Safety Enforcement via Programmable Strategies in Maude*

M. Alpuente[a], D. Ballis[b], S. Escobar[a], D. Galán[a], J. Sapiña[a]

[a] *VRAIN, Universitat Politècnica de València, Camino de Vera s/n, Apdo 22012, Valencia, 46071, Spain*
[b] *DMIF, University of Udine, Via delle Scienze, 206, Udine, 33100, Italy*

Abstract

This work aims to provide a general mechanism for safety enforcement in rewriting logic computations. Our technique relies on an assertion-guided model transformation that leverages the newly defined Maude strategy language for ensuring rich safety policies in non-deterministic programs. The transformed system is guaranteed to comply with user-defined invariants that are expressed in a strategy-based, pattern-matching logic, thus preventing the concurrent system to reach any unsafe states. The performance and scalability of the technique is empirically evaluated and benchmarked on a set of realistic programs.

Keywords: Program safety, rewriting strategies, rewriting logic, Maude, formal methods

1. Introduction

Invariant enforcement is a proactive approach to guarantee software safety. Invariants, which are generally introduced and used for maintenance and verification purposes, are logical assertions that are held to be true during the

*This work was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215, grant RTI2018-094403-B-C32 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe", and by Generalitat Valenciana under grant PROMETEO/2019/098.

Email addresses: `alpuente@upv.es` (M. Alpuente), `demis.ballis@uniud.it` (D. Ballis), `sescobar@upv.es` (S. Escobar), `dgalan@upv.es` (D. Galán), `jsapina@upv.es` (J. Sapiña)

system execution so that they allow programming errors and incorrect implementations to be detected whenever some assertion is falsified.

Maude is a high-level programming language and system that supports functional, concurrent, logic, and object-oriented computations. A *rewrite theory* (or Maude program) combines a term rewriting system R, which specifies the concurrent transitions of a system, with an *equational theory* $\mathcal{E}$ that specifies system states as terms of an algebraic datatype. The equational theory $\mathcal{E}$ is generally split into a set E of equations and a set Ax of axioms (i.e., distinguished equations that specify algebraic laws such as commutativity, associativity, and unity for some program operators). The equations of E are implicitly oriented from left to right as rewrite rules and operationally used as simplification rules, while the axioms of Ax are mainly used for Ax-matching so that rewrite steps in $\mathcal{R}$ are performed *modulo* the equations and axioms of $E \uplus Ax$.

In this paper, we propose a technique for enforcing invariants on Maude programs that are equipped with *system assertions* and with an *execution strategy* that constricts computation paths to follow a series of actions (i.e., rule applications). A system assertion (a.k.a. *state assertion*) consists of a pair $\Pi\#\varphi$ where Π (the *state template*) is a term and φ (the *state invariant*) is a quantifier-free first-order formula with equality that defines a safety property that must hold on all of the system states that match (modulo equations and axioms) the state template Π.

Maude does not provide direct support for expressing execution invariants. However, it does provide support to control the execution process by means of the newly defined strategy language [1]. Given a set of system assertions $\mathcal{A}$ and an overly general Maude program $\mathcal{R}$ (i.e., a program that deploys all desired traces but may disprove some of the assertions), our transformation coerces $\mathcal{R}$ into a strategy-controlled program $\mathcal{R}'$ that prevents the system from reaching any states not satisfying the assertions of $\mathcal{A}$. The program $\mathcal{R}'$ is obtained by superposing (on top of $\mathcal{R}$) a strategy module that is automatically generated from $\mathcal{R}$ and $\mathcal{A}$ and compels the program execution in such a way that every run complies with the assertions. Moreover, whenever the safety policy $\mathcal{A}$ is explicitly coupled with a fixed path strategy P, the synthesized control strategy ensures the fulfillment of the overall policy.

The main advantage of our technique is that it leverages the Maude strategy language to enforce the system assertions so that it does not depend on hard-coded, ad-hoc safety checks. Program execution with constraining assertions is typically based on either externally monitoring invariant fulfilment

or somehow integrating the assertions into the system code. While such an integration avoids the heavy performance penalties that are introduced by system monitors, integrating programs and assertions is far from trivial since they are usually expressed in different formalisms and live at different levels of abstraction. Moreover, after the integration, the invariants typically get lost amidst the code, which jeopardizes its location, tracing, and maintainance [2]. In our approach, invariants are expressed in separate Maude modules from the system code, thus naturally avoiding any mixing of invariants and code. As another advantage, more refined versions of a program can be incrementally built by simply adding new logical constraints into the given assertion set without any programming effort. This makes it possible to adapt existing Maude programs to predefined safety policies and allows the inexperienced user to largely forget about Maude (and Maude strategy language) syntax and semantics.

