking a more restricted representation, any bisimulation of the ARS (S^+, R) where $v\, R\, vs$ if $v \in \lambda(v)$ and λ is an intensional strategy.

4 Maude and its strategy language

Maude is a specification language [13, 12] based on *rewriting logic* [26], a general framework for modeling concurrency proposed in 1992 by José Meseguer. Its specifications are organized in modules of different kinds:

1. *Functional* modules define *membership equational logic* theories, composed of an order-sorted signature Σ, equations E, and sort membership axioms to express that a term t belongs to a sort s. Equations and membership axioms can be conditional.

$$(\forall X) \qquad \begin{matrix} t = t' \\ t : s \end{matrix} \quad \text{if} \ \bigwedge_i u_i = u_i' \ \wedge \ \bigwedge_j v_j : s_j$$

   For the specification to be executable, equations are oriented and functional modules must be confluent and terminating [12, §4.6]. Bidirectional relations, like commutativity, associativity and identity, are specified apart as *structural axioms*.

2. *System* modules specify rewriting theories $\mathcal{R} = (\Sigma, E, R)$ by adding rewriting rules R to a functional specification. Unlike equations, rewriting rules need be neither confluent nor terminating, so they are likely to express non-deterministic behavior.

$$(\forall X) \qquad t \Rightarrow t' \ \text{ if } \ \bigwedge_i u_i = u_i' \ \wedge \ \bigwedge_j v_j : s_j \ \wedge \ \bigwedge_k w_k \Rightarrow w_k'$$

   However, rules are required to be coherent with equations and axioms [12, 26]. Conditional rules take a third type of conditions called rewriting conditions, that are satisfied if the term w_k can be rewritten to match w_k'.

The language syntax is a natural ASCII encoding of the mathematical notation above. Modules are a collection of declarations between `mod NAME is ... endm` (or `fmod`/`endfm` for functional modules). An operator $f : s_1 \times \cdots \times s_n \to s$ is defined as `op f : s1 .. sn -> s .` and structural axioms are inserted as attributes (`comm`, `assoc`, ...) between brackets. Equations are introduced by the keyword `eq` and rules by `rl`, prefixed by `c` if conditional.

▶ **Example 6.** The classical problem of the dining philosophers [18] is specified in the module `PHILOSOPHER-DINNER` of Listing 1. A philosopher is represented as a triple describing both hands contents (a fork ψ or nothing `o`) and an identifier. Rules `left` and `right` allow them to take the forks at their sides if they are in the table. The `release` rule restores both forks to the table. Since a circular table is represented by a list, we adopt the convention that the fork between the last and first philosophers is on the right, ensure it by the equation, and add a second `left` rule to allow the first philosopher to take this fork.

Listing 1 Dining philosophers problem specified in Maude.

```
fmod PHILOSOPHERS-TABLE is            *** functional module
  protecting NAT .                    *** import a module (natural n.)

  sorts Obj Phil List Table .         *** declare some sorts
  subsorts Obj Phil < List .          *** establish subsort relations

  op (_|_|_) : Obj Nat Obj -> Phil [ctor] .   *** constructor
  ops o ψ    : -> Obj [ctor] .
  op empty   : -> List [ctor] .
  op __      : List List -> List [ctor assoc id: empty] .
  op <_>     : List -> Table [ctor] .

  var L : List . var P : Phil .               *** declare a variable
  eq < ψ L P > = < L P ψ > .

  op initial : -> Table .
  eq initial = < (o | 0 | o) ψ (o | 1 | o) ψ (o | 2 | o) ψ > .
endfm

mod PHILOSOPHERS-DINNER is            *** system module
  protecting PHILOSOPHERS-TABLE .
  var Id : Nat .
  var X  : Obj .
  var L  : List .
  rl [left]    :       ψ (o | Id | X) => (ψ | Id | X) .
  rl [right]   :       (X | Id | o) ψ => (X | Id | ψ) .
  rl [left]    : < (o | Id | X) L ψ > => < (ψ | Id | X) L > .
  rl [release] :         (ψ | Id | ψ) => ψ (o | Id | o) ψ .
endm
```

Terms can be reduced equationally to a normal form using the `reduce` command. Rules can be applied using the `rewrite` and `frewrite` commands, which follow different fixed built-in strategies to choose which rule, which subterm, and which substitution to try first. Finally, `search` allows searching for any terms satisfying some given conditions in all possible rewriting paths from its argument [12, §5.4].

