Strategies, model checking and branching-time properties in Maude

Rubén Rubio[a,*], Narciso Martí-Oliet[a,b], Isabel Pita[a], Alberto Verdejo[a]

[a] *Facultad de Informática, Universidad Complutense de Madrid, Spain*
[b] *Instituto de Tecnología del Conocimiento, Universidad Complutense de Madrid, Spain*

Abstract

Rewriting logic and its implementation Maude are a natural and expressive framework for the specification of concurrent systems and logics. Its nondeterministic local transformations are described by rewriting rules, which can be controlled at a higher level using a builtin strategy language added to Maude 3. This specification resource would not be of much interest without tools to analyze their models, so in a previous work, we extended the Maude LTL model checker to verify strategy-controlled systems. In this paper, CTL* and μ-calculus are added to the repertoire of supported logics, after discussing which adaptations are needed for branching-time properties. The new extension relies on some external model checkers that are exposed the Maude models through general and efficient connections, profitable for future extensions and further applications. The performance of these model checkers is compared.

Keywords: Maude, Rewriting strategies, Branching-time properties, Model checking

1. Introduction

Rewriting logic [57, 60] is a natural and expressive framework for the formal specification and analysis of concurrent systems and logics. Their states are described as terms modulo equations and structural axioms, and their transitions are expressed using rewriting rules. Executing a rewrite system consists of the successive application of a rule in a matching position of the term, both chosen nondeterministically and independently at each step, yielding potentially many evolutions. The spatial and temporal locality of rules is the cornerstone of the natural and simple representation of concurrency, but it is sometimes convenient to tame this nondeterminism and capture the global behavior of the system or other kinds of restrictions. For example, the terms and deduction rules of an inference system can be expressed as a rewrite theory and be proven sound, but only a careful application of these rules will lead to the desired deductions. This idea is enunciated in the Kowalski's motto *Algorithm = Logic + Control* [46] and developed in the

*Corresponding author
Email addresses: rubenrub@ucm.es (Rubén Rubio), narciso@ucm.es (Narciso Martí-Oliet), ipandreu@ucm.es (Isabel Pita), jalberto@ucm.es (Alberto Verdejo)

Accepted authors' manuscript of the article published at J. Logic. Algebr. Program 123
DOI: 10.1016/j.jlamp.2021.100700

Lescanne's *Rule + Control* approach [51], which promotes the separation of the concerns of rules and their control. This is the purpose of strategies, which have been used in formal specification languages like ELAN [11], TOM [9], Stratego [15], and more recently Porgy [35] for graph rewriting. Unlike strategies usually considered for the λ-calculus [10] and abstract rewriting [5, 77], these are called *programmable strategies* because they are represented syntactically as arbitrary complex programs.

Maude [24, 25] is a specification language based on rewriting logic and an interpreter for executing and analyzing its specifications. Strategies have been used in Maude since its beginnings [26, 27] using reflection. However, reflective programs are verbose and difficult to understand for those not used to them, so an object-level strategy language was proposed, prototyped and tested, and finally implemented in Maude 3 [55]. Based on that experience and on earlier languages like ELAN and Stratego, its design puts special emphasis on separating rules from strategies, so that different strategies can be compositionally specified to control the same rewriting system easily. This new resource for writing formal specifications would be less attractive to Maude users if there were fewer means to work with strategy-controlled models than with standard ones. Hence, we extended the builtin Maude LTL model checker [33] to support them [71]. It has already been given various applications [4, 74, 73].

In this paper, we address model checking for strategy-controlled systems against branching-time properties, by first discussing the problem in abstract terms and then particularizing them to strategy-controlled Maude specifications, as we did for linear-time properties. In the general setting, a natural notion of satisfaction arises by considering only the subtree of executions allowed by the strategy when checking branching-time properties, in the same way we consider only the subset of allowed executions when checking linear-time properties. As a practical procedure for model checking according to this definition, we suggest transforming the model to incorporate the restrictions imposed by the strategy, which allows checking virtually any logic supported in the uncontrolled system using its standard algorithms. This is similar to our previous approach for linear-time properties, but this transformation must preserve the branching structure of the original model, for which a certain bisimilarity relation will be required. In order to check linear-time formulae on systems controlled by the Maude strategy language, we provided it with a small-step operational semantics that determines which are exactly the executions described by a strategy expression and is the base to construct the transformed model where those properties can be checked. However, we will see that the previous transformation is not appropriate and consistent for branching-time properties, and some additional adaptations are required. Following these principles, we now support logics like CTL, CTL*, and μ-calculus by means of external model checkers. All these logics are implemented in the language-independent model checker LTSmin [45], for which we have developed a plugin with on-the-fly access to the models in the C++ implementation of Maude. Other model checkers are also available as backends like NuSMV [21], the `pyModelChecking` library [18], Spot [31], and a custom μ-calculus implementation. All these backends are accessed uniformly using an extensible model-checking tool `umaudemc` implemented using a `maude` Python library we have developed [69]. The new model checkers can also be applied to standard Maude specifications for which there was no relevant support for branching-time properties thus far. Moreover, the connections developed for this work can be applied for other purposes, like visualization and other types of analysis.

Comparison with the workshop paper. This article extends the workshop paper [75], introducing the extensible architecture of `umaudemc` and its connection with other model checkers in addition to LTSmin. The presentation has been improved with further details, the performance of the model checkers has been compared, and related work is discussed.

Structure of the paper. Section 2 reviews some precedents required to follow the rest of the paper. Section 3 describes how model checking is understood for strategy-controlled systems in general. Section 4 explains how strategy-controlled systems are specified and model checked in Maude, while Section 5 discusses the specific problems that appear when checking branching-time properties and how they are solved. Section 6 introduces the connections to external model checkers, which are evaluated in Section 7. Related work is reviewed in Section 8. All the material, including the LTL and branching-time model checkers, their documentation and source code, the examples in this paper and many more, is available online [32].

2. Preliminaries

Let us recall some basic concepts and notation about strategies, rewriting logic and model checking, which will be extensively used along the paper. Informed readers may safely skip some sections. The Maude strategy language is also introduced together with the small-step operational semantics on which our model checker is based.

2.1. Strategies and transition systems

A *labeled transition system* (LTS) $\mathcal{A} = (S, A, R)$ is a set of states S, a set of labels or actions A, and a labeled binary relation $R \subseteq S \times A \times S$ on the states. Sometimes we consider plain transition systems $\mathcal{A} = (S, R)$ without transition labels, where $R \subseteq S \times S$ is a usual binary relation. They can be seen as a particular case of labeled transition systems with a single label τ for all transitions, so most claims about these are valid for those.[1] Arrows are often used to denote the transition relation, and we write $s \to^a s'$ for $(s, a, s') \in R$ and $s \to s'$ if $s \to^a s'$ for some $a \in A$. We call $s \to s'$ an *execution step* in $\mathcal{A}$, s' a *successor* of s, and an *execution* in $\mathcal{A}$ is a finite or infinite sequence of states $s_0 \to^{a_1} s_1 \to^{a_2} \cdots \to^{a_n} s_n$ linked by the relation. For convenience, we represent executions as finite words $s_0 a_1 s_1 \cdots a_n s_n \in (S \cup A)^*$ or infinite words $s_0 a_1 s_1 a_2 s_2 \cdots \in (S \cup A)^\omega$ alternating states and actions. In the unlabeled case, actions are dropped from the words $s_0 s_1 \cdots s_n$ or $s_0 s_1 \cdots$. Let $\Gamma^*_{\mathcal{A}} \subseteq S^*$, $\Gamma^\omega_{\mathcal{A}} \subseteq S^\omega$ and $\Gamma_{\mathcal{A}} \subseteq S^\infty$ designate the set of all finite and infinite executions of $\mathcal{A}$, and the union of both. A subscript $s \in S$ will be added to these sets $\Gamma_{\mathcal{A},s}$ to indicate that only executions starting from this state are included.

2.1.1. Strategies

In this general context, strategies have been defined in different ways in the literature [13], from which we consider two simple characterizations that we will use almost interchangeably:

[1] Labeled transition systems can also be embedded in plain transition system by pushing the actions on the states.

1. *Extensional strategies* are subsets $E \subseteq \Gamma_{\mathcal{A}}$ of allowed executions of $\mathcal{A}$.
2. *Intensional strategies* are partial functions $\lambda : (S \cup A)^+ \to \mathcal{P}(A \times S)$ that select the possible next steps to continue an execution $w \in (S \cup A)^+$ based on its history, where the states $(a, s') \in \lambda(ws)$ must always satisfy $s \to^a s'$. In the unlabeled case, this can be simplified to $\lambda : S^+ \to \mathcal{P}(S)$.

The second definition is widely used in games and other verification logics [61, 1], but the first one is simpler and more expressive. In fact, there is an extensional strategy $E(\lambda) \coloneqq \{s_0 a_1 s_1 \cdots \in (S \cup A)^\infty : (a_{k+1}, s_{k+1}) \in \lambda(s_0 \cdots a_k s_k)\}$ for every intensional strategy λ, but the converse is not true. Even if an intensional strategy λ_E can be defined from an extensional one E, $\lambda_E(w) \coloneqq \{(a, s) \in A \times S : wasw' \in E, w' \in (S \cup A)^\infty\}$, some information is lost and the inclusion $E \subseteq E(\lambda_E)$ may be strict. While an extensional strategy can selectively allow finite executions, in intensional strategies all the prefixes of accepted executions are accepted, because there is no way to indicate that an execution is complete. However, this will not be a problem for model checking, because we usually assume that all executions are nonterminating, or otherwise we complete the finite executions by repeating their last states forever and discard the incomplete ones. Another limitation is that extensional strategies are not necessarily closed while intensional strategies are; for example, the first type may allow executions of the form $a^n b^\omega$ for all $n \geq 0$ but not a^ω, while the second type cannot achieve that. This means that fairness restrictions cannot be represented in the strategy, but this is a reasonable assumption for practical executable strategies, and those restrictions can be treated apart as we suggest for future work.

In summary, we will represent strategies both intensionally and extensionally, using the most convenient representation in each occasion.

2.1.2. Execution trees

Since this work is focused on branching-time properties, we should see the executions of a transition system as a tree instead of as a collection of unrelated execution paths. The *execution tree* of $\mathcal{A}$ from a given state $i \in S$ is the tree whose root is i and whose nodes are states with all their successors as children. From the graph-theoretic point of view, this can be formalized as the graph $(\Gamma^*_{\mathcal{A},i}, \{(ws, wsas') : s \to^a s', w \in (S \cup A)^*\})$, which is acyclic and connected. Each vertex ws consists of the current execution state s and its history w down to the root, whose purpose is disambiguating repeated states that may appear at different branches or depths. However, when tree diagrams are drawn, the history is omitted as it can be inferred from the context. Notice that a strategy λ determines a subtree of that execution tree, namely $(\Gamma^*_{\mathcal{A},i}, \{(w, was) : (a, s) \in \lambda(w)\})$. In the unlabeled case, vertices are only words on states, as usual.

2.2. Rewriting logic

Rewriting logic models change by means of rewriting rules operating on the algebraic terms of an equational logic. These terms are built out of an order-sorted signature given by a set of sorts S and an $S^* \times S$-indexed collection Σ of function symbols $f : s_1 \cdots s_n \to s$. Sorts are related by a partial order $s_1 < s_2$ that means subsort inclusion. Given an S-sorted family of variables X, we consider the set of all terms $T_\Sigma(X)$ on these variables, and substitutions $\sigma : X \to T_\Sigma(X)$ as sort-preserving assignments from variables to terms. A substitution can be recursively extended to a function $\overline{\sigma} : T_\Sigma(X) \to T_\Sigma(X)$ that replaces

all occurrences of the variables in a term, and the composition $\sigma_2 \circ \sigma_1$ of two substitutions is defined $(\sigma_2 \circ \sigma_1)(x) := \overline{\sigma_2}(\sigma_1(x))$. It satisfies $\overline{\sigma_2 \circ \sigma_1} = \overline{\sigma_2} \circ \overline{\sigma_1}$ in the usual functional sense.[2] The line over the extension is usually omitted. Terms without variables $T_\Sigma := T_\Sigma(\emptyset)$ are called *ground terms*.

In a membership equational logic [12] (Σ, E) there are two classes of atomic sentences, conditional *equations* and sort *membership axioms*. Their optional conditions are in turn equations and sort membership formulas that yield Horn clauses of the form:

$$t = t' \quad \text{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \qquad\qquad t : s \quad \text{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j$$

where $t : s$ states that t has sort s. These statements induce an equality relation $=_E$ that identifies different terms up to provable equality by E. The initial term algebra $T_{\Sigma/E}$ is the quotient of the ground terms T_Σ modulo this relation. Although its elements $[t]$ are equivalence classes, we will usually write simply t when no confusion is possible.

A *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ is a membership equational logic theory (Σ, E) with a set R of rewriting rules. Possibly conditional rewriting rules have the form:

$$l \Rightarrow r \quad \text{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \wedge \bigwedge_k w_k \Rightarrow w_k'$$

The application of a rule to a term t is the replacement of an instance of l in some subterm of t by r instantiated accordingly, if the condition holds. Conditions of the third type are named *rewriting conditions*, and they are satisfied if the instance of each w_k can be rewritten by the rules in zero or more steps to match w_k'. Unlike equations, which are required to be confluent and terminating to make the evaluation of equality decidable and efficient, rules can yield nonterminating and diverging computations.

Given a set of labels A and an assignment $R \to A$ of a label to each rule in the logic, a rewriting system can be seen as a labeled transition system $(T_{\Sigma/E}, A, \to_R^1)$ whose steps $\to_R^1$ are the single application of a rule to any term in the class and whose actions A are the labels of the rules.[3] Strategies can be considered in this LTS.

Maude [24] is a specification and programming language, where equational and rewrite theories are described compositionally using a notation that does not differ much of the previous mathematical language. These specifications can be executed and analyzed with different commands included in the Maude interpreter and other tools. Further details are available in the Maude manual [24] and examples are included in the following sections.

2.3. The Maude strategy language

The Maude strategy language [24, §10] is used to control the application of rules by expressing rewriting strategies. Strategy expressions, whose syntax is specified by the α symbol in the grammar below, combine explicit application of rules with a small set of

[2] Funtional composition $f \circ g$ is understood in the order $(f \circ g)(x) = f(g(x))$.

[3] The transitions of a rewriting system can also be labeled by *proof terms* including all the details of the particular rule application, like its context and substitution [57]. However, rule labels are enough for our purposes.

programming constructs.

$$\begin{aligned}
\alpha &::= \beta \mid \texttt{top}(\beta) \mid \texttt{idle} \mid \texttt{fail} \mid \texttt{match}\ P\ \texttt{s.t.}\ C \mid \alpha\,;\alpha \mid (\alpha \mid \alpha) \mid \alpha\,* \mid\ \alpha\,?\,\alpha : \alpha \\
&\mid\ \texttt{matchrew}\ P\ \texttt{s.t}\ C\ \texttt{by}\ x\ \texttt{using}\ \alpha,\ \ldots, x\ \texttt{using}\ \alpha \mid \mathit{slabel} \mid \mathit{slabel}(\vec{t}) \\
&\mid\ \alpha + \mid \alpha\ ! \mid \alpha\ \texttt{or-else}\ \alpha \mid \texttt{test}(\alpha) \mid \texttt{try}(\alpha) \mid \texttt{not}(\alpha) \\
\beta &::= \mathit{rlabel} \mid \mathit{rlabel}[\rho] \mid \mathit{rlabel}\{\vec{\alpha}\} \mid \mathit{rlabel}[\rho]\{\vec{\alpha}\} \mid \texttt{all}
\end{aligned}$$

The first two rows are the essential part of the strategy language, including the rule application strategies under the β symbol, since the combinators in the third row can be defined in terms of those of the first two. The meaning of a strategy expression is usually described by the set of terms that its nondeterministic application on a given initial term produces. However, we are also interested in the intermediate states of the strategy-controlled rewriting and in the infinite rewriting sequences allowed by the strategy, which are crucial for model checking. Consequently, we have described the meaning of strategy expressions using a nondeterministic small-step operational semantics [71]. Its steps are defined on *execution states* $q \in \mathcal{XS}$ whose most basic form are pairs $t \,@\, \alpha_1 \cdots \alpha_n$ where t is the term being rewritten and $\alpha_1, \ldots, \alpha_n$ are the pending strategies to be executed, in that order. However, additional structure will be added to these sets as required by some specific combinators. In any case, the *subject term* being rewritten can be identified from an execution state with the projection cterm : $\mathcal{XS} \to T_\Sigma(X)$, whose definition on the simpler states is $\mathrm{cterm}(t \,@\, z) = t$. States $t \,@\, \varepsilon$ with an empty stack are called *solutions*, since no more work is pending and t can be seen as a result of the strategic computation. Substitutions $\theta : X \to T_\Sigma(X)$ may also be pushed to this execution stack, and they determine the value of the variables in the strategy expressions to their left. In the following, θ will always refer to the leftmost substitution of the current stack z, or to the identity function if there is none. The following are the core combinators of the language:

- The rule application strategy $\mathit{rlabel}[x_1 \texttt{<-} t_1\,, \ldots, x_n \texttt{<-} t_n]\{\alpha_1, \ldots, \alpha_m\}$ executes a single rewrite on the subject term using any rule with label *rlabel* under some optional restrictions, and produce all possible such rewrites as a result. For rules without rewriting conditions, its semantics is straighforward

$$t \,@\, \mathit{rlabel}[x_1 \texttt{<-} t_1\,, \ldots, x_n \texttt{<-} t_n]\ z \to_s t' \,@\, z$$

  The mapping from x_i to t_i between brackets is an optional substitution that is applied to both sides of the rule and its condition before matching, and whose values t_i are previously instantiated with the leftmost substitution θ in the stack z. For a rule with m rewriting conditions, exactly m strategies must be provided between curly brackets to control their evaluation. An additional execution state is introduced to hold a nested state for the rewriting fragment and other information like the matching substitution σ, the remaining rule condition C' and the remaining strategies, the right-hand side of the rule r, the context c where it is applied, and the original term t.

$$\begin{aligned}
&t \,@\, \mathit{rl}[x_1 \texttt{<-} t_1, \ldots, x_n \texttt{<-} t_n]\{\alpha_1, \ldots, \alpha_k\}\ z \\
&\quad \to_c \mathrm{rewc}(p_1 : \sigma(l_1) \,@\, \alpha_1\theta, \sigma, C', \alpha_2 \cdots \alpha_k, \theta, r, c; t) \,@\, z
\end{aligned}$$

  Notice that we have written $\to_s$ in the previous rule but $\to_c$ in the current one, because we want to distinguish which steps are *system* steps that apply rewrite

rules to the terms and which are *control* steps that only advance the execution of the strategy. In this case, the term is not actually rewritten until the last rewriting fragment has been solved and the remaining condition C_0 is empty or purely equational.

$$\mathrm{rewc}(p : t' \mathbin{@} \varepsilon, \sigma, C_0, \varepsilon, r, c; t) \mathbin{@} z \to_s c(\sigma'(r)) \mathbin{@} z$$

Meanwhile, the substitution σ is extended with the values yielded by matching the free variables of the target of each rewriting fragment with their solutions. The next rewriting fragment is then executed after evaluating the equational condition between them.

$$\begin{aligned}&\mathrm{rewc}(p : t' \mathbin{@} \varepsilon, \sigma, C_0 \wedge l \mathrel{=>} p' \wedge C, \alpha\vec{\alpha}, \theta, r, c; t) \mathbin{@} z \\ &\quad\to_c \mathrm{rewc}(p' : \sigma'(l) \mathbin{@} \alpha\,\theta, \sigma', C, \vec{\alpha}, \theta, r, c; t) \mathbin{@} z\end{aligned}$$

The subsearch is advanced by applying the semantics recursively. However, both control and system steps in the subsearch are seen as control steps of the whole state, since no rewrite is applied to the subject term. In effect, we define $\mathrm{cterm}(\mathrm{rewc}(p : q, \sigma, C, \vec{\alpha}, \theta, r, c; t)) = t$.

$$\frac{q \to_\bullet q'}{\mathrm{rewc}(p : q, \sigma, C, \vec{\alpha}, \theta, r, c; t) \mathbin{@} z \to_c \mathrm{rewc}(p : q', \sigma, C, \vec{\alpha}, \theta, r, c; t) \mathbin{@} z}$$

Rules are applied anywhere by default, but matching can be limited to the topmost position by surrounding the strategy with **top**.