This paper improves the preliminary approach of [3, 4], where an automated correction methodology for Maude programs is formalized with respect to a set of assertions. The correction was essentially achieved by transforming the original program rules into guarded program rules whose conditions, which are expressed by means of new (equationally-defined) functions, ensure the satisfiability of all of the assertions. The technique in [3, 4] applies to *topmost rewrite theories* (i.e., Maude programs in which terms can only be rewritten at the root position), and it not only modifies the program rules but also transforms the underlying equational theory by introducing the new equations and operators. This is in contrast with the new transformation approach for safety enforcement formalized in this paper, which allows more sophisticated policies to be considered on a wider class of strategy-controlled programs that are not necessarily topmost. Moreover, our novel transformation is conceptually simpler as it does not change the underlying equational theory. This allows the synthesized program refinements to not only be more akin to the original program but also more efficient.

This paper is organized as follows. After some technical preliminaries in Section 2, we introduce a running example that we use to illustrate the kind of software adaptation that we aim to produce automatically. Section 3 recalls the Maude strategy language that can be used to restrain and/or guide the rewrite process. Section 4 shows how safety policies can actually be defined as system assertions in our rewriting setting. Section 5 formalizes our safety enforcement methodology, while Section 6 illustrates and experimentally evaluates the STRASS system, which implements our proposed

transformation. Section 7 describes a typical safety-enforcement session in STRASS, and finally Section 8 concludes.

2. Preliminaries

Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting [5] and Rewriting Logic [6]. Some familiarity with the Maude language [7] is also required.

We consider a *signature* Σ of typed operators (i.e, function symbols). The operators type structure is encoded as a poset $(S, <)$ that models the usual subsort relation over a set of sort S [7]. We also consider an S-sorted family $\mathcal{V} = \{\mathcal{V}_s\}_{s \in S}$ of disjoint variable sets, and we write $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ for the corresponding term algebras. The set of variables that occur in a term t is denoted by $\mathcal{V}ar(t)$. We assume *pre-regularity* of the signature Σ: for each operator declaration $f : s_1 \times \cdots \times s_n \to s$, and for the set S_f containing all sorts s' that appear in operator declarations of the form $f : s'_1, \cdots, s'_n \to s'$ in Σ such that $s_i \leq s'_i$ for $1 \leq i \leq n$, the set S_f has a least sort. Thanks to pre-regularity of Σ, each term t in $\tau(\Sigma, \mathcal{V})$ has a *unique least sort* that is denoted by $ls(t)$.

A *position* w in a term t is represented by a sequence of natural numbers that addresses a subterm of t. Given a term t, we let $\mathcal{P}os(t)$ denote the set of positions of t. By $t_{|w}$, we denote the *subterm* of t at position w. A *substitution* $\sigma \equiv \{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$ is a mapping from the set of variables $\mathcal{V}$ to the set of terms $\tau(\Sigma, \mathcal{V})$, which is equal to the identity everywhere except for a set of variables $\{x_1, \ldots, x_n\}$.

A labelled *conditional* equation, or simply (*conditional*) equation, is an expression of the form $[l] : \lambda = \rho \ \mathit{if} \ C$, where l is a label (i.e., a name that identifies the equation), $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$, and C is a (possibly empty) sequence $c_1 \wedge \ldots \wedge c_n$, where each c_i is a Boolean expression[1]. When the condition C is empty, we simply write $[l] : \lambda = \rho$.

A labelled *conditional* rewrite rule, or simply (*conditional*) rule, is an expression of the form $[l] : \lambda \Rightarrow \rho \ \mathit{if} \ C$, where l is a label, $\lambda, \rho \in \tau(\Sigma, \mathcal{V})$, and C is a (possibly empty) conjunction of Boolean expressions $c_1 \wedge \ldots \wedge c_n$. When the condition C is empty, we simply write $[l] : \lambda \Rightarrow \rho$. By *label*$(r)$,

[1] Actually, Maude supports different kinds of conditions in equations such as equational conditions, membership tests, and matching conditions. Nonetheless, all of them can be interpreted as Boolean expressions whose canonical form is a truth value.

we denote the label of the rewrite rule r, while *labels*(R) denotes the set of rewrite rule labels for the rules in R. When no confusion can arise, rule and equation labels $[l]$ are often omitted.

An *equational theory* $\mathcal{E}$ is a pair (Σ, E), where Σ is a signature, $E = \Delta \cup Ax$ with Δ being a collection of (oriented) conditional equations and Ax a collection of equational axioms such as associativity, commutativity, and unity that can be associated with any binary operator of Σ. The equational theory $\mathcal{E}$ induces a congruence relation on the term algebra $\tau(\Sigma, \mathcal{V})$, which is denoted by $=_E$.

A *conditional rewrite theory* (or simply, rewrite theory) is a triple $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory and R is a set of conditional rewrite rules.