4.1 The strategy language

Effective manipulation of strategies requires expressing them in a convenient syntactical form. Since Maude is a reflective language, strategies have usually been expressed by explicitly applying rules at the metalevel. Because of its low-level, this is an awkward and error-prone method, so a strategy language was proposed [25, 15], conceived as an additional layer above functional and system specification. It has already been used in different contexts, among others [17, 24, 31, 33]. A strategy expression α can be executed to rewrite a term t using the **srewrite** t **using** α command. The language's basic component is rule application

$$\mathit{ruleLabel}\,\texttt{[}x_1\ \texttt{<-}\ t_1\texttt{,}\ \ldots\ \texttt{,}x_n\ \texttt{<-}\ t_n\texttt{]\{}\alpha_1\texttt{,}\ \ldots\ \texttt{,}\ \alpha_m\texttt{\}}$$

where rules are selected by their labels, an optional initial substitution can instantiate both rule sides before matching, and strategies between curly brackets must be provided for rules with rewriting conditions. The other basic construct is the test `match` P `s.t.` C that checks if the subject term matches the pattern P and the condition C is satisfied. Tests come in three flavors: `match` that matches on top only, `xmatch` that can also match fragments by structural axioms, and `amatch` that matches anywhere within the term. These operators are combined by various constructs like

- concatenation α`;`β, to execute α and then β on its results;
- alternation α`|`β, which non-deterministically chooses α or β;
- iteration α`*`, which executes α zero or more consecutive times;
- the constants `idle`, to do nothing, and `fail`, to discard the current execution path;
- the conditional α `?` β `:` γ, with condition α, positive β and negative γ branches, where γ is only executed if α does not produce any result. Otherwise, β is run after any of those.

Notice that concatenation, alternation, `idle`, and `fail` are the counterparts of regular expression constructors. Derived operators are available too, like α`+` $\equiv$ α`;`α`*`, `not(`α`)` $\equiv$ α `? fail : idle`, a normalization operator α`!` $\equiv$ α`* ; not(`α`)`, etc. To control *where* rules are applied, the language counts with the `top(`α`)` operator that restricts rule applications to the top symbol, and with a subterm rewriting operator

`matchrew` $P(x_1, \ldots, x_n)$ `s.t.` C `by` x_1 `using` α_1 `,` $\ldots$ `,` x_n `using` α_n

that matches the subject term against a pattern P, extracts the matched subterms, and rewrites them by means of substrategies α_1, ..., α_n in parallel. Like the `match` operator, three different versions of this operator exist. Finally, the language allows the declaration of named strategies with arguments in separate strategy modules **`smod`** `NAME` **`is`** ... **`endsm`**:

`strat` *slabel* [`:` *parameterTypes*] `@` *subjectType* `.`

Strategies are called *slabel*`(`t_1`,` ... `,`t_n`)` by citing its strategy name and providing the required arguments in a comma-separated list between parentheses, as a typical function invocation. They can be recursive and mutually recursive, and are defined in strategy modules with any number of (potentially conditional) definitions of the form

[`c`]**`sd`** *slabel*`(`*arguments*`)` `:=` *strategyExpression* [**`if`** C] `.`

where the strategy expression can use the variables in the left-hand side and in the condition. All strategy definitions whose left-hand side matches the call term are executed. An example is shown in Listing 2.

Listing 2 Dining philosophers strategy module.

```
smod DINNER-STRAT is
  protecting PHILOSOPHERS-DINNER .

  strats free parity @ Table .

  sd free := all ? free : idle .

  sd parity := (release
    *** The even take the left ψ first
    | (amatchrew L s.t. ψ (o | Id | o) := L /\ 2 divides Id
      by L using left)
    | left[Id <- 0]
    *** The odd take the right ψ first
    | (amatchrew L s.t. (o | Id | o) ψ := L
      /\ not (2 divides Id) by L using right)
```

```
     *** When they already have one, they take the other ψ
     | (amatchrew L s.t. (ψ | Id | o) ψ := L by L using right)
     | (matchrew M s.t. < L (o | Id | ψ) L' > := M
       by M using left[Id <- Id])
     ) ? parity : idle .
endsm
```

The DINNER-STRAT module imports the module PHILOSOPHERS-DINNER (Example 6) that it will control, and defines two recursive strategies. The first one, free, is the recursive application of any rule (all allows the application of any available rule) until it cannot be further applied. It behaves like the built-in rewrite strategy. The other strategy, parity, forces a particular order in which to take the forks, which is alternative for evens and odds, that is, for neighbors. In Section 5.1, we will see properties that are satisfied with parity but not with free.

More details on the language syntax and semantics are provided in [15, 9] and the companion web page, where the complete implementation of the strategy language is available for download.

4.2 An operational semantics for model checking

This section provides the basis to model check systems specified in Maude and controlled by its strategy language. In this situation, it is essential to know which rewriting paths are allowed by the strategy. However, the semantics of a strategy expression applied to a term has usually been given as a set of result terms [15], so that the intermediate execution states remain unknown. A small-step operational semantics is required to observe them all. One has already been given in [9] by means of a rewrite theory transformation. Still and all, it specifies a global strategic search where multiple execution paths advance in parallel, in a way that they cannot be easily isolated. Based on these, we propose a non-deterministic operational semantics whose derivations clearly denote the full rewriting paths allowed by the strategy. Some technical details have been omitted, but are available in [30].