- Tests check whether the subject term matches the pattern P satisfying the equational condition C.

$$t \mathbin{@} \textsf{match}\ \ P\ \ \textsf{s.t.}\ \ C\ z \to_c t \mathbin{@} z \qquad \text{if } t \text{ matches } \theta(P) \text{ and satisfies } \theta(C)$$

The execution only advances if the test succeeds, and the term is not changed. The initial keyword can be changed to **amatch** to match anywhere, or to **xmatch** to match with extension for structural axioms (see [24, § 4.8]).

- Strategies can be combined with a series of operators like concatenation $\alpha\,;\beta$ that applies β on every result of α,

$$t \mathbin{@} (\alpha\,;\beta)\ z \to_c t \mathbin{@} \alpha\ \beta\ z,$$

the union $\alpha \mid \beta$ that nondeterministically chooses between α and β,

$$t \mathbin{@} (\alpha \mid \beta)\ z \to_c t \mathbin{@} \alpha\ z \qquad\qquad t \mathbin{@} (\alpha \mid \beta)\ z \to_c t \mathbin{@} \beta\ z,$$

and the iteration $\alpha*$ that repeatedly executes α a nondeterministic number of times,

$$t \mathbin{@} (\alpha*)\ z \to_c t \mathbin{@} \alpha\ (\alpha*)\ z \qquad\qquad t \mathbin{@} (\alpha*)\ z \to_c t \mathbin{@} z.$$

Together with the constants **idle** and **fail**, whose results are always the initial or no term at all respectively,

$$t \mathbin{@} \textsf{idle}\ z \to_c t \mathbin{@} z \qquad \text{and no rule for } \textsf{fail},$$

this family of combinators resembles those of regular expressions.

- The conditional operator $\alpha\,?\,\beta : \gamma$ that behaves like $\alpha;\beta$ if the condition α produces any result,

$$t \mathbin{@} (\alpha\,?\,\beta : \gamma)\ z \to_c t \mathbin{@} \alpha\ \beta\ z,$$

but evaluates to the results of the negative branch γ if α does not produce any,

$$t \mathbin{@} (\alpha\,?\,\beta : \gamma)\ z \to_c t \mathbin{@} \gamma\ z.$$

This latter rule is only applied if the successors of $t \mathbin{@} \alpha\ \theta$ are finitely many and none is a solution. In general, we say that a strategy *fails* if it does not produce any result.

- The combinator **matchrew** P **s.t.** C **by** x_1 **using** α_1**,** $\ldots$**,** x_n **using** α_n allows rewriting selected subterms of the subject term. The subterms matching the distinct variables x_k in the pattern P are rewritten according to the corresponding strategies α_k in parallel, using the rules

$$\begin{aligned} &t \mathbin{@} \textbf{matchrew}\ P\ \textbf{s.t.}\ C\ \textbf{by}\ x_1\ \textbf{using}\ \alpha_1, \ldots, x_n\ \textbf{using}\ \alpha_n\ z \\ &\quad \to_c \mathrm{subterm}(x_1 : \sigma(x_1) \mathbin{@} \alpha_1\ \sigma, \ldots, x_n : \sigma(x_n) \mathbin{@} \alpha_n\ \sigma; \sigma_{-\{x_1,\ldots,x_n\}}(P)) \mathbin{@} z \end{aligned}$$

for any matching substitution σ for t in $\theta(P)$ satisfying $\theta(C)$, and

$$\frac{q_i \to_\bullet q_i'}{\mathrm{subterm}(\ldots, x_i : q_i, \ldots; t) \mathbin{@} z \to_\bullet \mathrm{subterm}(\ldots, x_i : q_i', \ldots; t) \mathbin{@} z}$$

where $\bullet$ can be either s or c, so that system (control) steps in a substate are system (control) steps on the whole state. Since the substates of subterm rewrite subterms of the subject term, it is natural that their system steps are system steps of the whole state, like rewrites in a subterm are rewrites in the whole term. In fact, the current term of a subterm state is defined recursively as

$$\mathrm{cterm}(\mathrm{subterm}(x_1 : q_1, \ldots, x_n : q_n; t) \mathbin{@} z) = t[x_1/\mathrm{cterm}(q_1), \ldots, x_n/\mathrm{cterm}(q_n)]$$

Finally, the results of a **matchrew** are the reassembled combinations of their solutions,

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

There are **amatchrew** and **xmatchrew** variants like for tests.

- Finally, it is possible to give names to strategy expressions and define them in strategy modules. They should be declared with the signature of the arguments they receive, and with the sort s where they are intended to be applied.

```
strat sname : s1 ... sn @ s .
```

Strategies are defined using conditional or unconditional strategy definitions that assign strategy expressions to those names.

```
sd sname(p1, ..., pn) := α .
csd sname(p1, ..., pn) := α if C .
```

Conditions C in strategy definitions share their syntax with equational conditions as explained in Section 2.2. These named strategies are called by writing their names followed by a comma-separated list of arguments between parentheses, if any,

$$t \,@\, slabel(t_1, \ldots, t_n)\, z \to_c t \,@\, \delta\, \sigma\, z,$$

and the righthand side δ of any definition whose lefthand side matches that call will be executed with the matching substitution σ giving value to its variables. Recursive and mutually recursive definitions are allowed, increasing the expressive power of the language.

There are more combinators that can be derived from the previous, for example, the α `or-else` β combinator, defined as α `?` `idle` `:` β, that executes β only if α fails.

In order to identify the rewriting paths that are allowed by a strategy, we define the relation $\twoheadrightarrow := \to_s \circ \to_c{}^*$ that executes a single system step preceded by as many control steps as required. Clearly, $q \twoheadrightarrow q'$ implies $\mathrm{cterm}(q) \to_R^1 \mathrm{cterm}(q')$, so the projections of the executions of this relation are actual rewriting paths. Consider the sets of complete finite and infinite executions of a strategy α from an initial term t, where $\twoheadrightarrow^a$ is a $\twoheadrightarrow$ step whose final system transition $\to_s$ applies a rule with label a,

$$\mathrm{Ex}^*(\alpha, t) := \{q_0 a_1 q_1 \cdots a_n q_n \,:\, q_0 = t \,@\, \alpha, q_k \twoheadrightarrow^{a_{k+1}} q_{k+1}, q_n \to_c^* t' \,@\, \varepsilon, t' \in T_\Sigma(X)\}$$
$$\mathrm{Ex}^\omega(\alpha, t) := \{q_0 (a_k q_k)_{k=1}^\infty \,:\, q_0 = t \,@\, \alpha, q_k \twoheadrightarrow^{a_{k+1}} q_{k+1}\}$$

Only those finite executions ending in a state where a solution can be reached by control steps are included. The extensional strategy denoted by α is then by definition

$$E(\alpha) := \bigcup_{t \in T_\Sigma(\emptyset)} E(\alpha, t) \quad \text{where } E(\alpha, t) := \mathrm{cterm}(\mathrm{Ex}^*(\alpha, t)) \cup \mathrm{cterm}(\mathrm{Ex}^\omega(\alpha, t))$$

where the projection cterm is naturally extended to words (leaving actions untouched) and languages. This strategy is intensional by definition.

In the Maude interpreter, a command `srewrite` t `using` α is available for rewriting using a strategy. It shows the results of the strategic rewriting, i.e. the last terms of its finite executions $\mathrm{Ex}^*(\alpha, t)$.

2.4. Model checking

Model checking [23] is an automated verification technique based on an exhaustive examination of the executions of a model to prove or refute properties of its dynamic behavior. Models are usually represented as transition systems whose states are annotated with *atomic propositions*, in terms of which the desired properties are expressed. Such construct receives the name of *Kripke structure* $\mathcal{K} = (S, \to, I, AP, \ell)$ where $(S, \to)$ is a labeled or unlabeled transition system, $I \subseteq S$ is a set of initial states, AP is the set of atomic propositions, and $\ell : S \to \mathcal{P}(AP)$ is the *labeling function* that declares which atomic properties are satisfied in each state. For simplicity, it is usually assumed that the transition relation $\to$ is *total*, i.e. that every state has a successor, and only infinite executions are considered. If it were not, we could apply the typical *stuttering extension* that repeats the last state of finite executions forever.

Properties are expressed using *temporal logics* with temporal operators to describe how atomic propositions must occur in time, which are usually separated into two classes [48]:

- *Linear-time* properties are universal properties satisfied by every possible execution of the system. In other words, time is seen as a line where the next step is already determined. The main example is Linear Temporal Logic [68] (LTL) and its multiple extensions.

- *Branching-time* properties reason about the whole execution tree, where multiple futures can be available at any moment. Well-known examples are the Computational Tree Logic [22] (CTL), and the more general CTL* [34] that includes both LTL and CTL.

Another classification distinguishes *state-based* and *action-based* properties [29], depending on whether formulae refer to propositions of the states or the actions of the models. However, both kinds of properties can be considered together, like in μ-calculus [47] and the Temporal Logic of Rewriting [59].

The semantics of temporal logics is usually defined by means of satisfaction relations $\mathcal{K}, s \vDash \varphi$. In the case of linear-time properties, the satisfaction of a formula φ is reduced to its satisfaction $\mathcal{K}, \pi \vDash \varphi$ for all the executions $\pi \in \Gamma^{\omega}_{\mathcal{K},s}$ of the system. We could say that a linear-time property accepts or rejects words, while a branching-time one does so with trees. The *model-checking problem* consists of deciding whether this satisfaction relation holds for a given model and property.

We conclude this section by recalling the notion of bisimulation between Kripke structures [23, § 26.3.1]. Many logics, including the ones implemented in this paper, CTL* and μ-calculus, satisfy the same properties in structures related by bisimulation.

Definition 1. *Given two (labeled) Kripke structures* $\mathcal{K}_1 = (S_1, A, R_1, I_1, AP, \ell_1)$ *and* $\mathcal{K}_2 = (S_2, A, R_2, I_2, AP, \ell_2)$*, a bisimulation is a relation* $B \subseteq S_1 \times S_2$ *such that if* $(s_1, s_2) \in B$ *then*

- $\ell_1(s_1) = \ell_2(s_2)$,
- *for every action* a *and state* s_1' *such that* $(s_1, a, s_1') \in R_1$*, there is some* $s_2' \in S_2$ *such that* $(s_2, a, s_2') \in R_2$ *and* $(s_1', s_2') \in B$.
- *the symmetric condition, with* s_2 *in the role of* s_1 *and so on.*

If the Kripke structures are not labeled, the same definition is valid by the usual embedding. Two states $s_1 \in S_1$ and $s_2 \in S_2$ are *bisimilar* if there is a bisimulation relation B such that $(s_1, s_2) \in B$. Two Kripke structures $\mathcal{K}_1$ and $\mathcal{K}_2$ as above are *bisimilar* if for every initial state $s_1 \in I_1$ there is a bisimilar initial state $s_2 \in I_2$ and vice versa.

3. Model checking strategy-controlled systems

Given a (labeled) transition system or Kripke structure $\mathcal{K} = (S, A, R, I, AP, \ell)$ and a strategy $E \subseteq \Gamma_{\mathcal{K}}$, we say that $(\mathcal{K}, E)$ is a strategy-controlled system. In a previous work [71], we have already discussed what should be understood for the satisfaction of a linear-time property by a strategy-controlled system. Looking at strategies as subsets of the executions of the original model, the notion for linear-time properties is natural and inexorable, properties should only be checked on those allowed executions.

Definition 2 ([71, Definition 2])**.** *Let φ be a linear-time formula, $(\mathcal{K}, E) \vDash \varphi$ if $\mathcal{K}, \pi \vDash \varphi$ for all $\pi \in E$.*

A similar definition could be proposed for branching-time properties, since these are checked on trees and strategies can be seen as subtrees of the execution tree of the original Kripke structure, as explained in Section 2.1.2. However, the definitions of branching-time logics do not usually mention trees explicitly, so we resort to an auxiliary Kripke structure to obtain a clear definition.

Definition 3 (unwinding)**.** *Given a Kripke structure $\mathcal{K}$ and a strategy λ, the unwinding $\mathcal{U}(\mathcal{K}, \lambda)$ of $\mathcal{K}$ according to λ is the Kripke structure $((S \cup A)^+, A, U, I, AP, \ell \circ \mathrm{last})$ where $(w, a, was) \in U$ if $(a, s) \in \lambda(w)$ and $\mathrm{last}(ws) = s$ for all $w \in (S \cup A)^*$.*

The unwinding of a transition system is a well-known concept [77], but in this case only the executions allowed by the strategy are included. As a graph, it is no other than the execution subtree corresponding to the strategy λ. In case the underlying transition system is unlabeled, the action labels can be removed from the previous definition. We define the satisfaction of a branching-time property by a strategy-controlled system as the satisfaction in the unwinding:

Definition 4. *Let φ be a branching-time formula, $(\mathcal{K}, E(\lambda)) \vDash \varphi$ if $\mathcal{U}(\mathcal{K}, \lambda) \vDash \varphi$.*

This definition coincides with the previous one on linear-time properties, because the executions of the unwinding projected by last are exactly those of the strategy. However, it does not have direct practical application since the Kripke structure $\mathcal{U}(\mathcal{K}, \lambda)$ is not finite. Fortunately, many logics are invariant by bisimulation, and we can try to find a bisimilar Kripke structure where the standard model-checking algorithms can be applied to decide the satisfaction of φ. The following theorem claims that this is always possible if the language $E(\lambda)$ is ω-regular, which is a quite general requirement in the context of model checking.

Theorem 1. *Given an intensional strategy λ, there is a finite Kripke structure $\mathcal{K}'$ bisimilar to $\mathcal{U}(\mathcal{K}, \lambda)$ if $E(\lambda)$ is ω-regular. The converse does not hold, but in that case $\ell(E(\lambda))$ is ω-regular.*

This is the program we will adopt regarding the Maude strategy language: finding a finite (so that model checking is decidable) Kripke structure bisimilar (so that the satisfaction of temporal properties is preserved) to the unwinding (to match Definition 4). In any case, the denotation of strategy expressions in Section 2.3 via the small-step operational semantics gives all the ingredients for Definition 4, so we have already unambiguously established whether a branching-time property is satisfied in a Maude specification with strategies. Whenever this denotation is ω-regular, Theorem 1 tells that the plan depicted at the beginning of the paragraph is a reasonable enterprise. How to find a finite Kripke structure and check properties in practice is discussed in the following sections.

3.1. Generalization of two logics for strategy-controlled systems

Now, we provide straightforward generalizations to systems controlled by strategies of the textbook semantics of two temporal logics, CTL* and μ-calculus. These definitions agree and confirm the soundness of Definition 4, since applying them to a system $(\mathcal{K}, E(\lambda))$ will be proven equivalent to applying the classical definitions to $\mathcal{U}(\mathcal{K}, \lambda)$.

*3.1.1. CTL**

CTL* [34] is a branching-time temporal logic that extends both LTL and CTL, written using the following grammar:

$$\Phi ::= \bot \mid \top \mid p \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \mathbf{A}\,\phi \mid \mathbf{E}\,\phi$$
$$\phi ::= \Phi \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \bigcirc\phi \mid \Diamond\phi \mid \Box\phi \mid \phi\,\mathbf{U}\,\phi$$

Terms built from ϕ, called *path formulae*, describe properties of fixed execution paths: $\bigcirc\phi$ tells that the property ϕ is satisfied in the next state of the path, $\Diamond\phi$ and $\Box\phi$ say that ϕ holds in some or all states of the path respectively, and $\phi_1\,\mathbf{U}\,\phi_2$ claims that ϕ_2 is satisfied in some state and ϕ_1 holds until then. Terms under the Φ symbol are called *state formulae* and refer to a state of the transition system, to the atomic properties $p \in AP$ it satisfies, and the paths leaving from it, quantified either universally $\mathbf{A}\,\phi$ or existentially $\mathbf{E}\,\phi$. LTL is the subset with formulae of the form $\mathbf{A}\,\phi$ where ϕ does not contain path quantifiers, and the initial $\mathbf{A}$ is left implicit. CTL is the subset in which every path operator is preceded by a quantifier. For example, the CTL formula $\mathbf{A}\,\Box\,(p \to \mathbf{E}\,\Diamond\,p)$ tells that it is possible to reach a state where q holds whenever p holds.