In a rewrite theory $\mathcal{R} = (\Sigma, \Delta \cup Ax, R)$, computations evolve by rewriting terms using the *equational rewriting* relation $\rightarrow_{R,E}$, which applies the rewrite rules in R to terms *modulo the equational theory* $E = (\Sigma, \Delta \cup Ax)$ [6]. A *computation* in $\mathcal{R} = (\Sigma, E, R)$ is hence a (possibly infinite) rewrite sequence $t_0 \rightarrow_{R,E} t_1 \ldots \rightarrow_{R,E} t_n$, where terms t_i are called *states*. States are irreducible forms (modulo Ax) computed by first applying a rewrite rule of R and then normalizing the resulting term by using the equations in Δ as simplification rules. We often use the notation $t \xrightarrow{r}_{R,E} t'$ to make the rule r used in a rewrite step explicit. Given any rewrite relation $\rightarrow$, by $\rightarrow^*$ (resp., $\rightarrow^+$), we denote the usual transitive and reflexive (resp., transitive) closure of $\rightarrow$.

The transition space of all computations in $\mathcal{R}$ from the initial state t_0 can be represented as a *computation tree* $\mathcal{T}_{\mathcal{R}}(t_0)$ whose branches specify all of the computations in $\mathcal{R}$ that originate from t_0.

We also consider a natural partition of the rewrite theory signature as $\Sigma = \mathcal{D} \uplus \Omega$, where Ω are the *constructor* symbols, which are used to define (irreducible) data values, and $\mathcal{D} = \Sigma \setminus \Omega$ are the *defined* symbols, which are evaluated away via equational simplification. Terms in $\tau(\Omega, \mathcal{V})$ are called *constructor* terms.

Nondeterministic as well as concurrent software can be formalized as rewrite theories which can be encoded in Maude. Therefore, throughout the paper, a rewrite theory is also called a Maude program. Let us see an example.

Example 2.1. *Consider a Maude program* $\mathcal{R}_{\texttt{DAM}}$ *that models a simplified, non-deterministic dam controlling system to monitor and manage the water*

volume of a given basin[2]. $\mathcal{R}_{\texttt{DAM}}$ *is a slight adaptation of the dam controller in [3]. In the program code, variable names are fully capitalized.*

We assume that the dam is provided with three spillways called `s1`, `s2`, *and* `s3`, *each of which has four possible aperture widths of increasing discharge capacity* `close`, `open1`, `open2`, `open3`. *Each spillway is formally specified by a term* `[S,O]`, *where* $\texttt{S} \in \{\texttt{s1}, \texttt{s2}, \texttt{s3}\}$ *and* $\texttt{O} \in \{\texttt{close}, \texttt{open1}, \texttt{open2}, \texttt{open3}\}$. *A global spillway configuration is a multiset* `[s1,O1] [s2,O2] [s3,O3]` *that groups together the three spillways by means of the usual associative and commutative infix, union operator* `__` *(written in mixfix notation with empty syntax) whose identity is the constant* `empty`. *System states are defined by terms of the form* `{ SC | V | T }` *where* `SC` *is a global spillway configuration,* `V` *is a rational number that indicates the basin water volume (in* m^3*), and* `T` *is a natural number that timestamps the current configuration in the style of Lamport clocks, i.e., defines a partial causal ordering of events.*

```
eq inflow = 3000 .                --- Basin water inflow
eq aperture(close) = 0 .          --- Outflow for a closed spillway
eq aperture(open1) = 200 .        --- Outflow for aperture width open1
eq aperture(open2) = 400 .        --- Outflow for aperture width open2
eq aperture(open3) = 1200 .       --- Outflow for aperture width open3

--- Basin water outflow for a given spillway configuration
eq outflow(empty) = 0 .
eq outflow([S,O] SS) = aperture(O) + outflow(SS) .
```

Figure 1: Equational definition of basin inflow and outflow.

Figure 1 shows the equational specification that formalizes basin water inflow and outflow. To keep the exposition simple, we assume that the basin water inflow is constant, while the basin outflow depends on the width of the spillway apertures and can be computed as the sum of the outflows of each spillway in the spillway configuration. Note that inflow and outflow values are measured in m^3/min *and are hard-coded into the dam controller. More realistic scenarios could be easily defined by sophisticating the basin inflow and outflow functions.*