First, we define the *strategy execution states* on which the semantics is defined. Essentially, states consist of a term t in some rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and a stack s of pending strategies and variable contexts, represented by substitutions $\sigma : X \to T_{\Sigma/E}(X)$. However, the subterm rewriting operator and rewriting conditions require executing strategies in nested contexts that are represented by the "subterm" and "rewc" symbols. In summary, execution states are generated by the following grammar where x is a variable, t is a term in $\mathcal{R}$, α is a strategy expression, and ε is the empty word.

$$\begin{aligned}
s &::= \varepsilon \mid \sigma s \mid \alpha s \\
p &::= t \mid \mathrm{subterm}(x : q, \dots, x : q; t) \mid \mathrm{rewc}(p : q, \sigma, C, \alpha \cdots \alpha, \sigma, t, t; t) \\
q &::= p \mathbin{@} s
\end{aligned}$$

Any execution state can be projected to a term by the recursive function $\mathrm{cterm}(t \mathbin{@} s) = t$, $\mathrm{cterm}(\mathrm{subterm}(x_1 : q_1, \dots, x_n : q_n; t) \mathbin{@} s) = t[x_1/\mathrm{cterm}(q_1), \dots, x_n/\mathrm{cterm}(q_n)]$ and finally $\mathrm{cterm}(\mathrm{rewc}(p : q, \sigma, C, \vec{\alpha}, \theta, r, c; t) \mathbin{@} s) = t$. Moreover, every stack designates a variable context by looking at the top-most substitution, $\mathrm{vctx}(\varepsilon) = \mathrm{id}$, $\mathrm{vctx}(\theta s) = \theta$ and $\mathrm{vctx}(\alpha s) = \mathrm{vctx}(s)$ where id is the identity function. States of the form $t \mathbin{@} \varepsilon$ are called *solutions* and represent successful strategy executions.

The semantics is defined by two distinct transition relations: control $\to_c$ and system $\to_s$ steps. The latter represents real transitions in the underlying system, i.e. rule rewrites, while the first does the auxiliary work to make strategies run. Among control transitions, some are devoted to handle alternation and iteration by taking non-deterministic choices,

$$t\,@\,\alpha \mathbin{|} \beta \to_c t\,@\,\alpha \qquad t\,@\,\alpha \mathbin{|} \beta \to_c t\,@\,\beta \qquad\qquad t\,@\,\alpha\texttt{*} \to_c t\,@\,\varepsilon \qquad t\,@\,\alpha\texttt{*} \to_c t\,@\,\alpha\alpha\texttt{*}$$

Concatenation is reduced by a rule $t\,@\,\alpha\,\texttt{;}\,\beta \to_c t\,@\,\alpha\beta$ that pushes α on top of β in the pending strategies stack. The rule for tests simply pops the operator on success

$$t\,@\,(\texttt{match}\ P\ \texttt{s.t.}\ C)\,\theta \to_c t\,@\,\theta \qquad \text{if there is } \sigma \text{ s.t. } \sigma(\theta(P)) = t \text{ and } \sigma(\theta(C)) \text{ holds}$$

where substitutions σ are extended to substitute variables within terms and conditions. The other test flavors have similar rules. There is no rule for the negative case: the execution path arrives to a deadlock state and will later be discarded. The positive case behaves like an `idle`, whose rule is $t\,@\,\texttt{idle} \to_c t\,@\,\varepsilon$. The semantics of conditionals is given by two rules

$$t\,@\,\alpha\ \texttt{?}\ \beta\ \texttt{:}\ \gamma \to_c t\,@\,\alpha\beta \qquad \frac{\text{the derivations from } t\,@\,\alpha\,\theta \text{ are finite and no solution is reached}}{t\,@\,(\alpha\ \texttt{?}\ \beta\ \texttt{:}\ \gamma)\,\theta \to_c t\,@\,\gamma\,\theta}$$

The first rule simply tries the condition followed by the positive branch: in case α does not produce any result, β will not be executed. Then, the second strategy must be triggered to run γ. Notice that the $\to_c$ rule is undecidable in general since the negative branch condition implies deciding whether the derivations from $t\,@\,\alpha\theta$ are all terminating.

Strategy calls are handled with rule instances of the form

$$t\,@\,sl(p_1, \ldots, p_n)\,\theta \to_c t\,@\,\delta\sigma\theta \qquad \text{for any matching } \sigma \text{ of } (t_i)_{i=1}^n \text{ in } (p_i)_{i=1}^n \text{ s.t. } \sigma(C) \text{ holds}$$

for each strategy definition `csd` $sl(t_1$`,` $\ldots$`,` $t_n)$ `:=` δ `if` C (or its unconditional version). When a strategy call finishes, its variable context is popped $t\,@\,\theta \to_c t\,@\,\varepsilon$.

The mission of the execution stack is holding pending strategies and also active call contexts. Only the top element determines the possible next steps, but the current variable context may be determined by a substitution buried by multiple strategies inside the stack. Rules have been defined for states with almost empty stacks, but they can be easily extended from the bottom as long as the variable context is preserved.

$$t\,@\,s = t\,@\,s\,\mathrm{id} \qquad\qquad \frac{t\,@\,s\,\theta \to_\bullet t'\,@\,s'\,\theta}{t\,@\,s\,s_0 \to_\bullet t\,@\,s' s_0} \text{ if } \mathrm{vctx}(s_0) = \theta$$

where $\to_\bullet$ can be replaced by both $\to_s$ and $\to_c$.