The semantics of CTL* is usually expressed by a satisfaction relation on states $\mathcal{K}, s \vDash \Phi$ and on paths $\mathcal{K}, \pi \vDash \phi$. However, when $\mathcal{K}$ is controlled by a strategy, a state formula like $\mathbf{E}\,\phi$ should not quantify over all paths, but only over those allowed by the strategy. Moreover, this subset of paths may depend not only on the last state but on the whole history of the execution. Consequently, the satisfaction relation for strategy-controlled systems replaces the state s by a (partially consumed) extensional strategy $\mathcal{K}, E \vDash \Phi$, and path formulae also carry a strategy in addition to the chosen path $\mathcal{K}, E, \pi \vDash \phi$ where $\pi \in E$.[4] To maintain this information in the following recursive definition, we introduce the operation $E \restriction ws \coloneqq \{s\pi : ws\pi \in E\}$ that gives the execution paths allowed by a strategy $E \subseteq S^\infty$ to continue from ws. Given $\pi = (\pi_k)_{k=0}^\infty$, we denote the suffix from k by $\pi^k = (\pi_{k+n})_{n=0}^\infty$, and the prefix of length $n+1$ by $\pi^{\leq n} = \pi_0 \cdots \pi_n$. For readability, the initial $\mathcal{K}$ is omitted.

1. $E \vDash p$ iff $\forall\,\pi \in E \quad p \in \ell(\pi_0)$
2. $E \vDash \neg\Phi$ iff $E \nvDash \Phi$
3. $E \vDash \Phi_1 \wedge \Phi_2$ iff $E \vDash \Phi_1$ and $E \vDash \Phi_2$
4. $E \vDash \mathbf{E}\,\phi$ iff $\exists\,\pi \in E \quad (E \restriction \pi_0), \pi \vDash \phi$
5. $E, \pi \vDash \Phi$ iff $E \vDash \Phi$
6. $E, \pi \vDash \neg\phi$ iff $E, \pi \nvDash \phi$
7. $E, \pi \vDash \phi_1 \wedge \phi_2$ iff $E, \pi \vDash \phi_1$ and $E, \pi \vDash \phi_2$
8. $E, \pi \vDash \bigcirc\varphi$ iff $(E \restriction \pi_0\pi_1), \pi^1 \vDash \varphi$
9. $E, \pi \vDash \phi_1\,\mathbf{U}\,\phi_2$ iff $\exists\,n \geq 0 \;\; E \restriction \pi^{\leq n}, \pi^n \vDash \phi_2 \wedge \forall\,0 \leq k < n \;\; E \restriction \pi^{\leq k}, \pi^k \vDash \phi_1$

By the usual equivalences, other operators are indirectly defined. This is a direct generalization of the classical semantic definition [34], and similar variations have appeared

[4] The definition of the satisfaction relation $\mathcal{K}, E, \pi \vDash \phi$ maintains the invariant that $\pi'_0 = \pi_0$ for all $\pi' \in E$. The fourth item in the definition includes $E \restriction \pi_0$ so that the invariant holds initially regardless of the input E.

in the literature when studying CTL* in the context of tree languages [79] and other extensions of this logic. The only substantial changes are in (4), where only executions in E are considered, and in (8) and (9), where the strategy argument is updated to the allowed executions from the time point where the recursive relation is evaluated. In fact, this definition coincides with the classical relation if we take $E = \Gamma^{\omega}_{\mathcal{K},s}$.

Proposition 1. *Given a CTL* formula Φ, $\mathcal{K}, s \vDash \Phi$ iff $\Gamma^{\omega}_{\mathcal{K},s} \vDash \Phi$.*

As promised in the first lines of this section, we finally claim that checking a CTL* property according to this generalized definition in $\mathcal{K}$ is the same as doing so with the standard definition on the unwinding or a bisimilar structure, because CTL* is invariant by bisimulation.

Proposition 2. *Given $(\mathcal{K}, E(\lambda))$ and a CTL* formula φ, $\mathcal{U}(\mathcal{K}, \lambda) \vDash \varphi$ iff $\mathcal{K}, E(\lambda) \vDash \varphi$.*

Proposition 3 ([8, Theorem 7.20])**.** *Two states, s_1 of $\mathcal{K}_1$ and s_2 of $\mathcal{K}_2$, are bisimilar iff $\mathcal{K}_1, s_1 \vDash \varphi \iff \mathcal{K}_2, s_2 \vDash \varphi$ for all CTL* (or for all CTL) formulae φ.*

3.1.2. -calculus

Modal μ-calculus [47] is an extension of the Hennessy-Milner logic [41] with least μ and great ν fixed-point operators. It can be used to express edge-aware properties on labeled transition systems using two modalities, $[a]\,\varphi$ that asserts that all states reachable by an a action satisfy φ, and $\langle a \rangle\,\varphi$ which claims the existence of a successor by a that satisfies φ. Formulae may contain variables Z bound by fixed-point operators.

$$\varphi ::= \bot \mid \top \mid p \mid Z \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid [a]\,\varphi \mid \langle a \rangle\,\varphi \mid \mu Z.\varphi \mid \nu Z.\varphi$$

The classical value of a μ-calculus formulae is the set of states in which it holds, written $[\![\phi]\!]_\eta$, where $\eta : \mathrm{Var} \to \mathcal{P}(S)$ is an assignment of values to the free variables that may appear in nested formulae. This logic is more expressive,[5] but less intuitive and popular than CTL* and its sublogics. However, model checkers for μ-calculus are available like muCRL2 [17] and LTSmin [45]. As well as the previous logics, μ-calculus is invariant by bisimulation.

Proposition 4 ([23, Theorem 6:10])**.** *If a state s_1 of $\mathcal{K}_1$ is bisimilar to a state s_2 of $\mathcal{K}_2$ then for every closed μ-calculus formula φ: $s_1 \in [\![\varphi]\!]_{\mathcal{K}_1,\eta_1}$ iff $s_2 \in [\![\varphi]\!]_{\mathcal{K}_2,\eta_2}$.*

The following generalization mimics the original definition [23, §6], but the denotation of a formula is a set of trees or strategies $\langle\!\langle\varphi\rangle\!\rangle_\xi \subseteq \mathcal{P}(\Gamma_\mathcal{K})$ instead of a set of states. The idea is that a system controlled by strategies $(\mathcal{K}, E)$ satisfies a μ-calculus formula φ iff $E \in \langle\!\langle\varphi\rangle\!\rangle_\xi$. A valuation is now $\xi : \mathrm{Var} \to \mathcal{P}(\mathcal{P}(\Gamma_\mathcal{K}))$ and $\xi[Z/U]$ is the function ξ with its value for the variable Z replaced by U.

1. $\langle\!\langle p\rangle\!\rangle_\xi = \{T \subseteq \Gamma_\mathcal{K} : \forall \pi \in T \quad p \in \ell(\pi_0)\}$
2. $\langle\!\langle \neg\varphi\rangle\!\rangle_\xi = \mathcal{P}(\Gamma_\mathcal{K}) \setminus \langle\!\langle\varphi\rangle\!\rangle_\xi$

[5] CTL* formulae can be translated into μ-calculus, but not all μ-calculus can be expressed in CTL*. For example, $\mathbf{A}\Diamond p$ is $\mu Z.(p \vee [\tau]\,Z)$ and $\mathbf{E}\Box p$ is $\nu Z.(p \wedge \langle\tau\rangle\,Z)$, being τ the only label. However, CTL* cannot express that p is satisfied at all even states $\mu Z.p \wedge [\tau][\tau]\,Z$.

3. $\langle\!\langle \varphi_1 \wedge \varphi_2 \rangle\!\rangle_\xi = \langle\!\langle \varphi_1 \rangle\!\rangle_\xi \cap \langle\!\langle \varphi_2 \rangle\!\rangle_\xi$
4. $\langle\!\langle Z \rangle\!\rangle_\xi = \xi(Z)$
5. $\langle\!\langle \langle a \rangle \varphi \rangle\!\rangle_\xi = \{T \subseteq \Gamma_{\mathcal{K}} : \exists\, sa\pi \in T \quad T \upharpoonright sa\pi_0 \in \langle\!\langle \varphi \rangle\!\rangle_\xi\}$
6. $\langle\!\langle [a] \varphi \rangle\!\rangle_\xi = \{T \subseteq \Gamma_{\mathcal{K}} : \forall\, sa\pi \in T \quad T \upharpoonright sa\pi_0 \in \langle\!\langle \varphi \rangle\!\rangle_\xi\}$
7. $\langle\!\langle \nu Z.\varphi \rangle\!\rangle_\xi = \bigcup \{F \subseteq \mathcal{P}(\Gamma_{\mathcal{K}}) : F \subseteq \langle\!\langle \varphi \rangle\!\rangle_{\xi[Z/F]}\}$
8. $\langle\!\langle \mu Z.\varphi \rangle\!\rangle_\xi = \bigcap \{F \subseteq \mathcal{P}(\Gamma_{\mathcal{K}}) : \langle\!\langle \varphi \rangle\!\rangle_{\xi[Z/F]} \subseteq F\}$

For instance, the denotation of an atomic proposition p takes all strategies whose paths satisfy p in their initial terms, instead of all states that satisfy p in the classical definition. Similarly, the modality $\langle a \rangle \varphi$ takes all strategies with a path that satisfy φ after an a transition. As usual, for the fixpoint in (6) to be well-defined, φ must be monotone, so every variable must be under an even number of negations. The semantic definition in (6) and (8) are a consequence of the usual equivalences, for example $[a]\varphi \equiv \neg\langle a \rangle \neg \varphi$.

The following two results are the counterparts of Propositions 1 and 2 for CTL*, and say that the definition is actually a generalization of the classical one, and that it is coherent with the procedure proposed for model checking strategy-controlled systems.

Proposition 5. *Given* $(\mathcal{K}, E)$ *and a closed* μ*-calculus formula* φ*,* $s \in [\![\varphi]\!]_{\mathcal{K},\eta}$ *iff* $\Gamma_{\mathcal{K},s} \in \langle\!\langle \varphi \rangle\!\rangle_{\mathcal{K},\xi}$ *for any* η *and* ξ*.*

Proposition 6. *Given* $(\mathcal{K}, E(\lambda))$ *and a closed* μ*-calculus formula* φ*,* $s \in [\![\varphi]\!]_{\mathcal{U}(\mathcal{K},\lambda),\eta}$ *for* $s\pi \in E$ *iff* $E \in \langle\!\langle \varphi \rangle\!\rangle_{\mathcal{K},\xi}$ *for any* η *and* ξ*.*

4. Specification and model checking in Maude by an example

In this section and through an example, we explain how strategy-controlled systems can be specified and model checked in Maude. The example is the simple and classical river-crossing puzzle, where a shepherd needs to cross a river carrying a wolf, a goat, and a cabbage. The only means is using a boat that only the shepherd can drive and with room for only one more passenger. Shipping the companions of the shepherd one by one would be a solution, but the wolf would eat the goat and the goat would eat the cabbage as soon as the shepherd leaves them alone. First of all, we should specify the signature of the problem as a functional module.

```
fmod RIVER-DATA is
  sorts Being Side Group River .
  subsorts Being Side < Group .

  ops shepherd wolf goat cabbage : -> Being [ctor] .
  ops left right : -> Side [ctor] .
  op __  : Group Group -> Group [ctor assoc comm] .
  op _|_ : Group Group -> River [ctor comm prec 50] .

  vars G1 G2 : Group .

  op initial : -> River .
  eq initial = left shepherd wolf goat cabbage | right .

  op risky : River -> Bool .
```

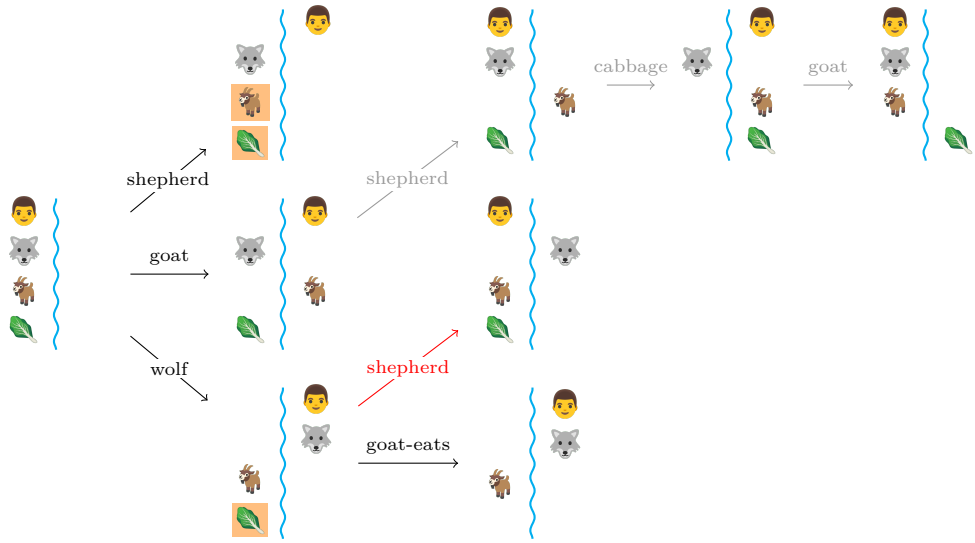

Figure 1: Partial rewriting tree for the river-crossing puzzle.

```
  eq risky(shepherd G1 | G2 wolf goat ) = true .
  eq risky(shepherd G1 | G2 goat cabbage) = true .
  eq risky(G1 | G2) = false [owise] .
endfm
```

The characters of the puzzle are declared as constants of sort `Being` with a multiple operator declaration (`ops`), and two other constants `left` and `right` of sort `Side` identify both sides of the river. A single being or side tag is a group, since their sorts are subsorts of `Group`, and more interesting groups can be built with the juxtaposition operator `__`. Two groups configure a river, one for each border, with the operator `_|_`. Associativity and commutativity are indicated by the `assoc` and `comm` attributes of their operator declarations. Groups are associative and commutative because they are sets, and the river is commutative because this will simplify the specification of rules. Finally, the `initial` position of the puzzle is defined by an equation to the term in which all characters are on the left border. The predicate `risky` identifies the states in which some being is at risk of being eaten, and it is defined with three equations.[6]

On top of this functional module, the possible moves of the game are specified using rules. The system module `RIVER` imports the functional module `RIVER-DATA` and defines a rule to cross the river with each character, and two more rules `wolf-eats` and `goat-eats` that make the mentioned animal eat its colleague one trophic level below.

```
mod RIVER is
  protecting RIVER-DATA .
```

[6] Equations annotated with the `otherwise` or `owise` attribute are executed only after all other equations have failed.

```
  vars L R : Group .

  rl [alone]   : shepherd L | R => L | R shepherd .
  rl [wolf]    : shepherd wolf L | R  => L | R shepherd wolf .
  rl [goat]    : shepherd goat L | R => L | R shepherd goat .
  rl [cabbage] : shepherd cabbage L | R
                              => L | R shepherd cabbage

  rl [wolf-eats] : wolf goat L | R shepherd
                     => wolf L | R shepherd .
  rl [goat-eats] : goat cabbage L | R shepherd
                       => goat L | R shepherd .
endm
```

The execution of these rules does not guarantee that the rules of the game are respected, since escaping from a risky state without applying `wolf-eats` or `goat-eats` is possible, as shown in Figure 1. This suggests that the eating rules must be applied eagerly before any movement rule is executed again, for which strategies will be helpful.[7]

This specification can already be executed within Maude. For instance, the `search` command finds terms matching a given pattern on the rewriting tree. We can use it to find out whether the goal position of the game can be reached.

```
Maude> search initial =>* left | right shepherd wolf goat cabbage  .

Solution 1 (state 32)
states: 33  rewrites: 64
empty substitution

No more solutions.
states: 36
```

The answer is affirmative, but we cannot be sure whether this state has been reached according to the rules of the game. In fact, the path that the search algorithm has followed to reach the goal position visits risky states, as can be seen using the `show path` command with the state number that appears next to the solution.

```
Maude> show path 32 .
state 0, River: right | left shepherd wolf goat cabbage
===[ rl ... [label wolf] . ]===>
state 2, River: left goat cabbage | right shepherd wolf
===[ rl ... [label alone] . ]===>
state 8, River: right wolf | left shepherd goat cabbage
...
state 32, River: left | right shepherd wolf goat cabbage
```

[7] In a previous Maude specification of the river-crossing puzzle [65], eating actions are written as equations so that they are applied eagerly by the Maude engine before the moving rules. However, this yields a rewrite theory where rules and equations are not coherent. A rewrite theory is *coherent* if for any term t rewritten by a rule to a term t', its canonical form u modulo equations and axioms can be rewritten to a term u' that is equationally equivalent to t', see [24, §5.3]. Coherence is assumed by Maude, which reduces terms to their canonical forms before applying a rule, not to miss any rewrite.

The second state in the path is a dangerous position where the `goat` can eat the `cabbage`, but this is not actually done in the third one. However, there may be other legitimate paths to the goal.

4.1. Controlling the system with strategies

In order to avoid that situation and enforce the game rules, various strategies will be defined in a strategy module `RIVER-STRAT` including `RIVER`.

```
smod RIVER-STRAT is
  protecting RIVER .

  var G : Group .

  strats oneCrossing eating cross&eat @ River .

  sd oneCrossing := alone | wolf | goat | cabbage .
  sd eating      := wolf-eats | goat-eats .
  sd cross&eat   := eating or-else oneCrossing .

  strats eagerEating safe @ River .

  sd eagerEating := (match left | right shepherd wolf
       cabbage goat) ? idle : (cross&eat ; eagerEating) .
  sd safe := (match left | G) ? idle
              : (oneCrossing ; not(eating) ; safe) .
endsm
```

The auxiliary strategies `oneCrossing` and `eating` apply any of the four movement rules and any of the two eating rules, respectively, and `cross&eat` applies either one according to the rules of the game, crossing only if eating is not possible. `eagerEating` is a recursive strategy that repeats this step forever or until the goal is found. `safe` is more restrictive and avoids visiting risky states by discarding all paths where eating is possible with $\mathbf{not}(\alpha) \equiv \alpha \,?\, \mathbf{fail} : \mathbf{idle}$. For example, the bottom branch of Figure 1 will be allowed by `eagerEating` but not by `safe`. Executing `safe` still requires visiting the risky state at the bottom to find out whether it is actually risky, but this execution path is discarded as if the state were never visited.