The system dynamics is specified by the eight rewrite rules in Figure 2,

[2]A complete Maude specification of the dam controller is available at the STRASS website at `http://safe-tools.dsic.upv.es/strass`.

```
rl [nocmd] : { SC | V | T } => { SC | V | T } .
rl [openC-1] : { [S,close] SS | V | T } => { [S,open1] SS | V | T } .
rl [open1-2] : { [S,open1] SS | V | T } => { [S,open2] SS | V | T } .
rl [open2-3] : { [S,open2] SS | V | T } => { [S,open3] SS | V | T } .
rl [close1-C] : { [S,open1] SS | V | T } => { [S,close] SS | V | T } .
rl [close2-1] : { [S,open2] SS | V | T } => { [S,open1] SS | V | T } .
rl [close3-2] : { [S,open3] SS | V | T } => { [S,open2] SS | V | T } .
crl [volume] : { SC | V | T } => { SC | V' | (T + deltaT) }
               if V' := (V + inflow * deltaT) - (outflow(SC) * deltaT) .
```

Figure 2: (Conditional) rewrite rules for the dam controlling system.

which implement system state transitions. The `openX-Y` *rewrite rules progressively increment the aperture width of a given spillway (e.g., the rule* `open1-2` *increases the aperture of the spillway* `S` *from level* `open1` *to level* `open2`*). Dually,* `closeX-Y` *rewrite rules progressively decrement the aperture width of a spillway. The rule* `nocmd` *specifies the empty command which basically states that no action is taken on the spillway configuration by the dam controller at time instant* `T`*. These eight rules, called* aperture command *rules, implement instantaneous spillway modifications that do not change the time instant or the basin water volume.*

The temporal evolution of the basin water volume is specified by the conditional rewrite rule `volume` *that computes the volume* `V'` *at time* `T + deltaT`*, given the input volume* `V` *at time* `T`*. The parameter* `deltaT` *is measured in time units (e.g., minutes) and can be set by the user. The volume computation changes the input volume* `V` *by adding the water inflow and subtracting the corresponding water outflow over the* `deltaT` *interval.*

It is worth noting that $\mathcal{R}_{\mathtt{DAM}}$ *does not implement any spillway management policy that safely restricts and guides the applications of the aperture command rules. Furthermore,* $\mathcal{R}_{\mathtt{DAM}}$ *does not encode a fair interleaving between the applications of the rule* `volume` *and the remaining aperture command rules; hence, there is no guarantee that a new basin water volume is computed after each spillway aperture change. These two facts might lead to anomalous computations that (i) may reach potentially hazardous system states (e.g., an extremely high water volume), and also (ii) may include meaningless (and possibly infinite) sequences of aperture command rule applications. In the rest of the paper, we show how Maude strategies and safety policies can be used together to solve (i) and (ii).*

3. The Maude Strategy Language

Rule-based rewriting is a highly nondeterministic process where, at every step, many rules could be applied at various positions. In order to provide a finer control on rule application, Maude 3.0 introduced a new specification layer on top of the standard system modules by means of Maude's *strategy language* [1, 7]. This (sub-)language allows the user to control the rewriting process respecting the *separation of concerns* principle since the rewrite theory is not modified in any way and could be executed according to different strategies.

For the purposes of this work, we consider the proper subset of the Maude strategy language that contains the primitive operators that allow the safety properties of our framework to be expressed.

Definition 3.1. [1] *A strategy (or strategy expression)* S *for a rewrite theory* $\mathcal{R}$ *is an expression that is built using the following abstract syntax.*

$$\texttt{fail} \mid \texttt{idle} \mid \texttt{l} \mid \texttt{amatch}\ P\ \texttt{s.t.}\ C \mid \alpha\ ;\ \beta \mid (\alpha \mid \beta) \mid \alpha\ ?\ \beta\ :\ \gamma \mid \alpha^* \mid \alpha^+ \mid \alpha! \mid \texttt{all} \mid \texttt{not}(\alpha)$$

where `l` *is a rewrite rule label,* α, β, γ *are strategies,* $P \in \tau(\Omega, \mathcal{V})$ *is a (possibly non-ground) constructor term, and* C *is an equational condition.*

A *path* strategy P is any strategy expression that only includes occurrences of rule labels and operators $\alpha\ ;\ \beta$, α^*, α^+, $\alpha!$, $(\alpha \mid \beta)$, `all`. We use path strategies to represent admissible sequences of rule applications which are used to guide the program space exploration. We also use the remaining strategy constructors as auxiliary operators to implement our transformation technique in Section 5. We denote the set of all the path strategies that can be built in $\mathcal{R}$ by $PStr(\mathcal{R})$.

The semantic interpretation of a strategy is defined in [1, 7] as a transformation from a term to a set of terms. Given the rewrite theory $\mathcal{R} = (\Sigma, E, R)$, the semantics of a strategy S, denoted as $[\![\mathsf{S}]\!]$, is a function $\tau(\Sigma) \to 2^{\tau(\Sigma)}$ that identifies those system states $t' \in [\![\mathsf{S}]\!](t)$ stemming from $t \in \tau(\Sigma)$ by finitely controlled executions of the rules of $\mathcal{R}$. Rules to be executed are given by the strategy expression, which is built by combining the strategy operators of Definition 3.1. For the sake of completeness, we recall the strategy constructor semantics of [1, 7] in the following definition.