The `matchrew` rewrites matched subterms independently via the subterm execution states,

$$\frac{q_i \to_s^c q_i'}{\mathrm{subterm}(\ldots, x_i : q_i, \ldots; t)\,@\,s \to_s^c \mathrm{subterm}(\ldots, x_i : q_i', \ldots; t)\,@\,s}$$

These structured states are created with the rule

$$\begin{aligned} t\,@\,(&\texttt{matchrew}\ P(x_1, \ldots, x_n)\ \texttt{s.t.}\ C\ \texttt{by}\ x_1\ \texttt{using}\ \alpha_1, \ldots, x_n\ \texttt{using}\ \alpha_n)\,\theta \\ &\to_c \mathrm{subterm}(x_1 : \sigma(x_1)\,@\,\alpha_1\,\rho, \ldots, x_n : \sigma(x_n)\,@\,\alpha_n\,\rho, \ldots)\,@\,\theta \end{aligned}$$

for any matching σ of $\theta(P)$ in t such that $\sigma(\theta(C))$ holds, and where $\rho(x) = \sigma(x)$ if $\sigma(x) \neq x$ and $\theta(x)$ otherwise. And they are resolved, once its subterms have arrived to solutions, with

$$\mathrm{subterm}(x_1 : t_1\,@\,\varepsilon, \ldots, x_n : t_n\,@\,\varepsilon; t)\,@\,s \to_c t[x_1/t_1, \ldots, x_n/t_n]\,@\,s$$

Finally, system transitions $\rightarrow_s$ are generated by rule applications. The execution of a maybe conditional rule without rewriting fragments is

$$t \mathbin{@} rl[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]\, \theta \rightarrow_s t[p/\sigma(\rho(r))] \mathbin{@} \theta$$

if for a position p within t and for a rule $l \rightarrow r$, there is a matching σ such that $t_p = \sigma(\rho(l))$ and $\sigma(\rho(C))$ holds, where ρ is the initial substitution that maps x_i to $\theta(t_i)$. If the rule application is surrounded by the `top` modifier, it is only applied on top. For rules with rewriting conditions l `=>` r, substrategies must be provided between curly brackets and these must be used to rewrite its condition fragments. This is achieved using a structured execution state similar to the subterm construct, where the nested state q executes the corresponding strategy in the rewriting fragment initial term.

$$t \mathbin{@} rl[\ldots, x_i \leftarrow t_i, \ldots]\{\alpha_1, \ldots, \alpha_m\}\, \theta \rightarrow_c \operatorname{rewc}(t_r : \sigma(t_l) \mathbin{@} \alpha_1\theta, \sigma, C, \alpha_2 \cdots \alpha_m, \theta, r, c; t)$$

for any rule rl with condition $C_0 \wedge t_l \texttt{=>} t_r \wedge C$ where C_0 is an equational condition, and any matching substitution σ and matching context c such that $\sigma(C_0)$ holds. Like in the previous case, the rule is first instantiated with the given initial substitution. The right-hand side of the rule r is kept to do the actual rewriting once the conditions have been checked:

$$\operatorname{rewc}(p : t' \mathbin{@} \varepsilon, \sigma, C_0, \varepsilon, \theta, r, c; t) \rightarrow_s c(\sigma'(r))$$

where the substitution σ' extends σ by matching t' against $\sigma(p)$ and satisfies the equational condition $\sigma'(C_0)$. If the remaining condition contains more rewriting fragments, these are tried one after another accumulating variable bindings:

$$\operatorname{rewc}(p : t' \mathbin{@} \varepsilon, \sigma, C_0 \wedge t_l \texttt{=>} t_r \wedge C, \alpha\vec{\alpha}, \theta, r, c; t) \rightarrow_c \operatorname{rewc}(t_r : \sigma'(t_l) \mathbin{@} \alpha\, \theta, \sigma', C, \vec{\alpha}, \theta, r, c; t)$$

The rewc state follows the transitions of the inner state,

$$\frac{q \rightarrow_\bullet q'}{\operatorname{rewc}(p : q, \sigma, C, \vec{\alpha}, \theta, r, c; t) \rightarrow_c \operatorname{rewc}(p : q', \sigma, C, \vec{\alpha}, \theta, r, c; t)}$$

However, transitions inside this nested state are always control transitions in the outer one, because they are not applied to the subject term.

Finally, this semantics can be used to define the translation of the strategy expression α to an abstract strategy $E(\alpha)$ in $(T_{\Sigma/E}, \rightarrow^1_{\mathcal{R}})$ where $T_{\Sigma/E}$ is the initial term algebra and $\rightarrow^1_{\mathcal{R}}$ the one-step rewrite relation of the rewrite theory $\mathcal{R}$. The derived relation $\twoheadrightarrow \, = \, \rightarrow_c^* \circ \rightarrow_s$, a single system transition preceded with all the necessary strategic preparation, has the property that $q \twoheadrightarrow q'$ implies $\operatorname{cterm}(q) \rightarrow^1_{\mathcal{R}} \operatorname{cterm}(q')$.