Now, we can ask whether the goal can be properly reached by evaluating `eagerEating` from the initial state with the `srewrite` command. The answer is positive.

```
Maude> srew initial using eagerEating .

Solution 1
rewrites: 74
result River: left | right shepherd wolf goat cabbage

No more solutions.
rewrites: 74
```

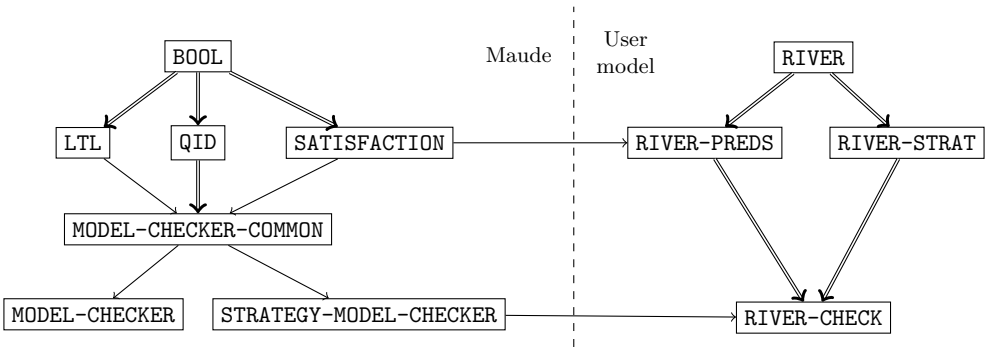

Figure 2: Structure of the strategy model-checker modules.

4.2. Preparing the specification for model checking

The last step for checking the model we have just specified, either with the previous model checkers or with those proposed in this paper, is the declaration of its atomic propositions on which temporal properties will be based. The Maude manual [24, §12] describes the required steps for strategy-free specifications, and the procedure does not hardly change for strategy-aware ones. It involves a few modules included in the `model-checker.maude` file shipped with the official and with our extended distribution of Maude, as shown in Figure 2.

Following with the example, the river-crossing puzzle is specified in the `RIVER` and `RIVER-STRAT` modules. First, we have to extend the system module `RIVER` by declaring some atomic propositions as Maude symbols and defining when they are satisfied.[8]

```
mod RIVER-PREDS is
  protecting RIVER .
  including SATISFACTION .

  subsort River < State .
  ops goal risky death : -> Prop [ctor] .

  var  R    : River .
  var  B    : Being .
  vars G G' : Group .

  eq left | right shepherd wolf goat cabbage |= goal = true .
  eq R |= goal = false [owise] .
  eq G cabbage | G' goat |= death = false .
  eq G cabbage goat | G' |= death = false .
  eq R |= death = true [owise] .
  eq R |= risky = risky(R) .
endm
```

[8]The declaration of the atomic propositions could have also been done in an extension of `RIVER-STRAT`. However, as a general principle, it is recommended not to include other content in strategy modules than strategy declarations and definitions, to emphasize their distinct concerns.

Atomic propositions must be declared within the sort `Prop` introduced by the `SATISFACTION` module of the model checkers' infrastructure. This module also declares a satisfaction symbol `_|=_ : State Prop -> Bool` that should be defined with equations for every state and atomic proposition. The states of the specified system must belong to the sort `State` appearing in the signature of `_|=_`, for what we have declared `River` as a subsort of `State`. Three propositions have been defined: `goal` that holds on the goal position, `death` that is only false when all eatable characters are in the scene, and `risky` that labels the risky states.

Finally, the `STRATEGY-MODEL-CHECKER` module, which gives access to the model checker, should be included in a new strategy module incorporating the property specification in `RIVER-PREDS` and the strategy specification in `RIVER-STRAT`.

```
smod RIVER-CHECK is
  protecting RIVER-STRAT .
  protecting RIVER-PREDS .
  including STRATEGY-MODEL-CHECKER .
  including MODEL-CHECKER .
endsm
```

At this point, given an initial term t and a strategy expression α, the Kripke structures that represent the strategy-free model and the strategy-aware model in `RIVER-CHECK` are completely specified. In the standard case, the model is the rewrite graph reachable from the initial term t where atomic propositions are evaluated using the `_|=_` symbol, i.e.

$$\mathcal{M} \coloneqq (T_{\Sigma/E}, A, (\rightarrow^1_{\mathcal{R}})^\bullet, \{t\}, AP_\Pi, L_\Pi)$$

where the actions A are the labels assigned to the rules, the atomic propositions AP_Π are the ground instances of `Prop` symbols, and L_Π maps a state to the set of those terms that are reduced to the term `true` by the equations. The relation $(\rightarrow^1_{\mathcal{R}})^\bullet$ is the one-step rule application $\rightarrow^1_{\mathcal{R}}$ where deadlock states are added a self loop to implement the stutter extension and work with infinite executions only. In the strategy-aware case, the model used by the LTL model checker guarantees that properties φ are only checked on the executions allowed by the strategy $E(\alpha, t)$, i.e. $\mathcal{M}, \pi \vDash \varphi$ for all $\pi \in E(\alpha, t)$. This Kripke structure is given by the graph of the small-step operational semantics described in Section 2.3,

$$\mathcal{M}_\alpha \coloneqq (\mathcal{XS}, A, \twoheadrightarrow, \{t \mathbin{@} \alpha\}, AP_\Pi, L_\Pi \circ \mathrm{cterm})$$

The relation $\twoheadrightarrow$ is also extended, but only on complete finite executions, i.e. only those leading to a solution by control transitions are added the self loop.[9]

These are the models internally used by the Maude LTL model checker and by our previous extension for strategy-controlled systems. For using the latter, the `STRATEGY-MODEL-CHECKER` module declares a symbol `modelCheck(`s`, `φ`, '`*name*`)` whose equational reduction invokes the verification of the LTL property φ from the initial state s controlled by the strategy whose name is *name*. The property φ is expressed as a Maude term whose syntax is specified in the `LTL` module in `model-checker.maude`. For instance, we can check the LTL properties $\Box(\mathit{risky} \rightarrow \bigcirc \mathit{death})$ with `eagerEating` and $\Diamond\, \mathit{goal}$ with `safe`.

[9] An execution state of the semantics may at the same time lead to a solution and to a new rewrite, so adding a loop is not always safe and these states must be duplicated.

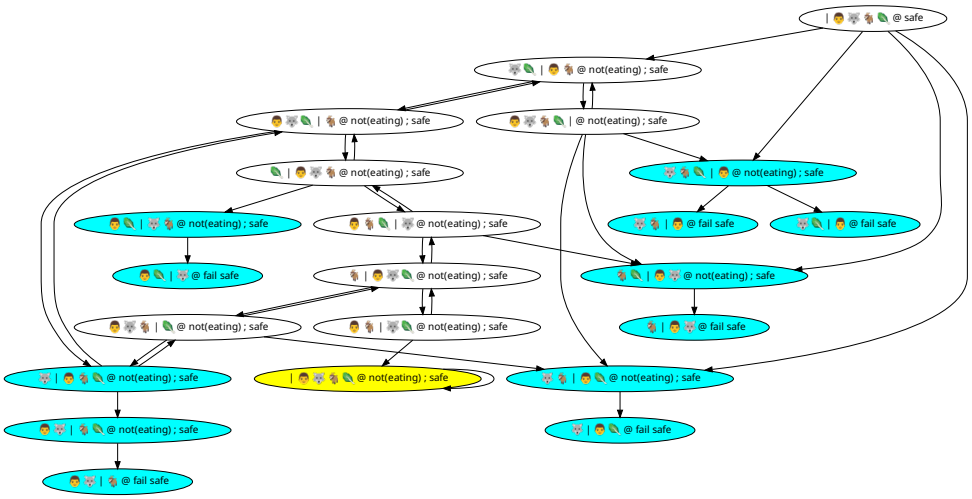

Figure 3: Model graph for the `safe` strategy.

The first one is satisfied, because eating rules are applied eagerly, but the second is not and a counterexample is shown.

```
Maude> red modelCheck(initial, [] (bad -> O death), 'eagerEating) .
rewrites: 123
result Bool: true
Maude> red modelCheck(initial, <> goal, 'safe) .
reduce in RIVER-CROSSING-SCHECK : modelCheck(initial, <> goal, 'safe) .
rewrites: 36
result ModelCheckResult: counterexample(
  {right | left shepherd wolf goat cabbage,'goat}
  {left wolf cabbage | right shepherd goat,'alone}
  {right goat | left shepherd wolf cabbage,'wolf}
  {left cabbage | right shepherd wolf goat,'goat}
  {right wolf | left shepherd goat cabbage,'cabbage},
  {left goat | right shepherd wolf cabbage,'alone}
  {left shepherd goat | right wolf cabbage,'alone})
```

This counterexample, not being as short as possible, shows that it is always possible to repeat movements in a loop. The complete graph for the `safe` strategy is shown in Figure 3, although the C++ implementation does not explicitly retain the strategy continuation of the semantics. The standard model checker can be used at the same time and it has a similar interface, where the strategy name is obviously omitted. For example, even in the uncontrolled system, the property $\Box(death \rightarrow \Box\neg\, goal)$ is satisfied.

```
Maude> red modelCheck(initial, [] (death -> [] ~ goal)) .
rewrites: 156
result Bool: true
```

In principle, using these Kripke structures, we will be able to check properties in no matter which logic. Without strategies, the Kripke structure directly represents the genuine rewrite graph, so there is no problem on applying other model-checking algorithms. However, our transformed strategy-aware structure only guarantees that its nonterminating executions coincide with the denotation of the strategy, but this is not enough. Looking at Figure 3, we see that some states marked in blue do not lead to any solution or infinite execution. They are the states where `not(eating)` has failed and where `eating` has been applied to figure it out on the fly. The depth-first search of the automata-theoretic approach used in the Maude LTL model checker ignores them automatically, since no cycle can be found through them, but algorithms for branching-time properties and tableau-based methods for LTL do not enjoy this property. These *failed states* can be safely removed when backtracking on the model generation or using an additional search, but there is another more serious problem that we explain in the following section.

5. Strategies and branching-time properties in Maude

The transition system yielded by the semantics is not ready for model checking branching-time properties, as seen in the previous section. However, the main reason is that states which are logically the same in the underlying system may be seen as distinct states due to the strategy continuation they hold, changing the tree structure of the model and making it depend on syntactical aspects of the strategies. We will illustrate this problem with an example of a simple vending machine:

```
mod VENDING-MACHINE is
  sorts Soup Thing Machine .
  subsort Thing < Soup .

  ops e a c : -> Thing [ctor] .
  op _[_] : Soup Soup -> Machine [ctor] .

  op empty : -> Soup [ctor] .
  op __ : Soup Soup -> Soup [ctor asoc comm id: empty] .

  vars O I : Soup .

  rl [put1]  : O e [I]     => O   [I e] .
  rl [apple] : O   [I e]   => O a [I] .
  rl [cake]  : O   [I e e] => O c [I] .
endm
```

The vending machine is a term O `[`I`]` where O represents the belongings of its user and I the content of its internal coin box. The machine can receive one euro coin `e` with the rule `put1`, and sells apples `a` and cakes `c` for one and two euros respectively. Let us consider $\alpha \equiv$ `put1 ; apple | put1 ; put1 ; cake` and $\beta \equiv$ `put1 ; (apple | put1 ; cake)`. These two different strategy expressions are essentially the same, because their abstract denotations coincide $E(\alpha) = E(\beta)$, so the vending machine must satisfy the same properties whether controlled by α or β according to Definition 4. Intuitively, these strategies can be identified with the plans of a person using the machine, where α has already decided which item to buy before inserting any coin, and β delays the choice until the

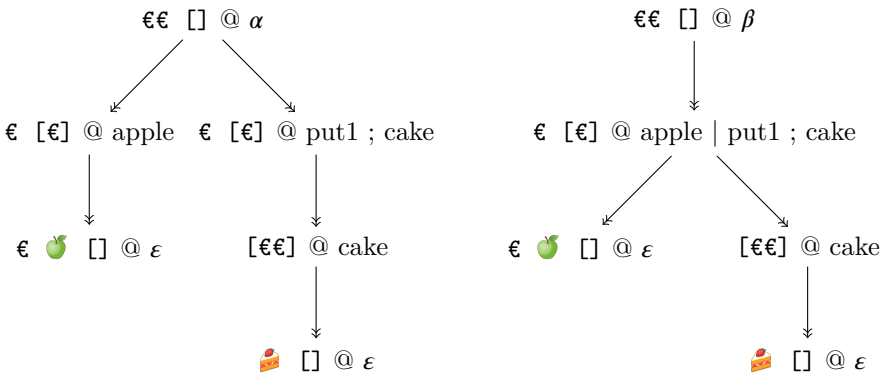

Figure 4: Strategy rewrite graph for α and β.

first coin is inserted. An external observer looking at the user interaction with the machine will not be able to distinguish when this choice has been made, it is not part of the *observable behavior*, and so it should be irrelevant for any property considered. This principle would not be obeyed if we applied standard algorithms on $\mathcal{M}_\alpha$ as Figure 4 shows. There, we can see the execution trees by the $\twoheadrightarrow$ relation from an initial configuration with two coins `e e [empty]` using both α (left) and β (right). Disregarding the strategy continuations after the `@` sign, i.e. projecting the nodes by the cterm function, we obtain rewriting trees where terms are connected by one-step rule rewrites. However, the tree for α cannot be considered a subtree of the execution tree of $(T_{\Sigma/E}, \rightarrow_R^1)$ because it contains repeated children. In any case, the branching structures of the execution trees for α and β and of their projections are manifestly different, and so they can be distinguished by branching-time temporal properties, as the CTL property $\mathbf{A} \bigcirc \mathbf{E} \Diamond \mathit{hasCake}$ attests.

```
mod VENDING-MACHINE-PREDS is
  protecting VENDING-MACHINE .
  including SATISFACTION .

  sort Machine < State .
  op hasCake : -> Prop [ctor] .

  vars I O : Soup .
  eq O c [I] |= hasCake = true .
  eq O   [I] |= hasCake = false [owise] .
endm
```

In effect, in the immediate successors of the root of the α tree the full path is already chosen, and in the left one no cake is ever bought. On the contrary, there is only one immediate successor of the initial state for β, where we can still choose the right branch to get the cake.

The ambiguity on the satisfaction of the atomic property by the strategy $E(\alpha) = E(\beta)$ should be avoided. In this example, the problem would be solved if the two successors of the root in the execution tree for α were combined into a single state, whose projection will be well-defined since they share the same term. Merging successors with a common

base term is a general solution to the problem that can be applied locally, solves the ambiguity, and produces a Kripke structure bisimilar to the unwinding of the strategy as desired. The following definition formalizes this construction and the removal of failed states discussed in the previous section. Remember that a state is valid

$$\text{valid}(q) \coloneqq \exists\, t \in T_\Sigma \quad q \to^*_{s,c} t\,@\,\varepsilon \quad \vee \quad \exists\, (q_n)_{n=1}^{\infty} \quad q \twoheadrightarrow q_1 \twoheadrightarrow q_2 \twoheadrightarrow \cdots$$

if a solution or a nonterminating execution can be followed from it.

Definition 5. *Given a strategy expression α and $t \in T_\Sigma$, we define the Kripke structure* $\mathcal{M}'_\alpha \coloneqq (\mathcal{XS}', A, [\twoheadrightarrow]', \{\{t\,@\,\alpha\}\}, \mathrm{AP}_\Pi, L_\Pi \circ \mathrm{cterm})$ *where*

$$\mathcal{XS}' = \{Q \subseteq \mathcal{P}(\mathcal{XS}) : \exists\, t \in T_\Sigma \quad \forall q \in Q \;\; \mathrm{cterm}(q) = t \;\wedge\; \exists\, q \in Q \;\; \mathrm{valid}(q)\},$$

and for any $Q, Q' \in \mathcal{XS}'$

$$Q\ [\twoheadrightarrow]'^{a}\ Q' \iff \exists\, t \in T_\Sigma \quad Q' = \{q' : q \twoheadrightarrow^a q',\, q \in Q, \mathrm{cterm}(q') = t\}$$

In summary, the states of $\mathcal{M}'_\alpha$ are sets of execution states with a common projection, and the successors of these sets are the union of the successors of their elements grouped by their subject terms and by the action.

Theorem 2. *$\mathcal{M}'_\alpha$ and $\mathcal{U}(\mathcal{M}, \lambda_{E(\alpha)})$ are bisimilar Kripke structures.*

Theorem 2 tells that $\mathcal{M}'_\alpha$ is an effective candidate to check branching-time properties on Maude specifications with strategies according to the ideas of Section 3. All these structures and propositions have been stated in terms of labeled transition systems, while state-based logics like LTL, CTL, and CTL* are defined on unlabeled transition systems. As we mentioned in Section 2.1, this is without loss of generality, because unlabeled transition systems can be viewed as labeled ones with a single arbitrary label. However, it is important that we forget about the labels of a labeled transition system before checking state-based properties, not only by efficiency reasons, but also by semantic ones. Otherwise, the labels will change the model semantics as the strategy continuations did in the previous section. For instance, suppose a strategy `r1 ; r2 | r3 ; r4` is applied to a term t_1 with the rules $t_1 \to^{\mathtt{r1}} t_2$, $t_1 \to^{\mathtt{r3}} t_2$, $t_2 \to^{\mathtt{r2}} t_3$, and $t_2 \to^{\mathtt{r3}} t_4$. The CTL property $\mathbf{A} \bigcirc \mathbf{E} \Diamond t_4$ will not be true if edge labels are considered, but it will if they are not, as it should be for state-based logics. On the contrary, for logics that operate on labeled transition systems like μ-calculus, the edge labels should be preserved and used to distinguish successor states when they are merged, because our notion of strategy $\lambda : (S \cup A)^+ \to \mathcal{P}(A \times S)$ conditions the next steps on the previous actions too. Using the same example, $\langle \mathtt{r1} \rangle \langle \mathtt{r4} \rangle \top$ should only be true if `r4` can be applied after `r1`. With these precautions, the following corollary claims that we can check CTL, CTL*, and μ-calculus properties, among others, using $\mathcal{M}'_\alpha$.