Definition 3.2. [7] *Let* $\mathcal{R} = (\Sigma, E, R)$ *be a rewrite theory. Let* $t \in \tau(\Sigma)$*. The strategy operators of Definition 3.1 have the following semantics.*

$$[\![\texttt{idle}]\!](t) = \{t\} \quad [\![\texttt{fail}]\!](t) = \emptyset \quad [\![\alpha|\beta]\!](t) = [\![\alpha]\!](t) \cup [\![\beta]\!](t) \quad [\![\alpha;\beta]\!](t) = \bigcup_{t' \in [\![\alpha]\!](t)} [\![\beta]\!](t')$$

$$[\![\texttt{l}]\!](t) = \{t' \in \tau(\Sigma) \mid t \xrightarrow{r}_{R,E} t' \textit{ for any } r \in R \textit{ s.t. label}(r) = \texttt{l}\}$$

$$[\![\texttt{amatch } P \texttt{ s.t. } C]\!](t) = \begin{cases} \{t\} & \textit{if } \theta \in \{\sigma \mid \exists w \in \mathcal{P}os(t) \textit{ s.t. } P\sigma =_E t_{|w}\} \wedge C\theta \textit{ holds} \\ \emptyset & \textit{otherwise} \end{cases}$$

$$[\![\alpha?\beta : \gamma]\!](t) = \begin{cases} [\![\alpha;\beta]\!](t) & \textit{if } [\![\alpha]\!](t) \neq \emptyset \\ [\![\gamma]\!](t) & \textit{if } [\![\alpha]\!](t) = \emptyset \end{cases}$$

$$[\![\alpha^*]\!](t) = [\![\texttt{idle} \mid \alpha;\alpha^*]\!](t) \quad [\![\alpha^+]\!](t) = [\![\alpha;\alpha^*]\!](t) \quad [\![\alpha!]\!](t) = [\![\alpha^*;(\alpha \ ? \ \texttt{fail} : \texttt{idle})]\!](t)$$

$$[\![\texttt{all}]\!](t) = \{t' \in \tau(\Sigma) \mid t \xrightarrow{r}_{R,E} t' \textit{ for any } r \in R\} \quad [\![\texttt{not}(\alpha)]\!](t) = [\![\alpha \ ? \ \texttt{fail} : \texttt{idle}]\!](t)$$

Roughly speaking, the `idle` strategy always succeeds and returns the given state t unaltered as the only result. In contrast, the `fail` strategy always fails for any given state in the sense that it produces no result (i.e., an empty set of resulting states). The rule label construct provides a fine control over rewrite rule applications: by specifying a strategy with the rule label `l`, only rules identified by `l` are applied to t (rules marked as `nonexec` are excluded) delivering the set of terms obtained by rewriting t with the rule `l`.[3] The sequential operator $\alpha \ ; \beta$ models strategy concatenation by first applying α to the initial state t and then applying β to the states yielded by α. The choice operator $\alpha \mid \beta$ executes α or β on the state t, and the results are both those of α and those of β. The `amatch` P `s.t.` C construct implements testing features: the test succeeds delivering the singleton $\{t\}$, when (i) there is a match modulo E between the pattern P and a subterm of t with matching substitution θ and (ii) the instantiated equational condition $C\theta$ holds. Otherwise, the test fails and $\emptyset$ is delivered as the result.

Conditional strategies are implemented via the operator $\alpha?\beta : \gamma$. This operator executes α and then β on its results, but if α does not produce any, it executes γ on the initial term. That is: α is the condition; β is the strategy for the positive branch, which applies to the reduced terms computed by α; and γ is the strategy for the negative branch, which is applied to the initial term only if α fails.

[3]In the case when the rule to be applied is conditional and includes rewrite expressions, strategies for each rewrite expression must also be provided.

The strategy $\texttt{not}(\alpha)$ fails when the strategy α succeeds, and it succeeds (as the idle strategy) when α fails. The strategies α^*, α^+, $\alpha!$ respectively specify the transitive and reflexive closure of any strategy α, the transitive closure of α, and the normalization operator $\alpha! = \alpha^*; \texttt{not}(\alpha)$ that repeatedly applies α until it cannot be further applied. The `all` strategy triggers a single rewrite step on t with all the rules available in R, yielding the corresponding set of reduced terms.

Given a strategy expression S, we can formalize the notion of computation tree of $\mathcal{R}$ w.r.t. S by resorting to the small-step operational semantics of [1] that determines exactly which computation paths are described by a strategy expression.