▶ **Definition 7.** *For a strategy expression α and a set of initial states I, we define*

$$\begin{aligned} E(\alpha) :=& \{t \operatorname{cterm}(q_1) \cdots \operatorname{cterm}(q_n) \cdots : t \mathbin{@} \alpha \twoheadrightarrow q_1 \twoheadrightarrow \cdots \twoheadrightarrow q_n \twoheadrightarrow \cdots, t \in I\} \\ & \cup \ \{t \operatorname{cterm}(q_1) \cdots \operatorname{cterm}(q_n) : t \mathbin{@} \alpha \twoheadrightarrow q_1 \twoheadrightarrow \cdots \twoheadrightarrow q_n \rightarrow_c^* t_n \mathbin{@} \varepsilon, t \in I\} \end{aligned}$$

Observe that finite derivations end with solutions, perhaps modulo some control transitions. The strategy $E(\alpha)$ is intensional and can be encoded in the ARS $\mathcal{A}' = (\mathcal{XS}, \twoheadrightarrow, \{t \mathbin{@} \alpha : t \in I\})$ with final states $F = \{q \in \mathcal{XS} : \exists\, t \in T_{\Sigma/E} \quad q \rightarrow_c^* t \mathbin{@} \varepsilon\}$, and $d = \operatorname{cterm}$ the descent function. $\mathcal{XS}$, the set of execution states, is always an infinite set, but we can restrict to the reachable execution states from the initial ones. For model checking to be decidable, the ARS needs to be finite. Some sufficient conditions can be established:

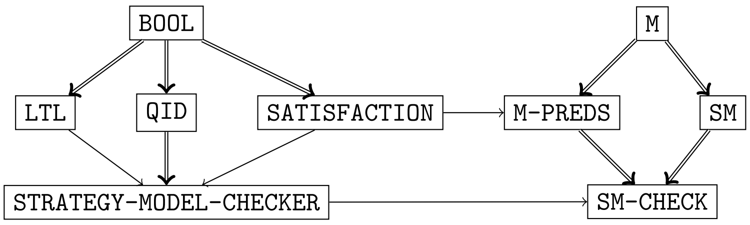


Figure 3 Structure of the strategy model checker modules.

▶ **Proposition 8.** *Given a term $t \in T_{\Sigma/E}$ and a strategy expression α, if the reachable terms from $t \mathbin{@} \alpha$ are finitely many, and the recursive strategy calls are tail recursive and their call arguments only take a finite number of values, $\twoheadrightarrow$ is decidable and the set of reachable execution states is finite.*

Both premises are reasonable, since they bound the number of state and strategy combinations that may appear during execution. This also implies the decidability of the $\twoheadrightarrow$ relation, whose threats are the negative branch of the conditional combinator and rewriting conditions. We understand by *reachable terms* all the elements of $T_{\Sigma/E}$ that occur while executing the strategy, while rewriting both the state and the rewriting conditions of rules. It is easy to observe that expressions without iterations and recursive calls never produce an infinite number of execution states, but they are not usually interesting.

Often, this sufficient condition holds and is checked easily, like in the example strategies of Listing 2. In the `free` and `parity` definitions, all strategy calls are tail recursive and do not take parameters. The reachable terms from `initial` are finitely many since, even by unrestricted rewriting, only 3^3 *tables* are reachable, as shown by counting the possible positions of the forks.

5 The Maude strategy-aware model checker

Following the principles of Section 3 and the strategy language semantics, the new Maude strategy-aware model checker was programmed in C++ as an extension of the already existing explicit-state LTL model checker [16]. Both have a similar interface [12, §10] and are accessed using separate special operators declared in Maude itself.

The built-in modules of the model checker appear on the left of Figure 3. All but `STRATEGY-MODEL-CHECKER` are shared with the standard model checker. The right side of the figure shows the typical structure of the user specification of the model and properties. The user must:

1. Specify the model in a system module `M`, and define strategies to control `M` in a strategy module `SM`.
2. In a protecting[2] extension of `M`, say `M-PREDS`, choose the sort of the model states, making it a subsort of the `State` sort declared in `SATISFACTION`, declare atomic propositions as operators of type `Prop`, and define the satisfaction relation `|=` for all of them.

```
fmod SATISFACTION is
  protecting BOOL . sorts State Prop .
  op _|=_ : State Prop -> Bool .
endfm
```

[2] Protecting means that it does not alter the signature, equations and rules of the types defined in `M`.

```
mod M-PREDS is
  protecting M .
  including SATISFACTION .
  subsort Foo < State .
  op p : -> Prop .
  eq F:Foo |= p = ... .
endm
```