Corollary 1. $(\mathcal{M}, E(\alpha, t)) \vDash \varphi \iff \mathcal{M}'_\alpha \vDash \varphi$ *for any bisimilarity-invariant temporal property φ.*

The generated transition system $\mathcal{M}_\alpha$ is finite and its transition decidable if the reachable states from the initial one are finitely many [71]. Since merged states are the combinations of normal execution states, the number of states can grow exponentially at worst, although it would usually decrease, like in the vending machine example.

Corollary 2. *If the reachable states from $t\,@\,\alpha$ by $\to_{s,c}$ are finitely many, $(\mathcal{M}, E(\alpha, t)) \vDash \varphi$ is decidable for LTL, CTL*, and μ-calculus.*

6. Model checking using external model checkers

The extension of the Maude LTL model checker for strategy-controlled specifications [71] generates as part of its job a labeled transition system, the $\mathcal{M}_\alpha$ of Section 4. With the adaptations described in Section 5, this LTS can be transformed into $\mathcal{M}'_\alpha$, where branching-time properties can be properly checked. Thanks to the modular design of the original model checker, adopted by our extension, this model is exposed as an abstract Kripke structure where the successors and the atomic properties satisfied by a state can be queried using C++ functions. Hence, model checking properties in other logics only requires implementing their algorithms and the adaptations on top of this interface. However, instead of writing our own model-checking algorithms, we have found convenient to reuse already used and tested implementations for the target logics, since they are ultimately based on Kripke structures. A good candidate is the language-independent model checker LTSmin [45], which is able to efficiently interact with our Kripke-like representation of the model on the fly at the C++ level and supports all logics we have considered here, CTL* and μ-calculus. In addition, we have established connections with other model checkers like NuSMV [21], the `pyModelChecking` [18] library, and Spot [31], and we have also written our own implementation of a μ-calculus algorithm. More details about these connections are given at the end of this section. Additional logics and backends can be added without much effort using this approach.

Aiming at discharging users from learning the particular syntax and mode of operation of the different backends, a common and simplified interface is provided by the *unified Maude model-checking tool* `umaudemc` [70]. This program has a graphical and a command-line interface where the model-checking problem data is entered and the results are shown. The command for checking a property is the following:

`umaudemc check` ⟨ *file name* ⟩ ⟨ *initial term* ⟩ ⟨ *formula* ⟩ [⟨ *strategy* ⟩]

The formula can be expressed in a syntax that extends the predefined Maude `LTL` module with operators for CTL* and μ-calculus. In the first case, the only new constructors are the universal `A_` and existential `E_` path quantifiers. For the μ-calculus, the syntax is extended with the universal modalities `[_]_` and `[.]_`, the existential modalities `<_>_` and `<.>_`, the fixed-point operators `mu_._` and `nu_._`, and variables.

```
*** CTL and CTL*
op A_ : Formula -> Formula [ctor prec 53] .
op E_ : Formula -> Formula [ctor prec 53] .

*** mu-calculus
subsort @MCVariable@ < Formula .

op <.>_  : Formula -> Formula [ctor prec 53 format (c o d)] .
op [.]_  : Formula -> Formula [ctor prec 53 format (c d d os d)] .
op <_>_  : @ActionSpec@ Formula -> Formula [ctor prec 53 ...] .
op [_]_  : @ActionSpec@ Formula -> Formula [ctor prec 53 ...] .
op mu_._ : @MCVariable@ Formula -> Formula [ctor prec 64] .
op nu_._ : @MCVariable@ Formula -> Formula [ctor prec 64] .

*** Action lists
sorts @ActionSpec@ @ActionList@ .
```

```
subsort @ActionList@ < @ActionSpec@ .
op __ : @ActionList@ @ActionList@ -> @ActionList@ [ctor assoc] .
op ~_ : @ActionList@ -> @ActionSpec@ [ctor] .
```

Modalities are generalized so that they can take one or more rule labels of the module as actions, separated by space. This follows the widespread notation $[C]\,\varphi \coloneqq \bigwedge_{a \in C} [a]\,\varphi$ and $\langle C \rangle\,\varphi \coloneqq \bigvee_{a \in C} \langle a \rangle\,\varphi$. In case C is the complete set of actions, a dot can be written instead. The complement of the list of actions can be specified by preceding it with the negation symbol `~`. Variables for μ-calculus can be any token that does not conflict with the other elements in the formula. The sorts `@ActionList@` and `@MCVariable@` are populated at the metalevel before parsing, based on the rule labels of the target module and on a previous scan of the formula. The `umaudemc` tool parses the input formula within this Maude signature, deduces the least-general logic this formula belongs to, and then calls the appropriate backend with the appropriate configuration.

To illustrate its usage, we will check some branching-time properties of the river-crossing puzzle. The CTL formula $\mathbf{A}\,\square\,\mathbf{E}\,\diamond\,\mathit{goal}$ expresses that every state of the river-crossing puzzle can be continued to a solution. This formula is satisfied when the system is controlled by the `safe` strategy, but not when using the `eagerEating` strategy or when the system runs uncontrolled.

```
$ umaudemc check river.maude initial 'A [] E <> goal' safe
The property is satisfied in the initial state
(16 system states, 264 rewrites).

$ umaudemc check river.maude initial 'A [] E <> goal' eagerEating
The property is not satisfied in the initial state
(43 system states, 4012 rewrites).

$ umaudemc check river.maude initial 'A [] E <> goal'
The property is not satisfied in the initial state
(36 system states, 3058 rewrites).
```

The reason is that no solution can be reached once a character has been eaten, which may happen in the last two cases. Counterexamples are only shown if the selected backend supports them, and the `-c` flag can be used to prefer one of these. The following counterexample confirms our explanation for the refutation of the last property.[10]

```
$ umaudemc check river.maude initial 'A [] E <> goal' -c
The property is not satisfied in the initial state
(36 system states, 125 rewrites)
| right | left shepherd wolf goat cabbage
∨ rl G' | shepherd G => G | shepherd G' [label alone] .
| right shepherd | left wolf goat cabbage
∨ rl shepherd G' | wolf goat G => shepherd G' | wolf G [label wolf-eats] .
0 right shepherd | left wolf cabbage
```

However, the property $\mathbf{A}\,\square\,(\mathit{risky} \vee \mathit{death} \vee \mathbf{E}\,\diamond\,\mathit{goal})$ holds under the `eagerEating` strategy.

[10] For a branching-time logic, counterexamples can be provided for purely universal formulae and examples for purely existential formulae. In case both quantifications are mixed, a prefix of the path until the second quantifier applies can be given.

```
$ umaudemc check river.maude initial \
    'A [] (risky \/ death \/ E <> goal)' eagerEating
The property is satisfied in the initial state
(43 system states, 1088 rewrites).
```

We can also check μ-calculus properties, like the fact that the only initial movement not leading to a risky state is `goat`:

```
$ umaudemc check river.maude initial \
    '[ alone wolf cabbage ] risky /\ < goat > ~ risky'
The property is satisfied in the initial state
(5 system states, 18 rewrites, 15 game states).
```

Then, we wonder if the goal can be reached without moving the goat again: this is the property $[\texttt{goat}](\mu Z.\, goal \vee \langle \texttt{alone wolf cabbage} \rangle Z)$ where the fixed-point subformula describes the states where the goal can be reached using any sequence of moves other that `goat`. The answer is no if the rules of the game are respected as in the `eagerEating` strategy:

```
$ umaudemc check river.maude initial \
    '[ goat ] (mu Z . goal \/ < ~ goat > Z)' eagerEating
The property is not satisfied in the initial state
(43 system states, 192 rewrites, 364 game states).
```

Notice that we have replaced the list of labels `alone wolf cabbage` by `~ goat` to illustrate the complement notation for actions. These are not exactly the same, because the complement of `goat` also includes the rules `wolf-eats` and `goat-eats`, but they do not change the satisfaction of the property. On the contrary, the uncontrolled system satisfies the formula, since it can pass by forbidden states:

```
$ umaudemc check river.maude initial \
    '[ goat ] (mu Z . goal \/ < ~ goat > Z)'
The property is satisfied in the initial state
(33 system states, 168 rewrites, 362 game states).
```

While the `umaudemc` tool automatically enables the branching-time adaptations of the model according to the input formula, these defaults can be overwritten with the `--purge-fails` and `--merge-states` options. Coming back to the vending machine example of Section 5, with the `merge-states` adaptation disabled, we can see that the CTL property $\mathbf{A} \bigcirc \mathbf{E} \diamondsuit \mathit{hasCake}$ is not satisfied when the system is controlled by the strategy α, but it is when controlled by the equivalent strategy β:

```
$ umaudemc check vending.maude initial 'A O E <> hasCake' \
  'put1 ; apple | put1 ; put1 ; cake' --merge-states=no
The property is not satisfied in the initial state
(6 system states, 72 rewrites)

$ umaudemc check vending.maude initial 'A O E <> hasCake' \
  'put1 ; (apple | put1 ; cake)' --merge-states=no
The property is satisfied in the initial state
(5 system states, 60 rewrites).
```

However, when states are properly merged, the property is satisfied for both strategy expressions as follows from Corollary 1:

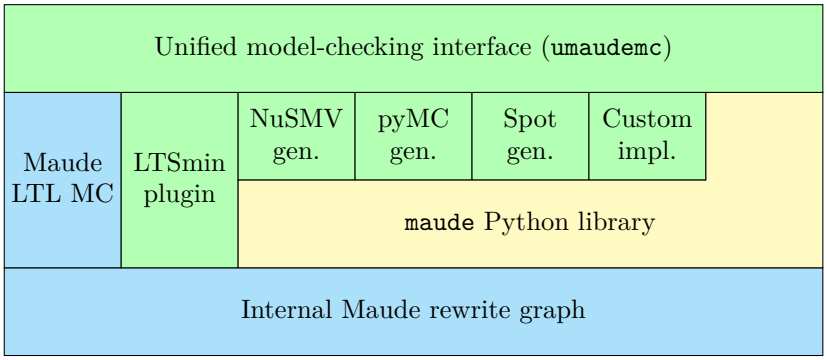

Figure 5: Architecture of the `umaudemc` model-checking tool.

```
$ umaudemc check vending.maude initial 'A O E <> hasCake' \
  'put1 ; apple | put1 ; put1 ; cake'
The property is satisfied in the initial state
(6 system states, 52 rewrites).
```

The transition systems generated for each model with the different adaptations can be observed with the `umaudemc graph` command. For instance, `umaudemc graph river.maude initial safe` would generate something similar to Figure 3, and its states in blue can be removed with the `--purge-fails=yes` option.

6.1. The architecture of `umaudemc`

As we have seen with the examples of the previous section, the `umaudemc` tool allows checking temporal properties on both standard and strategy-controlled Maude specifications regardless of which model-checking backend is doing the job behind the scenes. All of them rely on the internal Maude rewrite graph used by the Maude LTL model checker [33] and by our extension for strategy controlled systems [71], which correspond to the C++ classes `StateTransitionGraph` and `StrategyTransitionGraph` in their implementations. As illustrated in Figure 5, some backends access these graphs directly while others use a Python library called `maude` [69] that we have developed for this and other projects. This library exposes all relevant Maude entities and operations as objects and methods in Python by directly interacting with the Maude implementation at the binary level,[11] including the strategy-controlled and the standard rewrite graphs. Exploring these graphs and evaluating atomic propositions on them, models are generated for the various supported backends.

The `maude` library is also used directly by the `umaudemc` tool to process the problem data, the verification results and the counterexamples, and to produce printable graphs of the models. The extended language of temporal properties admitted by the tool is specified in a Maude module, parsed using the library, and translated to the syntax of the temporal properties supported by the selected backend. Whether the adaptations

[11] The `maude` library is a language binding implemented using the SWIG interface generator. More details are available in its repository [69].

	LTL	CTL	CTL*	μ-calculus	Lines
Extended Maude	on-the-fly				1200
LTSmin	on-the-fly	X	X	X	1140
pyModelChecking	tableau	X	X		147
NuSMV	tableau	X			199
Spot	automata				203
Builtin		X		X	400

Table 1: Logics supported by the backends in `umaudemc`.

of Section 5 are applied or not is also decided depending on the problem data, and they are implemented in C++ inside the LTSmin plugin or in Python for the backends based on the `maude` library. The `umaudemc` tool will detect which backends are installed and call the most convenient for each supported logic, although the search order can be changed with the `--backend` option. In addition to the Maude LTL model checker and LTSmin, which is described in Section 6.2, the available backends are:

- NuSMV [21], which supports LTL and CTL. The model is communicated by writing a low-level specification file in the NuSMV format. It calculates counterexamples for CTL properties too.
- `pyModelChecking` [18] is a Python library that targets LTL (by the tableau method), CTL, and CTL* model checking. The Kripke structure is constructed as a Python object from the Maude model.
- Spot [31] is a C++ framework for LTL and ω-automata manipulation with a Python library. Models are built as Kripke structures using this library, but it admits more complex ω-automata. It also admits on-the-fly model checking, but not through the Python interface.
- Our own implementation in Python of the μ-calculus model-checking algorithm in [14], using the Zielonka algorithm [82] for parity game solving.

Table 1 summarizes which logics can be checked with each backend. Although LTSmin supports all logics we have considered, other model checkers are easier to install, provide more informative output, or exhibit better performance in some cases despite their less efficient connection, as discussed in Section 7. Adding connections to other model checkers and logics is relatively simple, as suggested by the number of code lines written for each backend in Table 1, since the models described in Sections 4 and 5 are easily accessible and compatible in principle with any logic.

6.2. The LTSmin language plugin

LTSmin [45] is a collection of generic model-checking programs that can operate on models expressed in different specification languages. These models are exposed as Kripke structures by some builtin or pluggable language modules using its *Partitioned Next State Interface* (PINS). In order to check properties with this toolset, we have implemented a language module for Maude. The module `libmaudemc` is a shared C library linked

with the implementation of Maude[12] that exports the functions required by the PINS interface. Model checking a temporal property using LTSmin and the Maude plugin consists of the following steps:

1. The `pins2lts-*` model-checking tools of LTSmin are called with the problem data and with a `--loader` argument indicating the path of the Maude plugin. The Maude language module is loaded in memory using the POSIX's `dlopen` API, so that its exported functions and global variables required by the PINS interface can be accessed. One of these functions is called to pass the Maude-specific problem data to the plugin (the initial term, the strategy, and some other parameters) and prepare the Kripke structure that will be made available to the model-checking algorithms.
2. When the model-checking algorithm for the given logic wants to know if an atomic property p is satisfied in a state, it calls the `state_label` function of the plugin that evaluates the term cterm(Q) `|=` p and returns its Boolean result. When the model checker requires the successors of a state, it calls the `next_state` function that enumerates them with their corresponding edge labels.
3. The verification result is printed to the terminal. In the μ-calculus case, a parity game is generated instead, which has to be solved by an external tool from the mCRL2 project [17].

The Kripke structures presented to LTSmin are $\mathcal{M}$, $\mathcal{M}_\alpha$ or $\mathcal{M}'_\alpha$ depending on whether the model is controlled by a strategy or not, and on the arguments `--purge-fails` and `--merge-states` passed to the language plugin. In fact, the `next_state` function called in the second step is chosen at the beginning from a small set of alternative functions that implement the adaptations described in Section 5 for state-based or edge-based branching-time logics. This choice could be inferred from the temporal formulae to be checked, but this information is never passed to the language plugin. The same happens with the atomic propositions, which must be supplied directly with the `--aprops` argument, since the set of potential atomic proposition can be infinite as they are regular Maude operators with parameters. These disadvantages and the choice of an appropriate LTSmin command for a given property are avoided when using the `umaudemc` utility. Among the different programs included in LTSmin, `umaudemc` invokes its sequential explicit-state model checker `pins2lts-seq` for LTL and μ-calculus properties with action specifications,[13] and the symbolic model checker `pins2lts-sym` for CTL, CTL*, and μ-calculus properties without action specifications (i.e. using only the `<.>` and `[.]` modalities).

The Maude language module does not take full advantage of LTSmin. Its PINS interface allows representing states as vectors of integer indices and declaring dependencies between their entries, so that the model-checking algorithms can use them for better efficiency and parallelization. However, our plugin's states are single indices to the internal Maude rewrite graph. Automatically partitioning an arbitrary Maude specification and inferring dependencies between the resulting parts seems to be a very complex task.

[12] Maude is usually distributed as a single binary, but we have built it as a shared library `libmaude` to distribute the interpreter and this plugin together without including twice the same executable code.

[13] Notice that μ-calculus properties that refer to both edge and state labels cannot be verified with the last LTSmin version at the moment of writing (3.0.2). We have proposed a change that makes it possible and the modified version is available for download in [32] in the meantime.

7. Evaluation

We have tested and compared the performance of the model-checking backends described in this article using the collection of strategy-controlled Maude specifications and temporal properties available at the Maude strategy language website [32]. Most of these examples are relatively small (classical concurrency problems, games, models translated from other model-checking tools, etc) and they have been specifically written to test our model checker, but some others have a greater size and interest on their own. The results cannot be interpreted as a comparison of the model-checking tools themselves, since the figures also reflect the efficiency of the connections to our Maude models. The reader should also keep in mind from Section 6 that they all operate on the same Kripke structure produced by Maude from a strategy-controlled specification, with some common adjustements in the case of branching-time properties. In the same website, the complete listing of the test cases and their results are available for reproducibility, and they can be executed with the `test` subcommand of the `umaudemc` tool.