Definition 3.3. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and let S be a strategy for $\mathcal{R}$. Let $t_0 \in \tau(\Sigma)$. The* computation tree $\mathcal{T}^{\mathsf{S}}_{\mathcal{R}}(t_0)$ *for t_0, w.r.t $\mathcal{R}$ and S, is a sub-tree of $\mathcal{T}_{\mathcal{R}}(t)$ whose branches represent computations of the form $t_0 \rightarrow_{R,E} t_1 \ldots \rightarrow_{R,E} t_n$, $n \geq 0$, where each t_i is computed by the strategy according to the small-step semantics of [1] for $0 \leq i \leq n$.*

Note that Definition 3.3 ensures that t_n is reachable from state t_0 using the strategy S whenever $t_n \in [\![\mathsf{S}]\!](t_0)$. Intuitively, $\mathcal{T}^{\mathsf{S}}_{\mathcal{R}}(t)$ refines the computation tree $\mathcal{T}_{\mathcal{R}}(t)$ by getting rid of those computations of $\mathcal{T}_{\mathcal{R}}(t)$ that do not satisfy the strategy S, thereby reducing the search space of the rewrite theory $\mathcal{R}$.

For convenience sake, we introduce the following notation. Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and a set L of rewrite rule labels, we define the macro $\texttt{all-}(L) = l_1 \mid \ldots \mid l_n$ where $\{l_1, \ldots, l_n\} = \mathit{labels}(R) \setminus L$.

Example 3.4. *Let us consider the Maude program $\mathcal{R}_{\texttt{DAM}}$ of Example 2.1 together with the path strategy* `all-(volume)` *whose definition is*

```
(nocmd | openC-1 | open1-2 | open2-3 | close1-C | close2-1 | close3-2)
```

Roughly speaking, given a term t, the strategy `all-(volume)` *rewrites t by using only aperture command rules, while the* `volume` *rule cannot be applied.*

On top of `all-(volume)`*, we can build the more complex path strategy* `alt` *that enforces a volume computation after any application of an aperture command rule. This can be easily achieved by the strategy*

```
alt = (all-(volume) ; volume) +
```

This way, any computation in $\mathcal{T}^{\texttt{alt}}_{\mathcal{R}_{\texttt{DAM}}}(t)$ correctly alternates aperture commands with volume updates. Note that such a rule alternation cannot be

directly achieved by our previous transformation [3] since it does not support Maude strategies. Indeed, in [3], we define ad-hoc *and more complex data structures for states in order to enforce safe rule interleavings, while alternation is hardcoded into the rewrite rule definition itself.*

Given the strategy S, the *computation tree* $\mathcal{T}_{\mathcal{R}}^{\mathsf{S}}(t)$ can be explored by using the Maude search command `srewrite` (`srew`, for short) whose (simplified) syntax is `srew` $[n]$ t `using` S, where n is an optional parameter representing a bound on the number of *solutions* (i.e., reachable terms) $t' \in [\![\mathsf{S}]\!](t)$ to be computed.

Example 3.5. *Given the Maude program* $\mathcal{R}_{\texttt{DAM}}$ *of Example 2.1 and the path strategy* `alt` *of Example 3.4,*

$$t = \texttt{\{[s1,close] [s2,close] [s3,close] | 5000 | 0\}}$$

is an initial state modeling a dam configuration at time 0 *with three closed spillways and a basin volume equal to 5000* m^3*. Then, the following Maude command generates the first two states at time* 5 *which are reachable from t using* `alt`*.*

```
Maude> srew [2] {[s1,close] [s2,close] [s3,close] | 5000 | 0 } using alt .

Solution 1
rewrites: 74 in 0ms cpu (1ms real) (165919 rewrites/second)
result State: {[s1,close] [s2,close] [s3,close] | 20000 | 5}

Solution 2
rewrites: 74 in 0ms cpu (1ms real) (157782 rewrites/second)
result State: {[s1,open1] [s2,close] [s3,close] | 19000 | 5}
```

Solution 1 is obtained by first applying the `nocmd` *rule, which does not change the spillway configuration, and then computing the new basin volume. Solution 2 represents a state where spillway* `s1` *has been opened and a new basin volume has been computed according to the new spillway configuration.*

Path strategies provide a fine control on rule applications and are a valuable tool to restrain and/or guide the rewrite process. However, they are not designed to enforce safety properties on system states. Indeed, given a computation $t_1 \rightarrow_{R,E} t_2 \rightarrow_{R,E} \ldots \rightarrow_{R,E} t_n$ in $\mathcal{T}_{\mathcal{R}}^{\mathsf{P}}(t)$, we can only infer that t_n is reachable from t_1 using the path strategy P, but there is no way to discover whether some state t_i in the considered computation violates a safety constraint.