3. Declare a strategy module, say `SM-CHECK`, to combine the model `M` with the property specification in `M-PREDS` and the strategies `SM`. Import `STRATEGY-MODEL-CHECKER` too.

```
smod SM-CHECK is
  protecting M-PREDS .
  protecting SM .
  including STRATEGY-MODEL-CHECKER .
endsm
```

Once this is done, model checking is invoked using the operator

```
op modelCheck : State Formula Qid QidList
  ~> ModelCheckerResult [special (...)] .
```

which receives an initial state, an LTL formula as defined in the `LTL` module [12, §10], and a strategy identifier, which must correspond to a strategy without parameters defined in the module. The last argument is an optional list of *opaque* strategy names: when a strategy in this list is called, instead of the transitions occurring during the strategy execution, the model checker will see direct transitions to its results. This optional feature produces traces that do not fit in the base `M` model, but allows model checking coarse-grain and fine-grain models with little changes.

The result is either `true` if the model satisfies the specification, or a counterexample, expressed as a cycle of rewriting steps and a path to it, if it does not. Additionally, the model checker optionally outputs an extended dump, from which graphical representations of the system automaton and counterexamples can be generated using an auxiliary program. The model checker, the auxiliary program, additional documentation, and various examples can be downloaded at `http://maude.sip.ucm.es/strategies`.

The fundamentals of the strategy-aware model checker are given by the small-step operational semantics of Section 4.2 and the fundamentals of the original model checker. The model controlled by a strategy α from an initial term $t \in T_{\Sigma/E}$ can be encoded in the Kripke structure

$$\mathcal{K} = (\mathcal{XS}, \twoheadrightarrow, \{t \,@\, \alpha\}, AP_\Pi, L_\Pi \circ \mathrm{cterm}),$$

where atomic propositions are defined as in the standard model checker, for Π the signature of `M-PREDS` and D its set of equations,

$$AP_\Pi := \{\ \theta(p(x_1, \ldots, x_n)) \mid p \in \Pi, \theta \text{ ground substitution }\}.$$

The labeling function $L_\Pi : T_{\Sigma/E} \to \mathcal{P}(AP_\Pi)$ is given by

$$L_\Pi([t]) := \{\theta(p(x_1, \ldots, x_n)) \in AP_\Pi \mid (E \cup D) \vdash\ t \vDash \theta(p(x_1, \ldots, x_n)) = \mathit{true}\}.$$

For the model checking to be decidable the conditions in [12, §10.3] for the standard one must be fulfilled, and additionally the reachable execution states must be finitely many.

5.1 Example: dining philosophers

In this section, we resume the dining philosophers problem (Example 6) to model check some properties with different strategies. Although the example is presented with three philosophers, it can be instantiated with many more. We already know that some unwanted situations may appear during the dinner: a philosopher may starve or, even worse, none of them could be able to eat. First, we express the collection of properties “the philosopher n eats” as atomic propositions and prepare the model checker input data.

Listing 3 Atomic proposition for the dining philosophers example.

```
mod DINNER-PREDS is
  protecting PHILOSOPHERS-DINNER .    *** From Example 6 (Listing 1)
  including SATISFACTION .

  subsort Table < State .
  op eats : Nat -> Prop [ctor] .

  var  Id  : Nat .
  vars L M : List .

  eq < L (ψ | Id | ψ) M > |= eats(Id) = true .
  eq < L > |= eats(Id) = false [owise] .        *** otherwise
endm
```

Then, we put together and include the built-in model checker module.

```
smod DINNER-CHECK is
  protecting DINNER-PREDS .
  protecting DINNER-STRAT .        *** From Listing 2
  including STRATEGY-MODEL-CHECKER .
  including MODEL-CHECKER .        *** the standard model checker too
endsm
```

Now, we can check that the property $\Box\,\Diamond\,(\mathit{eats}(0) \vee \mathit{eats}(1) \vee \mathit{eats}(2))$ (no *deadlock*) does not hold for free rewriting, either using the standard model checking or the strategy-aware one with the `free` strategy, but it does hold when using the `parity` strategy.

```
Maude> red modelCheck(initial,
             [] <> (eats(0) \/ eats(1) \/ eats(2))) .
ModelCheckerSymbol: Examined 4 system states.
rewrites: 43 in 4ms cpu (0ms real) (10750 rewrites/second)
result ModelCheckResult: counterexample(
  {< (o | 0 | o) ψ (o | 1 | o) ψ (o | 2 | o) ψ >,'left}
  {< (ψ | 0 | o) ψ (o | 1 | o) ψ (o | 2 | o) >, 'left}
  {< (ψ | 0 | o) (ψ | 1 | o) ψ (o | 2 | o) >,'left},
  {< (ψ | 0 | o) (ψ | 1 | o) (ψ | 2 | o) >,deadlock})

Maude> red modelCheck(initial,
             [] <> (eats(0) \/ eats(1) \/ eats(2)), 'parity) .
StrategyModelCheckerSymbol: Examined 12 system states.
rewrites: 159 in 0ms cpu (2ms real) (~ rewrites/second)
result Bool: true
```