The plots in Figure 6 compare the time spent by the different backends to execute the same model-checking problems ordered by their number of states. These problems are given by a Maude module, an initial term, a strategy expression, and a temporal formula. For every test case c and backend b, the plot shows a specific marker determined by b in the coordinates

$$(n_c^{\text{LTSmin}}, (t_c^b - t_e^b) \,/\, (t_c^{\text{LTSmin}} - t_e^{\text{LTSmin}})),$$

where n_c^b and t_c^b are respectively the number of states and the execution time of the test case c in the backend b, and e is an empty test case with a single state and a trivial property. In other words, looking at a fixed vertical rule we can compare the performance of the different backends for a test case, since the height of the marks indicates the proportion of time a backend has taken to complete its task respect to LTSmin, so that higher means worse. LTSmin has been chosen as a common reference since it supports all considered logics, and so it can run all test cases. Even though the number of states are referred to a fixed backend, this figure is essentially a property of the test case and it is usually the same for all tools.[14] Moreover, we have subtracted the initialization time t_e^b before calculating the coefficients, because small examples are highly influenced by the quite different initialization times of the backends, with LTSmin being a thousand times slower on the empty example e than the Maude LTL model checker. In order to compare the new supported model checkers with the builtin Maude one, the left plot includes tests against linear-time properties. The results on our smaller collection of branching-time properties are shown in the right plot.

For LTL properties, the Maude model checker is usually and expectedly the fastest, since it is directly connected to the rewrite graphs. However, Spot is often and LTSmin sometimes very close. The peaks in the Maude curve above 10^3 states are caused by the different order in which states are explored by the backends. Although all these cases evaluate properties that are satisfied, in which the whole state space has to be expanded, some exploration orders may detect the equivalence of two states earlier in some corner

[14] The number of states may differ in test cases where the temporal property does not hold, since counterexamples can be found sooner or later by the different on-the-fly implementations, and in some corner cases explained in the following.

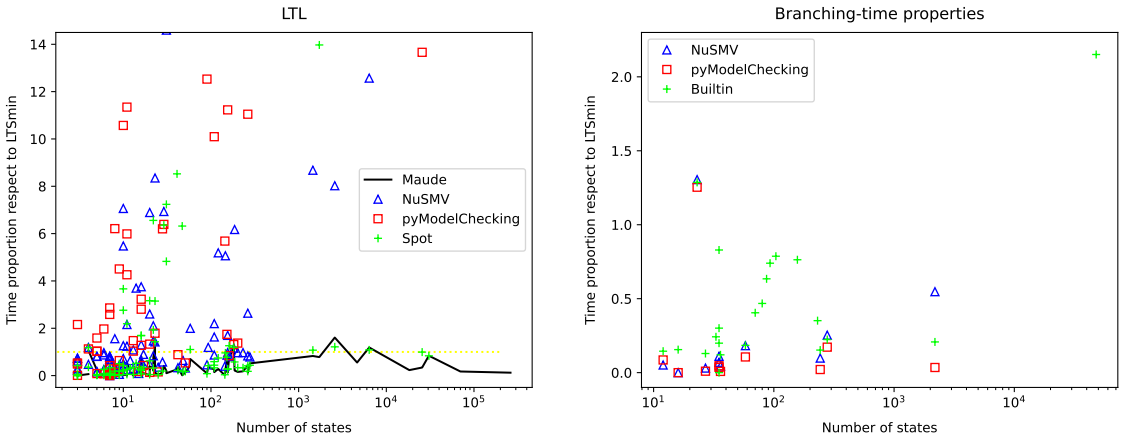

Figure 6: Compared performance of the model-checking backends.

cases. Surprinsingly, the Python-based algorithms are more efficient than LTSmin, but only with small examples. NuSMV and `pyModelChecking` do not behave bad for lower sizes, even though their algorithms do not operate on the fly, but they do not terminate in reasonable time and memory limits when the problems are big enough.

Regarding branching-time properties, although the size of the examples is small, we observe that LTSmin exhibits the worst performance and all other backends check the same test cases in half of the time. However, for the biggest μ-calculus problem checked, its performance is better than that of our builtin backend.

8. Related work

Three independent but related topics are addressed in this work, model checking, strategies and rewriting, which have had fruitful interactions. In this section, we review on related work the combinations of these topics towards approaches that are close to what has been presented here.

Strategies and rewriting. Strategies are inherent to rewriting and reduction, and so their study dates back to the origins of λ-calculus. When specifying the behavior of algorithms and other systems, the Kowalski's motto *Algorithm = Logic + Control* [46] is translated in this context to the *Rule + Strategies* approach [67, 51], where strategies express an additional level of specification that controls the rule rewriting system compositionally and without mixing their concerns. In addition to Maude's, other strategy languages have appeared like ELAN [11], Stratego [15], TOM [9], and ρLog [54] for term rewriting, and Porgy [35] for graph rewriting. Strategy-controlled specifications have been used to describe many examples of systems from different fields [42, 76, 80, 72, 2, 36]. However, the verification techniques used for these tools do not include model checking as understood here.

Model checking and rewriting. Various model checkers have been proposed for rewriting systems in the Maude context. The main one is the Maude LTL model

checker [33] integrated in the Maude interpreter and applied on many real models, among others [52, 63, 62]. In addition, μ-calculus model checkers have been once implemented in Maude itself [81, 50], the Real Time Maude [64] framework includes a Timed CTL model checker for real-time systems, the Maude LTL logical model checker [6] symbolically verifies infinite-state systems using narrowing, and some fragments of the Temporal Logic of Rewriting (TLR*) [59], whose relation with strategies is discussed below, can be checked with different implementations [7, 56]. Model checking has also been used in the CafeOBJ [30, 40] language, despite not including a dedicated model checker.

Model checking and strategies. The relation between model checking and strategies is a wide and active research topic in the context of games, multiagent and open systems, where strategies are usually followed by the players or agents to achieve some defined goal and where many properties can be expressed in terms of strategies. Representative logics are the *Alternating-Time Logics* ATL and ATL* [1] that respectively include the usual CTL and CTL* operators, but whose path quantifiers **E** and **A** are replaced by the strategic modalities $\langle\!\langle A \rangle\!\rangle$ and $[\![A]\!]$. The meaning of $\langle\!\langle A \rangle\!\rangle \varphi$ is that a strategy can be chosen for each of the agents in the coalition A to make φ hold, regardless of what the agents not in A do. The more expressive *Strategy Logic* (SL) [61] of Mogavero, Murano, Perelli, and Vardi is extended with strategy variables that can be existentially $\langle\!\langle x \rangle\!\rangle \varphi$ and universally $[\![x]\!]\, \varphi$ quantified and then assigned $(a, x)\, \varphi$ to one or more agents a. The satisfaction of their formulae is defined recursively, and so they have a concept of strategy-controlled checking to evaluate the φ and $(a, x)\, \varphi$ subformulae that coincides with our Definition 4, or more precisely, to the generalized CTL* semantics of Section 3.1. However, the semantics of these logics is crucially influenced by the type of strategy considered [44, 16], which in most cases is deterministic and sometimes memoryless, although the literature on variations of strategy logics is extensive, and the case of general intensional strategies are covered in logics like USL [20] and $\mathsf{SL}^{\prec}$ [38]. Our model-checking problem cannot be directly seen as a particular case of the problem for these logics, because strategies do not appear explicitly and they are always bound by quantifiers. On the contrary, the associated satisfaction problem for CTL* properties, whether there is a strategy such that the formula is satisfied under its control, can be expressed as a very particular case of the last two mentioned logics. In their full generality, model checkers have not been implemented for these logics as far as we know since their problems are very hard, but some subsets can be effectively model checked. MCMAS [53] is an extensive open-source model checker for multi-agent systems that supports ATL as verification logic, and restrictions of Strategy Logic through extensions [19]. In addition to the verification result, they can also synthesize strategies for the agents that make the formula hold. Strategy synthesis is related to the so-called *controller synthesis* [3] with important industrial applications. In a similar but different context lies Uppaal Stratego [28] from the Uppaal [49] modeling environment for real-time systems. This tool allows synthesizing strategies to make a property hold as in the other mentioned tools, but these strategies can be later used to execute the constrained system and model check it against other properties, considering only this restricted *strategy-space*. Strategies in this case are memoryless and deterministic, but they follow the same idea of this work.

Strategies have also been applied to reduce the search space for the sole purpose of model checking, by guiding its search to a counterexample or witness of the desired property, like search heuristics [66]. For example, this has been done [78] with a language

of reachability expressions including union, concatenation and iteration operators that resemble those of Maude and similar strategy languages.

Temporal Logic of Rewriting and other logics. Meseguer's Temporal Logic of Rewriting (TLR*) [59] is also connected with strategies and the Maude strategy language. This logic extends CTL* with *spatial action patterns* that symbolically designate a collection of rule applications. They can be used in path formulae to indicate how the next transition to be executed should be, and so TLR* is at the same time a state-based and edge-based temporal logic. For example, the property $\Box(\neg\text{goat} \rightarrow \bigcirc \text{risky})$ says that any action other than `goat` would lead to a risky state. Spatial action patterns can be more complex and include restrictions on the variables and the context where rules are applied. In fact, they are very similar to a combination of rule applications, the `top` modifier, and the **matchrew** of the Maude strategy language, to which they can be translated. The relation with strategies comes from the possibility of checking certain properties on infinite-state systems by a strategy-controlled exploration. Guarantee formulae (those only containing the temporal operators $\bigcirc$, $\Box$, and $\mathbf{U}$ without negations) can be translated to strategy expressions whose evaluation is a semidecision procedure for the original formulae. In Sections 6 and 7 of [58], a strategy language similar to that of Maude is introduced for this particular purpose.

$$b ::= \top \mid \bot \mid p \mid \neg b \mid b \wedge b \mid b \vee b$$
$$e ::= \mathit{idle} \mid \delta \mid \mathit{any} \mid e \wedge e \mid (e|e) \mid e\,;e \mid e+ \mid e\,\mathbf{U}\,e \mid e.b$$

However, some combinators of this language are neither available nor expressible in the current Maude strategy language, and so it cannot be used to implement these procedures. Notice that the strategy-aware model checker is not needed for that, but only the execution engine of strategies. On the other hand, writing a TLR* model checker for finite-state systems would be reasonably simple using the tools and connections developed in this work.

Finally, *propositional dynamic logic* (PDL) [37], *linear dynamic logic* (LDL) [39] and their variations are partially related to the model-checking problem of strategy-controlled systems. Their formulae $\langle r \rangle \varphi$ include complex actions r built using edge labels, regular expression combinators, and tests, which can be seen as a subset of the strategy language too. Atomic propositions and more complex temporal formulae can be checked at the end of those sequences of actions or at arbitrary points during them using tests. Unlike in our approach, properties are not checked in the system restricted by the action patterns but at some execution points indicated by them.

9. Conclusions and future work

Strategies are a useful resource for elaborating modular rewriting-based specifications, where simpler rules represent the local transformations of the model, and strategies describe at a higher level restrictions that capture its global behavior, guide them towards a goal, apply them more efficiently, etc. The current version of the Maude specification language [24] includes an LTL model checker for rewriting-based specifications and a strategy language to control rewriting, but these are independent and properties cannot be checked on strategy-controlled models. Therefore, in a previous work, we extended the

Maude LTL model checker to handle strategy-controlled systems [71]. The fundamental idea is that properties should only be checked on the executions allowed by the strategy, and a small-step operational semantics was defined to determine which are those for an expression in the Maude strategy language. Using this semantics, the original Kripke structure can be transformed to another one whose executions are exactly those allowed by the strategy, in which properties can be checked using standard algorithms. In this paper, we extend the discussion to branching-time properties, realizing that strategies can also be seen as subtrees of the execution trees of the original model where properties can be checked. However, the model transformation proposed for linear-time properties must be adapted to maintain the bisimilarity with the unwinding of the original model, so that properties are soundly checked.

In order to effectively verify branching-time properties, both on strategy-controlled and standard Maude specifications, we have implemented connections with external model checkers, with support for CTL* and μ-calculus. The infrastructure used to connect these model checkers is valuable by itself, and uses a library that allows manipulating and accessing the Maude entities and models from other programming languages [69]. It can be easily used to connect other visualization or verification tools, and to write programs that use Maude as a formal engine. All model checkers can be transparently accessed through a unified `umaudemc` tool [70] that provides extended information and graphical representations of the models and counterexamples. The performance of the connections to the external model checkers is comparable to the builtin Maude model checker.

This work can be extended with more logics and model-checking backends. Adding some would be specially affordable with the new tools, like the Property Specification Logic [43], already supported by some of the current backends, Meseguer's Temporal Logic of Rewriting and μ-calculus of rewriting. We can also relax the restriction to intensional strategies and try an alternative interpretation of the Maude strategy language, making the iteration behave as a Kleene star, so that fairness constraints can be expressed in the strategy itself. Exploring the satisfaction and strategy synthesis problem mentioned when discussing strategic logics could be another direction of future work.

Declaration of competing interest. The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements. Research partially supported by MCI Spanish projects *TRACES* (TIN2015-67522-C3-3-R) and ProCode-UCM (PID2019-108528RB-C22). Rubén Rubio is partially supported by MU grant FPU17/02319.

Appendix A. Proofs

We do not include all the details in the proofs, which are usually tedious inductive checks, but only sketch the main ideas and the most problematic steps.

Lemma 1. *Every closed ω-language is recognized by a deterministic Büchi automaton with trivial acceptance conditions.*

Proof. Since the language is ω-regular, there must be a Büchi automaton that recognizes it. Since the language is closed, the limits of all executions are allowed, so the acceptance conditions if any are superfluous and can be removed (we have proved this in [71]). The automaton with trivial acceptance conditions can then be determinized by the powerset construction used for finite automata, since the obstacle that impedes determinizing arbitrary Büchi automata are the Büchi conditions. □

Lemma 2. *If $\mathcal{K}$ and $\mathcal{K}'$ are bisimilar, $\bigcup_{s\in I} \ell(\Gamma_{\mathcal{K},s}) = \bigcup_{s\in I'} \ell'(\Gamma_{\mathcal{K}',s})$.*

Proof. $\mathcal{K}$ and $\mathcal{K}'$ are interchangeable in the lemma, so it is enough to prove $G \subseteq G'$, if G and G' are the unions in the statement, by induction with the property $p(ws) = \exists i' \in I', w's' \in \Gamma_{\mathcal{K}',i'} \quad \ell(ws) = \ell'(w's') \wedge (s,s') \in B$ for all $i \in I$ and $ws \in \Gamma_{\mathcal{K},i}$, where B is a bisimulation. The infinite executions are the limits of the finite ones, so they coincide too. □

Theorem 1. *Given an intensional strategy λ, there is a finite Kripke structure $\mathcal{K}'$ bisimilar to $\mathcal{U}(\mathcal{K},\lambda)$ if $E(\lambda)$ is ω-regular. The converse is not true, but in that case $\ell(E(\lambda))$ is ω-regular.*

Proof. In summary, the Büchi automaton for $E(\lambda)$ gives the finite Kripke structure $\mathcal{K}'$, with transition labels somehow clouding the proof. If $E(\lambda)$ is ω-regular and closed, there is a deterministic automaton $M = (Q, S \cup A, \delta, \iota, Q)$ for it with trivial Büchi conditions by Lemma 1. Moreover, all words accepted by this language alternate states in S with actions in A. Let $\mathcal{K}'$ be $(Q{\times}S, R', I', AP, \ell{\circ}\pi_2)$ where $I' = \{(q,s) : q \in \delta(\iota,s), s \in I\}$, π_2 is the second projection of the pair, and $((q,s),a,(q',s')) \in R' \iff \exists\, q_m \in Q \quad q_m \in \delta(q,a) \wedge q' \in \delta(q_m,s')$. $\mathcal{U}(\mathcal{K},\lambda)$ is bisimilar to $\mathcal{K}'$ by the following relation $B = \{(ws,(q,s)) : q \in \hat{\delta}(\iota,ws)\}$ where $\hat{\delta}(q,\varepsilon) = \{q\}$ and $\hat{\delta}(q,wx) = \bigcup_{q'\in\hat{\delta}(q,w)} \hat{\delta}(q',x)$. First, it is clear that states related by B have the same label. Then, if $ws \to wsas'$ and $(ws,(q,s)) \in B$, the state (q',s') where $q' \in \hat{\delta}(\iota,wsas')$ satisfies $((q,s),a,(q',s')) \in R'$ and $(wsas',(q',s')) \in B$ (the determinism is M is used here). $\hat{\delta}(\iota,wsas')$ is not empty because $wsas'$ is a prefix of an execution allowed by the strategy, and so recognized by M. The other simulation is proven similarly.

The converse is not true. For an unlabeled counterexample, take $\mathcal{K} = (\{a,b\}, \{a,b\}^2, \{a\}, \varnothing, \varnothing)$, $\lambda(w) = \{a,b\}$ if $w = a^n$ for n prime and $\{a\}$ otherwise, and $\mathcal{K}' = (\{c\}, \{(c,c)\}, \{c\}, \varnothing, \varnothing)$. $E(\lambda)$ is not ω-regular and $\mathcal{K}'$ is bisimilar to $\mathcal{U}(\mathcal{K},\lambda)$ by the only possible total relation. Hence, we only prove that $\ell(E(\lambda))$ is ω-regular. Given $\mathcal{K}' = (S', R', I', AP, \ell')$, we define the ω-automaton $M = (Q, \mathcal{P}(AP) \cup A, \delta, \iota, Q)$ where $Q = S' \times \{0,1\} \cup \{\iota\}$ and $\delta(\iota,P) = \{(s,1) : \ell(s) = P, s \in I'\}$, $\delta((s,1),a) = \{(s',0) : (s,a,s') \in R'\}$ and $\delta((s,0),P) = \{(s,1)\}$ if $P = \ell(s)$. It is clear that the runs of M are of the form $\iota(s_0,1)(s_1,0)(s_1,1)(s_2,0)(s_2,1)\cdots$ and they accept words $s_0a_1s_1a_2s_2\cdots$ that coincide with the executions of $\mathcal{K}'$. By Lemma 2, the projected executions of $\mathcal{K}'$ coincide with those of $\mathcal{U}(\mathcal{K},\lambda)$ and these are exactly $\ell(E(\lambda))$. Hence, M is a Büchi automaton for the ω-regular language $\ell(E(\lambda))$. Finally, $\ell(E(\lambda))$ being ω-regular is not enough for the existence of a bisimilar finite $\mathcal{K}'$. The previous counterexample can be refined to show this.