Example 3.6. *Given the Maude program $\mathcal{R}_{\mathtt{DAM}}$ of Example 2.1 and the path strategy* `alt` *of Example 3.4,*

$$t = \texttt{\{[s1,open3] [s2,close] [s3,close] | 49989000 | 0\}}$$

is an initial state with an extremely high and potentially dangerous water volume. By executing

```
srew [2] [s1,open3] [s2,close] [s3,close] | 49989000 | 0  using alt .
```

we explore the first two states that are reachable from t using the alt *strategy yielding the following two solutions:*

```
Solution 1
rewrites: 76 in 0ms cpu (0ms real) (737864 rewrites/second)
result State: {[s1,open3] [s2,close] [s3,close] | 50000000 | 5}

Solution 2
rewrites: 76 in 0ms cpu (0ms real) (622950 rewrites/second)
result State: {[s1,open3] [s2,open1] [s3,close] | 49999000 | 5}
```

Note that the first solution keeps the spillway configuration untouched and increases the water volume reaching the critical threshold of 50 million m^3.

In the next section, we formalize an assertion language that complements path strategies by allowing safety properties to be specified and enforced on system states in order to get rid of those computations that exhibit unsafe behaviors.

4. An Assertion Language for Specifying Safety Policies

A *safety policy* for a rewrite theory $\mathcal{R}$ is given as a set $\mathcal{A}$ of system assertions that specify properties of the software system encoded by $\mathcal{R}$. The assertions are expressed as (quantifier-free) first-order formulas (predicates) that are built using the usual Boolean operators. The truth values are given by the formulas *true* and *false*, and the usual conjunction (*and*), disjunction (*or*), and negation (*not*)[4] are used to express composite properties. Variables in the formulas are not quantified. Besides Boolean operators, φ may

[4]Note that the *not* operator implements the usual Boolean negation whose semantics differs from the semantics of the `not` strategy operator of Definition 3.2.

include Maude built-in operators as well as user-defined predicates/functions that can be equationally specified. Satisfiability of formulas is checked via equational rewriting. More specifically, a formula φ holds in a rewrite theory $\mathcal{R}$ if it can be reduced to *true* using the equations of $\mathcal{R}$ oriented from left to right as rewrite rules.

Definition 4.1. *Given a rewrite theory $\mathcal{R} = (\Omega \cup \mathcal{D}, E, R)$, with $E = \Delta \cup Ax$, a state assertion has the form $\Pi \# \varphi$, where Π is a constructor term in $\tau(\Omega, \mathcal{V})$ and φ is a quantifier-free first-order logic formula.*

Operationally, a state assertion $\Pi \# \varphi$ defines a generic safety property for a state t that specifies a logic invariant φ which must be enforced in any subterm of t that is an instance (modulo Ax) of Π.

Definition 4.2. *Let $\mathcal{R} = (\Sigma, E, R)$, with $E = \Delta \cup Ax$, be a rewrite theory. Let t be a state in $\mathcal{R}$. A state assertion $\Pi \# \varphi$ holds in t if for every $w \in \mathcal{P}os(t)$ and for every substitution σ, if $t|_w =_{Ax} \Pi\sigma$ then $\varphi\sigma$ evaluates to true. We also say that t is* safe *w.r.t. $\Pi \# \varphi$.*

It is worth noting that a state assertion $\Pi \# \varphi$ trivially holds in t, when there is no subterm of t that matches Π modulo Ax.

Given a set $\mathcal{A}$ of state assertions and a state t, t is *safe* w.r.t. $\mathcal{A}$ iff for each $a \in \mathcal{A}$, a holds in t.

Example 4.3. *Let us consider the user-defined function* `openSpillways(SC)` *that returns the number of open spillways in the spillway configuration* `SC`*, whose equational definition is*

```
eq openSpillways(empty) = 0 .
eq openSpillways([S,O] SC) = if (O =/= close) then (1 + openSpillways(SC))
                                             else openSpillways(SC) fi .
```

and the safety policy $\mathcal{A}_{\texttt{DAM}}$ of Figure 3 for the dam controller of Example 2.1 that specifies some safety constraints to prevent basin critical situations.