However, `parity` does not guarantee that all of them eat, because the property $\Diamond\,\mathit{eats}(0)$ does not hold. The model checker presents a counterexample, where the philosopher number two takes both forks and releases them in a loop, while the rest keep inactive:

```
Maude> red modelCheck(initial, <> eats(0), 'parity) .
rewrites: 55 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult: counterexample(nil,
  {< (o | 0 | o) ψ (o | 1 | o) ψ (o | 2 | o) ψ >,'left}
  {< (o | 0 | o) ψ (o | 1 | o) (ψ | 2 | o) ψ >,'right}
  {< (o | 0 | o) ψ (o | 1 | o) (ψ | 2 | ψ) >,'release})
```

In order to ensure that all of them eat, the strategy should act as a referee. A succinct and direct solution is fixing turns, like in the following strategy:

```
sd turns(K, N) := left[Id <- K] ; right[Id <- K] ; release ;
             turns(s(K) rem N, N) .
sd turns := turns(0, 3) .
```

If the number of diners is greater, a more parallel version can be written allowing n div 2 philosophers to eat at the same time. By model checking with `turns`, we obtain

```
Maude> red modelCheck(initial,
  [] (<> eats(0) /\ <> eats(1) /\ <> eats(2)), 'turns) .
rewrites: 131 in 0ms cpu (1ms real) (~ rewrites/second)
result Bool: true
```

5.2 Implementation notes

We have programmed the new model checker in C++ as part of a modified version of the Maude interpreter that includes full strategy language support. The implementation reuses both the existing explicit-state LTL model checker and the existing infrastructure for strategic execution [15], which we have completed to support strategy modules and the `matchrew` operator. This infrastructure is based on a collection of *tasks*, which reflect continuations and call frames, and *processes* that are in charge of finding matches, applying rules, etc. The already existing model checker follows the automata-theoretic approach [11, §4] based on testing the emptiness of the language recognized by the synchronous product of the model and the negation of the linear property as Büchi automata. The model automaton is built specifically for the system controlled by the strategy, while the LTL to Büchi automaton translation and the nested depth-first algorithm are reused.

Each model state corresponds to a strategy execution state in the proposed semantics, and stores some identifying information and a list of processes from which successors are obtained on-the-fly when requested. Control operations $\to_c$ are handled as in usual execution, but rule rewrites trigger the commitment of a new state. Different techniques are used to detect cycles and already visited states in order to reuse previous work.

6 Conclusions and future work

Strategies and languages to express them are useful resources to specify restrictions and global control in rewriting systems, following the *separation of concerns* principle. This approach has already been used to specify deduction procedures, semantics of programming languages, chemical and biological processes, ... Such models need to be formally verified and analyzed. In this paper, we show that model checking has a natural definition in this context, and we tell how to effectively model check specifications for the Maude strategy language, by means of a small-step operational semantics that emphasizes rewriting sequences.

This procedure can be applied to other strategy representations. A model checker for systems specified in Maude and controlled by its strategy language has been implemented in C++ as an extension of the existing explicit-state LTL model checker, which can be downloaded from `http://maude.sip.ucm.es/strategies`. It has already been tested with classical examples and its performance is comparable to the original model checker.

The ongoing work comprises the study of branching-time properties and other theoretical aspects, as well as the development of examples to exploit and test the performance of the tool. The model checker can also be improved by providing clearer counterexamples with more information on the strategy execution, and updating the inherited LTL-to-automaton algorithm to more recent and efficient proposals [3, 22].

References

1 Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002. doi:10.1145/585265.585270.

2 Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.

3 Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to Büchi automata translation: Fast and more deterministic. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *LNCS*, pages 95–109. Springer, 2012. doi:10.1007/978-3-642-28756-5_8.

4 Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking Rewriting on Java. In Franz Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007. doi:10.1007/978-3-540-73449-9_5.

5 H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 131. North Holland, 2nd edition, 2014.

6 Massimo Bartoletti, Maurizio Murgia, Alceste Scalas, and Roberto Zunino. Modelling and Verifying Contract-Oriented Systems in Maude. In Santiago Escobar, editor, *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, volume 8663 of *LNCS*, pages 130–146. Springer, 2014. doi:10.1007/978-3-319-12904-4_7.

7 Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An Overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1-4, 1998*, volume 15 of *ENTCS*, pages 55–70. Elsevier, 1998. doi:10.1016/S1571-0661(05)82552-6.

8 Tony Bourdier, Horatiu Cirstea, Daniel J. Dougherty, and Hélène Kirchner. Extensional and Intensional Strategies. In Maribel Fernández, editor, *Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009, Brasilia, Brazil, 28th June 2009*, volume 15 of *EPTCS*, pages 1–19, 2009. doi:10.4204/EPTCS.15.1.

9 Christiano Braga and Alberto Verdejo. Modular Structural Operational Semantics with Strategies. In Rob van Glabbeek and Peter D. Mosses, editors, *Proceedings of the Third Workshop on Structural Operational Semantics, SOS 2006, Bonn, Germany, August 26, 2006*, volume 175(1) of *ENTCS*, pages 3–17. Elsevier, 2007. doi:10.1016/j.entcs.2006.10.024.