□

Proposition 1. *Given a CTL* formula Φ, $\mathcal{K}, s \vDash \Phi$ iff $\Gamma^{\omega}_{\mathcal{K},s} \vDash \Phi$.*

Proof. The key fact is that the possible continuations of any finite execution ws for $\Gamma^\omega_{\mathcal{K}}$ only depend on its final state s, since the executions are unrestricted. Hence, $\Gamma^\omega_{\mathcal{K},s} \restriction (ws') = \Gamma^\omega_{\mathcal{K},s'}$ for all $ws' \in \Gamma^*_{\mathcal{K},s}$.Then, $\mathcal{K}, s \vDash \Phi$ iff $\Gamma^\omega_{\mathcal{K},s} \vDash \Phi$ and $\mathcal{K}, s \vDash \phi$ iff $\Gamma^\omega_{\mathcal{K},\pi_0}, \pi \vDash \phi$ can be easily proven by induction on the formula. Almost syntactically, $\Gamma^\omega_{\mathcal{K},s}$ can be replaced by s and the strategy can be removed in the path relation to obtain the classical definition. The two properties clearly hold in 1 to 7. In 8 and 9, with $E = \Gamma^\omega_{\mathcal{K},\pi_0}$, we can observe that $E \restriction \pi_0\pi_1 = \Gamma^\omega_{\mathcal{K},\pi_1}$ and $\pi_1 = (\pi^1)_0$, and that $E \restriction \pi^{\leq n} = \Gamma^\omega_{\mathcal{K},\pi_n}$ with $\pi_n = (\pi^n)_0$. The induction hypothesis can then be applied. □

Lemma 3. *For every $ws_0 \in S^+$ prefix in $E(\lambda)$, $E(\lambda) \restriction ws_0 = \{\mathrm{flat}(\pi) : \pi \in \Gamma_{\mathcal{U}(\mathcal{K},\lambda),ws_0}\}$ where* $\mathrm{flat}((ws_0)(ws_0s_1)(ws_0s_1s_2)\cdots) \coloneqq s_0s_1s_2\cdots$.

Proof. Executions in $\mathcal{U}(\mathcal{K},\lambda)$ are of the form $(ws_0)(ws_0s_1)(ws_0s_1s_2)\cdots$ where $s_0s_1\cdots$ is an execution in E. For the $\supseteq$ inclusion, take $\Gamma_{\mathcal{U}(\mathcal{K},\lambda),ws_0} \ni \pi = (ws_0)(ws_0s_1)(ws_0s_1s_2)\cdots$, whose $\mathrm{flat}(\pi) = s_0s_1s_2\cdots$ and $ws_0s_1s_2\cdots \in E$ since $s_{n+1} \in \lambda(s_n)$. Hence, $\mathrm{flat}(\pi) = s_0s_1\cdots \in E \restriction ws_0$ by definition. For the other $\subseteq$ inclusion, $s_0s_1\cdots \in E \restriction ws_0$ implies $ws_0s_1\cdots \in E$, so $\pi = (ws_0)(ws_0s_1)\cdots \in \Gamma_{\mathcal{U}(\mathcal{K},\lambda),ws_0}$ and $\mathrm{flat}(\pi) = s_0s_1\cdots$ is in the set. □

Proposition 2. *Given $(\mathcal{K}, E(\lambda))$ and a CTL* formula φ, $\mathcal{U}(\mathcal{K},\lambda) \vDash \varphi$ iff $\mathcal{K}, E(\lambda) \vDash \varphi$.*

Proof. We follow an inductive proof on the structure of CTL* formulae with the more general property $\mathcal{U}(\mathcal{K},\lambda), w \vDash \varphi$ iff $\mathcal{K}, E(\lambda) \restriction w \vDash \varphi$ for all $w \in S^+$. Path formulae need to be handled simultaneously, so the inductive property also includes $\mathcal{U}(\mathcal{K},\lambda), \pi \vDash \varphi$ iff $\mathcal{K}, E(\lambda) \restriction \pi_0, \mathrm{flat}(\pi) \vDash \varphi$ (in the lefthand side executions are successions of growing S^+ words while in the righthand side they are successions of S states). To facilitate reading, we will omit the $\mathcal{U}(\mathcal{K},\lambda)$ and $\mathcal{K}$ prefix when writing the satisfaction relations, and use E for $E(\lambda)$.

- (p, atomic propositions) By definition, $ws \vDash p$ iff $p \in \ell(s)$, and $E \restriction ws \vDash p$ iff $p \in \ell(s')$ for all $s'w' \in E \restriction ws = \{sw'' : wsw'' \in E\}$. Then, s' can only be s and both conditions coincide.
- ($\Phi_1 \wedge \Phi_2$) In the standard side, the conjunction is satisfied iff $w \vDash \Phi_i$ for both $i = 1, 2$. In the strategy side, this happens iff $E \restriction w \vDash \Phi_i$. By induction hypothesis on both Φ_i the equivalence holds.
- ($\neg\Phi$) The same inductive argument can be used for negation.
- ($\mathbf{A}\,\varphi$) This formula is satisfied iff $\pi \vDash \varphi$ for all $\pi \in \Gamma^\omega_{\mathcal{U}(\mathcal{K},\lambda),w}$ in the $\mathcal{U}(\mathcal{K},\lambda)$ side. In the strategy side, this is $E \restriction w, \rho \vDash \varphi$ for all $\rho \in E \restriction w$. Using Lemma 3, all these ρ are exactly those $\mathrm{flat}(\pi)$, and applying the induction hypothesis on φ, both statements are equivalent.

Let π be $(ws_0)(ws_0s_1)\cdots$, we then target the path satisfaction cases:

- ($\bigcirc\varphi$) We should prove that $\pi \vDash \bigcirc\varphi$ is equivalent to $E \restriction ws_0, s_0s_1\cdots \vDash \bigcirc\varphi$. Their definitions translate these to $\pi^1 \vDash \varphi$ and $(E \restriction ws_0) \restriction s_0s_1, (s_0s_1\cdots)^1 \vDash \varphi$. But they are equivalent by induction hypothesis on φ, since $(E \restriction ws_0) \restriction s_0s_1 = E \restriction ws_0s_1 = E \restriction \pi_1 = E \restriction (\pi^1)_0$ and $(s_0s_1\cdots)^1 = s_1s_2\cdots = \mathrm{flat}(\pi^1)$.

- $(\varphi_1 \mathbf{U} \varphi_2)$ The formula holds in the standard sense if there is an $n \in \mathbb{N}$ such that $\pi^n \vDash \varphi_2$ and for all k such that $0 \leq k < n$ then $\pi^k \vDash \varphi_1$. In the strategy side, the formula holds if again there is an $n \in \mathbb{N}$ such that $(E \restriction ws_0) \restriction s_0 \cdots s_n, s_n s_{n+1} \cdots \vDash \varphi_2$ and $(E \restriction ws_0) \restriction s_0 \cdots s_k, s_k s_{k+1} \cdots \vDash \varphi_1$ for all $0 \leq k < n$. Since $(E \restriction ws_0) \restriction s_0 \cdots s_k = E \restriction ws_0 \cdots s_k = E \restriction (\pi^k)_0$ and $s_k s_{k+1} \cdots = \mathrm{flat}(\pi^k)$ for all $k \in \mathbb{N}$, the induction hypothesis can be applied to φ_1 and φ_2 to conclude the property for $\varphi_1 \mathbf{U} \varphi_2$.
- (Φ) $\pi \vDash \Phi$ is defined as $\pi_0 \vDash \Phi$ in the standard sense, and $E \restriction ws_0, s_0 s_1 \cdots \vDash \Phi$ is $E \restriction ws_0 \vDash \Phi$ in the strategy case. Since $\pi_0 = ws_0$, both statements are related as in the induction property. The hypothesis on Φ itself can be applied, considering that state satisfaction is below path satisfaction in the induction order (we have never used this argument in reverse), and then they are equivalent.

A complete subset of CTL* constructors has been handled in the proof, the derived operators follow from the well-known semantic equivalences. □

Proposition 5. *Given $(\mathcal{K}, E)$ and a closed μ-calculus formula φ, $s \in [\![\varphi]\!]_{\mathcal{K},\eta}$ iff $\Gamma_{\mathcal{K},s} \in \langle\!\langle \varphi \rangle\!\rangle_{\mathcal{K},\xi}$ for any η and ξ.*

Proof. This property can be proven inductively, adding the variable valuations to the inductive property and the premise that $\eta(Z) \ni s$ iff $\Gamma_{\mathcal{K},s} \in \xi(Z)$ for all variables Z. For the initial φ, this premise is trivially satisfied since we can take $\eta(Z) = \varnothing = \xi(Z)$ regardless of the given two, since the formula is closed. We will not detail some trivial cases:

- (p) By definition, $s \in [\![p]\!]_\eta$ is $p \in \ell(s)$ and $\Gamma_s \in \langle\!\langle p \rangle\!\rangle_\xi$ is $\forall \pi \in \Gamma_s \; p \in \ell(\pi_0)$. Since Γ_s are the executions of $\mathcal{K}$ starting at s, $\pi_0 = s$ and both statements are equivalent.
- $(\langle a \rangle \varphi)$ $s \in [\![\langle a \rangle \varphi]\!]_\eta$ if there is an $s' \in S$ such that $s \to^a s'$ and $s' \in [\![\varphi]\!]_\eta$. On the other side, $\Gamma_s \in \langle\!\langle \langle a \rangle \varphi \rangle\!\rangle_\xi$ holds iff there is $saw \in \Gamma_s$ such that $\Gamma_s \restriction saw_0 = \Gamma_{w_0} \in \langle\!\langle \varphi \rangle\!\rangle_\xi$. The induction hypothesis with $s' = w_0$ lets us conclude the property.
- $(\nu Z.\varphi)$ $s \in [\![\nu Z.\varphi]\!]_\eta$ iff there is a set V such that $s \in V$ and $V \subseteq [\![\varphi]\!]_{\eta[Z/V]}$. In the strategy side, $\Gamma_s \in \langle\!\langle \nu Z.\varphi \rangle\!\rangle_\xi$ iff there is an F such that $\Gamma_s \in F$ and $F \subseteq \langle\!\langle \varphi \rangle\!\rangle_{\xi[Z/F]}$. Assuming there exists a V with these properties $(\Rightarrow)$, consider $F = \{\Gamma_s : s \in V\}$. In other words, $s \in V$ iff $\Gamma_s \in F$, so $\eta[Z/V]$ and $\xi[Z/F]$ are properly related. Hence, by induction hypothesis on φ, $\Gamma_s \in \langle\!\langle \varphi \rangle\!\rangle_{\xi[Z/F]}$ iff $s \in [\![\varphi]\!]_{\eta[Z/V]}$, so $F \subseteq \langle\!\langle \varphi \rangle\!\rangle_{\xi[Z/F]}$ as we wanted to prove. In the opposite direction $(\Leftarrow)$, assuming the existence of an F with the mentioned properties, consider $V = \{s \in S : \Gamma_s \in F\}$ and the proof is the same.

□

Proposition 6. *Given $(\mathcal{K}, E(\lambda))$ and a closed μ-calculus formula φ, $s \in [\![\varphi]\!]_{\mathcal{U}(\mathcal{K},\lambda),\eta}$ for $s\pi \in E$ iff $E \in \langle\!\langle \varphi \rangle\!\rangle_{\mathcal{K},\xi}$ for any η and ξ.*

Proof. Let us inductively prove the more general property that $[\![\varphi]\!]_\eta \ni w$ iff $E \restriction w \in \langle\!\langle \varphi \rangle\!\rangle_\xi$ provided that $\eta(Z) \ni w$ iff $E \restriction w \in \xi(Z)$ for all variables Z.

- (p) By definition, $ws \in [\![\varphi]\!]_\eta$ iff $p \in \ell(s)$, and $E \upharpoonright ws \in \langle\!\langle\varphi\rangle\!\rangle_\xi$ iff $p \in \ell(\pi_0)$ for all $\pi \in E \upharpoonright ws$. However, π_0 must be s since $E \upharpoonright ws = \{sw' : wsw' \in E\}$, so both sides are equivalent.
- (Z) The value of Z in both contexts is respectively $\eta(Z)$ and $\xi(Z)$, so the property directly follows from the assumption about these two functions.
- $(\varphi_1 \wedge \varphi_2)$ The standard definition says $[\![\varphi_1 \wedge \varphi_2]\!]_\eta = [\![\varphi_1]\!]_\eta \cap [\![\varphi_2]\!]_\eta$ and the strategy one is $\langle\!\langle\varphi_1 \wedge \varphi_2\rangle\!\rangle_\xi = \langle\!\langle\varphi_1\rangle\!\rangle_\xi \cap \langle\!\langle\varphi_2\rangle\!\rangle_\xi$. Hence, the property holds by induction hypothesis on φ_1 and φ_2.
- $(\neg\varphi)$ By definition, $[\![\neg\varphi]\!]_\eta = S^+ \setminus [\![\varphi]\!]_\eta$ and $\langle\!\langle\neg\varphi\rangle\!\rangle_\xi = \mathcal{P}(\Gamma_\mathcal{K}) \setminus \langle\!\langle\varphi\rangle\!\rangle_\xi$, so the property holds by induction hypothesis on φ.
- $(\langle a\rangle\varphi)$ $ws \in [\![\langle a\rangle\varphi]\!]_\eta$ iff there is an $(a, s') \in \lambda(ws)$ such that $wsas' \in [\![\varphi]\!]_\eta$ according to the standard definition of μ-calculus and the transition relation on $\mathcal{U}(\mathcal{K}, \lambda)$. On the other side, $E \upharpoonright ws \in \langle\!\langle\langle a\rangle\varphi\rangle\!\rangle_\xi$ iff there is a $w' \in (S \cup A)^\infty$ such that $saw' \in E \upharpoonright ws$ and $(E \upharpoonright ws) \upharpoonright saw'_0 = E \upharpoonright wsaw'_0 \in \langle\!\langle\varphi\rangle\!\rangle_\xi$.

  By definition of $E(\lambda)$, there is a $w' \in (S \cup A)^\infty$ such that $wsaw' \in E$ iff $(a, w'_0) \in \lambda(ws)$. Hence, by induction hypothesis on φ and taking $w'_0 = s'$, we conclude that the property holds.
- $(\nu Z.\varphi)$ According to the standard definition, $ws \in [\![\nu Z.\varphi]\!]_\eta$ iff there is a $V \subseteq S^+$ such that $V \subseteq [\![\varphi]\!]_{\eta[Z/V]}$ and $ws \in V$. According to our definition for strategies, $E \upharpoonright ws \in \langle\!\langle\nu Z.\varphi\rangle\!\rangle_\xi$ iff there is an $F \subseteq \mathcal{P}(\Gamma_\mathcal{K})$ such that $F \subseteq \langle\!\langle\varphi\rangle\!\rangle_{\xi[Z/F]}$ and $E \upharpoonright ws \in F$. Both implications can be proven like in the previous proposition, but taking $F = \{E \upharpoonright w : w \in V\}$ for a given V, and $V = \{w \in (S \cup A)^+ : E \upharpoonright w \in F\}$ for a given F.

□

Theorem 2. *$\mathcal{M}'_\alpha$ and $\mathcal{U}(\mathcal{M}, \lambda_{E(\alpha)})$ are bisimilar Kripke structures.*

Proof. Let $f : (T_\Sigma \cup A)^+ \to \mathcal{P}(\mathcal{XS})$ be defined by $f(t_0 a_1 t_1 \cdots a_n t_n) = \{q_n \in \mathcal{XS} : t_0 @ \alpha = q_0 \twoheadrightarrow^{a_1} \cdots \twoheadrightarrow^{a_n} q_n, \mathrm{cterm}(q_k) = t_k\}$. For any $w \in (T_\Sigma \cup A)^+$ and $Q \neq \varnothing$, $f(w)\,[\twoheadrightarrow]\,Q$ holds if and only if $\exists\, t \in T_\Sigma, a \in A \;\; Q = f(wat)$, since:

$$\begin{aligned} f(w)\,[\twoheadrightarrow]\,Q &\iff \exists\, t \in T_\Sigma, a \in A \;\; Q = \{q' \in \mathcal{XS} : q \in f(w), q \twoheadrightarrow^a q', \mathrm{cterm}(q') = t\} \\ &\iff \exists\, t \in T_\Sigma, a \in A \;\; Q = \{q' \in \mathcal{XS} : t_0 @ \alpha = q_0 \twoheadrightarrow^{a_1} \cdots \twoheadrightarrow^{a_n} q_n \twoheadrightarrow^a q', \\ &\qquad\qquad \mathrm{cterm}(q') = t, \mathrm{cterm}(q_k) = w_k\} = f(wat) \end{aligned}$$

The relation $R = \{(t_0 w, f(t_0 w)) : w \in (T_\Sigma \cup A)^*, f(t_0 w) \neq \varnothing\}$ is the bisimulation we are looking for. Clearly, $(t_0, \{t_0 @ \alpha\}) \in R$ and $\ell_{\mathrm{last}}(ws) = \ell(s) = \ell(\mathrm{cterm}(Q))$ if $(ws, Q) \in R$. Given two words $v, w \in (T_\Sigma \cup A)^+$, R only relates them to $f(v)$ and $f(w)$, respectively. $(v, w) \in U$ implies $w = vat$ by definition of U, and then $f(v)\,[\twoheadrightarrow]\,f(w)$ follows from the previous paragraph. Given two non-empty sets Q and Q' such that $Q\,[\twoheadrightarrow]\,Q'$, and a word w with $f(w) = Q$, we must find a w' such that $(w, w') \in R$ and $f(w') = Q'$. However, we already have it thanks to the previous paragraph and $f(w)\,[\twoheadrightarrow]\,Q'$, since there is some a and t such that $f(wat) = Q'$. It remains to prove that $(w, wat) \in U$, i.e. $(a, t) \in \lambda(w)$, but since there are no failed states in $\mathcal{M}'_\alpha$, any step of the semantics must be allowed by the strategy. □

References

[1] Alur, R., Henzinger, T.A., Kupferman, O., 2002. Alternating-time temporal logic. Journal of the ACM 49, 672–713. doi:10.1145/585265.585270.