10 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008. doi:10.1016/j.scico.2007.11.003.

11 Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. doi:10.1007/978-3-319-10575-8.

12 Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual v2.7.1*, July 2016. URL: `http://maude.lcc.uma.es/manual271/maude-manual.html`.

13 Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007. `doi:10.1007/978-3-540-71999-1`.

14 Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2-3):247–267, 2004. `doi:10.1007/s10009-002-0104-3`.

15 Steven Eker, Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Deduction, Strategies, and Rewriting. In Myla Archer, Thierry Boy de la Tour, and César Muñoz, editors, *Proceedings of the 6th International Workshop on Strategies in Automated Deduction, STRATEGIES 2006, Seattle, WA, USA, August 16, 2006*, volume 174(11) of *ENTCS*, pages 3–25. Elsevier, 2007. `doi:10.1016/j.entcs.2006.03.017`.

16 Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL Model Checker. In Fabio Gadducci and Ugo Montanari, editors, *Proceedings of the Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19-21, 2002*, volume 71 of *ENTCS*, pages 162–187. Elsevier, 2004. `doi:10.1016/S1571-0661(05)82534-4`.

17 Mercedes Hidalgo-Herrero, Alberto Verdejo, and Yolanda Ortega-Mallén. Using Maude and Its Strategies for Defining a Framework for Analyzing Eden Semantics. In Sergio Antoy, editor, *Proceedings of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2006, Seattle, WA, USA, August 11, 2006*, volume 174(10) of *ENTCS*, pages 119–137. Elsevier, 2007. `doi:10.1016/j.entcs.2007.02.051`.

18 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

19 Hélène Kirchner. Rewriting Strategies and Strategic Rewrite Programs. In Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott, editors, *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, volume 9200 of *LNCS*, pages 380–403. Springer, 2015. `doi:10.1007/978-3-319-23165-5_18`.

20 Saul Kripke. A Completeness Theorem in Modal Logic. *Journal of Symbolic Logic*, 24(1):1–14, 1959. `doi:10.2307/2964568`.

21 Orna Kupferman and Moshe Y. Vardi. Memoryful Branching-Time Logic. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 265–274. IEEE Computer Society, 2006. `doi:10.1109/LICS.2006.34`.

22 Weiwei Li, Shuanglong Kan, and Zhiqiu Huang. A Better Translation From LTL to Transition-Based Generalized Büchi Automata. *IEEE Access*, 5:27081–27090, 2017. `doi:10.1109/ACCESS.2017.2773123`.

23 Si Liu, Muntasir Raihan Rahman, Stephen Skeirik, Indranil Gupta, and José Meseguer. Formal Modeling and Analysis of Cassandra in Maude. In Stephan Merz and Jun Pang, editors, *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, volume 8829 of *LNCS*, pages 332–347. Springer, 2014. `doi:10.1007/978-3-319-11737-9_22`.

24 Narciso Martí-Oliet, Miguel Palomino, and Alberto Verdejo. Strategies and simulations in a semantic framework. *Journal of Algorithms*, 62(3-4):95–116, 2007. `doi:10.1016/j.jalgor.2007.04.002`.

25 Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Towards a Strategy Language for Maude. In Narciso Martí-Oliet, editor, *Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004*, volume 117 of *ENTCS*, pages 417–441. Elsevier, 2004. `doi:10.1016/j.entcs.2004.06.020`.

26 José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. `doi:10.1016/0304-3975(92)90182-F`.

27 Fabio Mogavero, Aniello Murano, Giuseppe Perelli, and Moshe Y. Vardi. Reasoning About Strategies: On the Model-Checking Problem. *ACM Transactions in Computational Logic*, 15(4):34:1–34:47, 2014. doi:10.1145/2631917.

28 Martin R. Neuhäußer and Thomas Noll. Abstraction and Model Checking of Core Erlang Programs in Maude. In Grit Denker and Carolyn Talcott, editors, *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1-2, 2006*, volume 176 (4) of *ENTCS*, pages 147–163. Elsevier, 2007. doi:10.1016/j.entcs.2007.06.013.

29 Amir Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. doi:10.1109/SFCS.1977.32.

30 Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo. Model checking strategy-controlled rewriting systems (extended version). Technical Report 02/19, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2019. URL: http://maude.sip.ucm.es/strategies/tr0219.pdf.

31 Gustavo Santos-García and Miguel Palomino. Solving Sudoku Puzzles with Rewriting Rules. In Grit Denker and Carolyn Talcott, editors, *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1-2, 2006*, volume 176 (4) of *ENTCS*, pages 79–93. Elsevier, 2007. doi:10.1016/j.entcs.2007.06.009.

32 Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

33 Alberto Verdejo and Narciso Martí-Oliet. Basic completion strategies as another application of the Maude strategy language. In Santiago Escobar, editor, *Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011*, volume 82 of *EPTCS*, pages 17–36, 2011. doi:10.4204/EPTCS.82.2.