[2] Andrei, O., Ibanescu, L., Kirchner, H., 2006. Non-intrusive formal methods and strategic rewriting for a chemical application, in: Futatsugi, K., Jouannaud, J., Meseguer, J. (Eds.), Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday, Springer. pp. 194–215. doi:10.1007/11780274_11.

[3] Asarin, E., Maler, O., Pnueli, A., 1995. Symbolic controller synthesis for discrete and timed systems, in: Antsaklis, P.J., Kohn, W., Nerode, A., Sastry, S. (Eds.), Hybrid Systems II, Proceedings of the Third International Workshop on Hybrid Systems, Ithaca, NY, USA, October 1994, Springer. pp. 1–20. doi:10.1007/3-540-60472-3_1.

[4] Atzei, N., Bartoletti, M., Lande, S., Yoshida, N., Zunino, R., 2019. Developing secure bitcoin contracts with BitML, in: Dumas, M., Pfahl, D., Apel, S., Russo, A. (Eds.), Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019, ACM. pp. 1124–1128. doi:10.1145/3338906.3341173.

[5] Baader, F., Nipkow, T., 1998. Term Rewriting and All That. Cambridge University Press. doi:10.1017/CBO9781139172752.

[6] Bae, K., Escobar, S., Meseguer, J., 2013. Abstract logical model checking of infinite-state systems using narrowing, in: van Raamsdonk, F. (Ed.), 24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. pp. 81–96. doi:10.4230/LIPIcs.RTA.2013.81.

[7] Bae, K., Meseguer, J., 2015. Model checking linear temporal logic of rewriting formulas under localized fairness. Science Computer Programming 99, 193–234. doi:10.1016/j.scico.2014.02.006.

[8] Baier, C., Katoen, J., 2008. Principles of Model Checking. MIT Press.

[9] Balland, E., Brauner, P., Kopetz, R., Moreau, P., Reilles, A., 2007. Tom: Piggybacking rewriting on Java, in: Baader, F. (Ed.), Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings, Springer. pp. 36–47. doi:10.1007/978-3-540-73449-9_5.

[10] Barendregt, H., 2014. The Lambda Calculus: Its Syntax and Semantics. volume 131. 2 ed., North Holland.

[11] Borovanský, P., Kirchner, C., Kirchner, H., Ringeissen, C., 2001. Rewriting with strategies in ELAN: A functional semantics. Int. J. Found. Comput. Sci. 12, 69–95. doi:10.1142/S0129054101000412.

[12] Bouhoula, A., Jouannaud, J.P., Meseguer, J., 1997. Specification and proof in membership equational logic, in: Bidoit, M., Dauchet, M. (Eds.), TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 14-18, 1997, Proceedings, Springer. pp. 67–92. doi:10.1007/BFb0030589.

[13] Bourdier, T., Cirstea, H., Dougherty, D.J., Kirchner, H., 2009. Extensional and intensional strategies, in: Fernández, M. (Ed.), Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009, Brasilia, Brazil, 28th June 2009, pp. 1–19. doi:10.4204/EPTCS.15.1.

[14] Bradfield, J.C., Walukiewicz, I., 2018. The mu-calculus and model checking, in: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (Eds.), Handbook of Model Checking. Springer, pp. 871–919. doi:10.1007/978-3-319-10575-8_26.

[15] Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E., 2008. Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming 72, 52–70. doi:10.1016/j.scico.2007.11.003.

[16] Bulling, N., Jamroga, W., 2014. Comparing variants of strategic ability: how uncertainty and memory influence general properties of games. Auton. Agents Multi Agent Syst. 28, 474–518. doi:10.1007/s10458-013-9231-3.

[17] Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C., 2019. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability, in: Vojnar, T., Zhang, L. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II, Springer. pp. 21–39. doi:10.1007/978-3-030-17465-1_2.

[18] Casagrande, A., 2020. pyModelChecking. URL: https://pypi.org/project/pyModelChecking.

[19] Cermák, P., Lomuscio, A., Murano, A., 2015. Verifying and synthesising multi-agent systems against

one-goal strategy logic specifications, in: Bonet, B., Koenig, S. (Eds.), Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA, AAAI Press. pp. 2038–2044. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9959.

[20] Chareton, C., Brunel, J., Chemouil, D., 2015. A logic with revocable and refinable strategies. Inf. Comput. 242, 157–182. doi:10.1016/j.ic.2015.03.015.

[21] Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A., 2002. NuSMV 2: An opensource tool for symbolic model checking, in: Brinksma, E., Larsen, K.G. (Eds.), Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings, Springer. pp. 359–364. doi:10.1007/3-540-45657-0_29.

[22] Clarke, E.M., Emerson, E.A., 1981. Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Kozen, D. (Ed.), Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981, Springer. pp. 52–71. doi:10.1007/BFb0025774.

[23] Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (Eds.), 2018. Handbook of Model Checking. Springer. doi:10.1007/978-3-319-10575-8.

[24] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C., 2020-10. Maude Manual v3.1. URL: http://maude.lcc.uma.es/maude31-manual-html/maude-manual.html.

[25] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L., 2007. All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. volume 4350 of *Lecture Notes in Computer Science*. Springer. doi:10.1007/978-3-540-71999-1.

[26] Clavel, M., Meseguer, J., 1996. Reflection and strategies in rewriting logic, in: Meseguer, J. (Ed.), Proceedings of the First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3-6, 1996, Elsevier. pp. 126–148. doi:10.1016/S1571-0661(04)00037-4.

[27] Clavel, M., Meseguer, J., 1997. Internal strategies in a reflective logic, in: Gramlich, B., Kirchner, H. (Eds.), Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction, pp. 1–12.

[28] David, A., Jensen, P.G., Larsen, K.G., Mikucionis, M., Taankvist, J.H., 2015. Uppaal Stratego, in: Baier, C., Tinelli, C. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings, Springer. pp. 206–211. doi:10.1007/978-3-662-46681-0_16.

[29] De Nicola, R., Vaandrager, F.W., 1990. Action versus state based logics for transition systems, in: Guessarian, I. (Ed.), Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings, Springer. pp. 407–419. doi:10.1007/3-540-53479-2_17.

[30] Diaconescu, R., Futatsugi, K., 2002. Logical foundations of CafeOBJ. Theor. Comput. Sci. 285, 289–318. doi:10.1016/S0304-3975(01)00361-9.

[31] Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L., 2016. Spot 2.0 - A framework for LTL and ω-automata manipulation, in: Artho, C., Legay, A., Peled, D. (Eds.), Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings, pp. 122–129. doi:10.1007/978-3-319-46520-3_8.

[32] Eker, S., Martí-Oliet, N., Meseguer, J., Pita, I., Rubio, R., Verdejo, A., 2021. Strategy language for Maude. URL: http://maude.ucm.es/strategies.

[33] Eker, S., Meseguer, J., Sridharanarayanan, A., 2004. The Maude LTL model checker, in: Gadducci, F., Montanari, U. (Eds.), Proceedings of the Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19-21, 2002, Elsevier. pp. 162–187. doi:10.1016/S1571-0661(05)82534-4.

[34] Emerson, E.A., Halpern, J.Y., 1986. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. J. ACM 33, 151–178. doi:10.1145/4904.4999.

[35] Fernández, M., Kirchner, H., Pinaud, B., 2019. Strategic port graph rewriting: an interactive modelling framework. Mathematical Structures in Computer Science 29, 615–662. doi:10.1017/S0960129518000270.

[36] Fernández, M., Kirchner, H., Pinaud, B., Vallet, J., 2018. Labelled graph strategic rewriting for social networks. J. Log. Algebraic Methods Program. 96, 12–40. doi:10.1016/j.jlamp.2017.12.005.

[37] Fischer, M.J., Ladner, R.E., 1979. Propositional dynamic logic of regular programs. J. Comput. Syst. Sci. 18, 194–211. doi:10.1016/0022-0000(79)90046-1.

[38] Giacomo, G.D., Maubert, B., Murano, A., 2020. Nondeterministic strategies and their refinement

in strategy logic, in: Calvanese, D., Erdem, E., Thielscher, M. (Eds.), Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020, pp. 294–303. doi:10.24963/kr.2020/30.

[39] Giacomo, G.D., Vardi, M.Y., 2013. Linear temporal logic and linear dynamic logic on finite traces, in: Rossi, F. (Ed.), IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013, IJCAI/AAAI. pp. 854–860. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997.

[40] He, T., Li, H., Qin, G., 2011. Model checking analysis of observational transition system with SMV, in: Liu, C., Chang, J., Yang, A. (Eds.), Information Computing and Applications - Second International Conference, ICICA 2011, Qinhuangdao, China, October 28-31, 2011. Proceedings, Part II, Springer. pp. 537–544. doi:10.1007/978-3-642-27452-7_73.

[41] Hennessy, M., Milner, R., 1980. On observing nondeterminism and concurrency, in: de Bakker, J.W., van Leeuwen, J. (Eds.), Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings, Springer. pp. 299–309. doi:10.1007/3-540-10003-2_79.

[42] Hidalgo-Herrero, M., Verdejo, A., Ortega-Mallén, Y., 2007. Using Maude and its strategies for defining a framework for analyzing Eden semantics, in: Antoy, S. (Ed.), Proceedings of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2006, Seattle, WA, USA, August 11, 2006, Elsevier. pp. 119–137. doi:10.1016/j.entcs.2007.02.051.

[43] IEEE, 2010. 1850-2010 - IEEE Standard for Property Specification Language (PSL). doi:10.1109/IEEESTD.2010.5446004.

[44] Jamroga, W., Murano, A., 2014. On module checking and strategies, in: Bazzan, A.L.C., Huhns, M.N., Lomuscio, A., Scerri, P. (Eds.), International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14, Paris, France, May 5-9, 2014, IFAAMAS/ACM. pp. 701–708. URL: http://dl.acm.org/citation.cfm?id=2615845.

[45] Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T., 2015. LTSmin: High-performance language-independent model checking, in: Baier, C., Tinelli, C. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings, Springer. pp. 692–707. doi:10.1007/978-3-662-46681-0_61.

[46] Kowalski, R.A., 1979. Algorithm = logic + control. Commun. ACM 22, 424–436. doi:10.1145/359131.359136.

[47] Kozen, D., 1983. Results on the propositional mu-calculus. Theor. Comput. Sci. 27, 333–354. doi:10.1016/0304-3975(82)90125-6.

[48] Lamport, L., 1980. "Sometime" is sometimes "not never" - on the temporal logic of programs, in: Abrahams, P.W., Lipton, R.J., Bourne, S.R. (Eds.), Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980, ACM Press. pp. 174–185. doi:10.1145/567446.567463.

[49] Larsen, K.G., Yi, W., Petterson, P., David, A., Nielsen, B., Skou, A., Håkansson, J., Rasmussen, J.I., Krcál, P., Larsen, U., Mikucionis, M., Mokrushin, L., et al., . UPPAAL. URL: http://www.uppaal.org/.

[50] Lechner, U., 1996. Object-oriented specifications of distributed systems in the mu-calculus and Maude, in: Meseguer, J. (Ed.), Proceedings of the First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3-6, 1996, Elsevier. pp. 385–404. doi:10.1016/S1571-0661(04)00048-9.

[51] Lescanne, P., 1990. Implementations of completion by transition rules + control: ORME, in: Kirchner, H., Wechler, W. (Eds.), Algebraic and Logic Programming, Second International Conference, Nancy, France, October 1-3, 1990, Proceedings, Springer. pp. 262–269. doi:10.1007/3-540-53162-9_44.

[52] Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J., 2014. Formal modeling and analysis of Cassandra in Maude, in: Merz, S., Pang, J. (Eds.), Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings, Springer. pp. 332–347. doi:10.1007/978-3-319-11737-9_22.

[53] Lomuscio, A., Qu, H., Raimondi, F., 2017. MCMAS: an open-source model checker for the verification of multi-agent systems. Int. J. Softw. Tools Technol. Transf. 19, 9–30. doi:10.1007/s10009-015-0378-x.

[54] Marin, M., Kutsia, T., 2006. Foundations of the rule-based system ρlog. J. Appl. Non Class. Logics 16, 151–168. doi:10.3166/jancl.16.151-168.

[55] Martí-Oliet, N., Meseguer, J., Verdejo, A., 2004. Towards a strategy language for Maude, in: Martí-Oliet, N. (Ed.), Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004, Elsevier. pp. 417–441. doi:10.1016/j.entcs.2004.06.020.

[56] Martín, O., Verdejo, A., Martí-Oliet, N., 2014. Model checking TLR* guarantee formulas on infinite systems, in: Iida, S., Meseguer, J., Ogata, K. (Eds.), Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi, Springer. pp. 129–150. doi:10.1007/978-3-642-54624-2_7.

[57] Meseguer, J., 1992. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155. doi:10.1016/0304-3975(92)90182-F.

[58] Meseguer, J., 2007. The Temporal Logic of Rewriting. techreport UIUCDCS-R-2007-2815. Department of Computer Science, University of Illinois at Urbana-Champaign. URL: http://hdl.handle.net/2142/11293.

[59] Meseguer, J., 2008. The temporal logic of rewriting: A gentle introduction, in: Degano, P., De Nicola, R., Meseguer, J. (Eds.), Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday, Springer. pp. 354–382. doi:10.1007/978-3-540-68679-8_22.

[60] Meseguer, J., 2012. Twenty years of rewriting logic. J. Log. Algebr. Program. 81, 721–781. doi:10.1016/j.jlap.2012.06.003.

[61] Mogavero, F., Murano, A., Perelli, G., Vardi, M.Y., 2014. Reasoning about strategies: On the model-checking problem. ACM Trans. Comput. Log. 15, 34:1–34:47. doi:10.1145/2631917.

[62] Neuhäußer, M.R., Noll, T., 2007. Abstraction and model checking of Core Erlang programs in Maude, in: Denker, G., Talcott, C. (Eds.), Proceedings of the 6th International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1-2, 2006, Elsevier. pp. 147–163. doi:10.1016/j.entcs.2007.06.013.

[63] Ogata, K., 2017. Model checking the iKP electronic payment protocols. J. Inf. Secur. Appl. 36, 101–111. doi:10.1016/j.jisa.2017.08.006.

[64] Ölveczky, P.C., 2014. Real-Time Maude and its applications, in: Escobar, S. (Ed.), 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, Springer. pp. 42–79. doi:10.1007/978-3-319-12904-4_3.

[65] Palomino, M., Martí-Oliet, N., Verdejo, A., 2005. Playing with Maude, in: Abdennadher, S., Ringeissen, C. (Eds.), Proceedings of the 5th International Workshop on Rule-Based Programming, RULE 2004, Aachen, Germany, June 1, 2004, Elsevier. pp. 3–23. doi:10.1016/j.entcs.2004.07.012.

[66] Pearl, J., 1984. Heuristics. Addison-Wesley series in artificial intelligence, Addison-Wesley.

[67] Pettorossi, A., Proietti, M., 2002. Program derivation = rules + strategies, in: Kakas, A.C., Sadri, F. (Eds.), Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I, Springer. pp. 273–309. doi:10.1007/3-540-45628-7_12.

[68] Pnueli, A., 1977. The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, IEEE Computer Society. pp. 46–57. doi:10.1109/SFCS.1977.32.

[69] Rubio, R., 2020a. Language bindings for Maude. URL: https://fadoss.github.io/maude-bindings.

[70] Rubio, R., 2020b. Unified Maude model-checking tool (umaudemc). URL: https://github.com/fadoss/umaudemc.

[71] Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A., 2019a. Model checking strategy-controlled rewriting systems, in: Geuvers, H. (Ed.), 4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany, Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 34:1–34:18. doi:10.4230/LIPIcs.FSCD.2019.31.

[72] Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A., 2019b. Parameterized strategies specification in Maude, in: Fiadeiro, J., Țuțu, I. (Eds.), Recent Trends in Algebraic Development Techniques, Springer. pp. 27–44. doi:10.1007/978-3-030-23220-7_2.

[73] Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A., 2020a. Metalevel transformation of strategies, in: 7th International Workshop on Rewriting Techniques for Program Transformation and Evaluation, WPTE 2020, Paris, France, pp. 1–10. URL: http://maude.ucm.es/wpte20/papers/WPTE_2020_rubio_et_al_strategies.pdf.

[74] Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A., 2020b. Simulating and model checking membrane systems using strategies in Maude, in: 7th International Workshop on Rewriting Techniques for Program Transformation and Evaluation, WPTE 2020, Paris, France, pp. 1–10. URL: http://maude.ucm.es/wpte20/papers/WPTE_2020_rubio_et_al_simulating.pdf.

[75] Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A., 2020c. Strategies, model checking and branching-time properties in Maude, in: Escobar, S., Martí-Oliet, N. (Eds.), Rewriting Logic and Its Applications - 13th International Workshop, WRLA 2020, Virtual Event, October 20-22, 2020, Revised Selected Papers, Springer. pp. 156–175. doi:10.1007/978-3-030-63595-4_9.

[76] Santos-García, G., Palomino, M., 2007. Solving Sudoku puzzles with rewriting rules, in: Denker, G., Talcott, C. (Eds.), Proceedings of the 6th International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1-2, 2006, Elsevier. pp. 79–93. doi:10.1016/j.entcs.2007.06.009.

[77] Terese, 2003. Term Rewriting Systems. Cambridge University Press.

[78] Thomas, D., Chakraborty, S., Pandya, P.K., 2008. Efficient guided symbolic reachability using reachability expressions. Int. J. Softw. Tools Technol. Transf. 10, 113–129. doi:10.1007/s10009-007-0057-7.

[79] Thomas, W., 1989. Computation tree logic and regular ω-languages, in: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (Eds.), Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings, Springer. pp. 690–713. doi:10.1007/BFb0013041.

[80] Verdejo, A., Martí-Oliet, N., 2011. Basic completion strategies as another application of the Maude strategy language, in: Escobar, S. (Ed.), Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011, pp. 17–36. doi:10.4204/EPTCS.82.2.

[81] Wang, B.Y., 2004. μ-calculus model checking in Maude, in: Martí-Oliet, N. (Ed.), Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004, Elsevier. pp. 135–152. doi:10.1016/j.entcs.2004.06.025.

[82] Zielonka, W., 1998. Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. 200, 135–183. doi:10.1016/S0304-3975(98)00009-7.