More specifically, assertion `a1` *states that, in every system state, the basin water volume must be less than 50 million m^3 to avoid dam bursts and potentially disastrous floods. Assertion* `a2` *specifies that, whenever the basin water volume is greater than 40 million m^3, all of the spillways must be open and the aperture width of at least one spillway must be maximal (level* `open3`*). Assertion* `a3` *requires the closure of all of the spillways when the basin water*

```
(a1) { SC | V | T } # (V < 50000000)
(a2) { [ S1,O1 ] [ S2,O2 ] [ S3,O3 ] | V | T} # (V > 40000000) implies (
                     (O1 == open3 and O2 =/= close and O3 =/= close) or
                     (O2 == open3 and O1 =/= close and O3 =/= close) or
                     (O3 == open3 and O1 =/= close and O2 =/= close)
                    )
(a3) { SC | V | T } # (V < 10000000) implies (openSpillways(SC) == 0)
(a4) { SC | V | T } # ((V >= 10000000) and (V <= 40000000))
                                     implies (openSpillways(SC) == 2)
```

Figure 3: Safety policy $\mathcal{A}_{\texttt{DAM}}$ for the dam controller $\mathcal{R}_{\texttt{DAM}}$.

volume is particularly low (10 million m^3). Finally, assertion `a4` *specifies the spillway handling for an intermediate water volume (10 million $m^3 \leq \texttt{V} \leq 40$ million m^3); in this scenario, we require exactly two spillways to be constantly open.*

5. Computing Safe Maude Programs

Program transformation techniques have been successfully applied to program refinement, specialization, and optimization. Inferring program transformations has many uses, such as bug fixing, refactoring, and program optimization [8].

In our approach, given a Maude program $\mathcal{R}$ together with a safety policy $\mathcal{A}$ and a path strategy P for $\mathcal{R}$, the program $\mathcal{R}$ is transformed into a program $\mathcal{R}'$ that restricts the computations of $\mathcal{R}$ to a subset that is deemed safe w.r.t. $\mathcal{A}$. More specifically, the transformed program $\mathcal{R}'$ overlays a strategy module on top of $\mathcal{R}$. This module encodes each state assertion of $\mathcal{A}$ into equivalent strategy expressions and smoothly integrates them into P in order to drive the system execution in such a way that (i) unsafe states cannot be reached, (ii) unsafe computations are pruned away, and (iii) all computations that satisfy $\mathcal{A}$ are kept.

5.1. Using Maude Strategies to Encode State Assertions

Given a state assertion $a \in \mathcal{A}$, we define a state filter for a, that is, a Maude strategy that checks whether a holds in a state t.

Definition 5.1. *Let $\Pi\#\varphi$ be a state assertion in $\mathcal{A}$. A* state filter *for $\Pi\#\varphi$ is a strategy $\mathsf{F}_{\Pi\#\varphi}$ defined as $\mathsf{F}_{\Pi\#\varphi} := \texttt{not}(\texttt{amatch}\ \Pi\ \texttt{s.t.}\ \mathit{not}(\varphi))$.*

Example 5.2. *Consider the assertion* `a1` *of the safety policy of Example 4.3. The corresponding state filter is*

$$\mathsf{F}_{\texttt{a1}} = \texttt{not(amatch(\{SC | V | T\} s.t.}\ \ \textit{not}\texttt{(V < 50000000))}$$

where the different meaning of operators **`not`** *and not is explained in Footnote 4.*

Roughly speaking, given $a = (\Pi \# \varphi)$ and a term t, applying the state filter F_a to t can be thought of as a binary test that either fails (delivering the empty set if a does not hold in t) or it succeeds (delivering the singleton $\{t\}$, otherwise). More formally, the following proposition holds.

Proposition 5.3. *Let* $\mathcal{R} = (\Sigma, E, R)$ *be a rewrite theory. Given a state assertion* $a \in \mathcal{A}$*, and a term* t *in* $\tau(\Sigma)$

$$[\![\mathsf{F}_a]\!](t) = \begin{cases} \emptyset & \textit{if } a \textit{ does not hold in } t \\ \{t\} & \textit{otherwise} \end{cases}$$

Proof. Immediate by Definition 3.2 (semantics of the strategy operators). □

Example 5.4. *Consider the rewrite theory* $\mathcal{R}_{\texttt{DAM}}$ *of Example 2.1 together with the state*

$$t = \texttt{\{[s1,close] [s2,close] [s3,close] | 50000010 | 0\}}$$

and the assertion `a1` *of the safety policy of Example 4.3. Clearly,* `a1` *does not hold in* t *since the water volume is greater than 50 millions* m^3*. Indeed,* $[\![\mathsf{F}_{\texttt{a1}}]\!](t) = \emptyset$ *as witnessed by the execution of the state filter* $\mathsf{F}_{\texttt{a1}}$ *on* t*:*

```
srew {[s1,close] [s2,close] [s3,close] | 50000010 | 0} using
                         not(amatch {SC | V | T} s.t. not(V < 50000000)) .

No solution.
rewrites: 3 in 0ms cpu (0ms real) (21739 rewrites/second)
```

A state filter for a state assertion can be naturally lifted to sets of state assertions by concatenating the state filters associated with each state assertion via the sequential strategy operator `;`.

Definition 5.5. *Let* $\{a_1, \ldots, a_n\}$ *be a safety policy. Then, the* state filter $\mathsf{F}_{\{a_1,\ldots,a_n\}}$ *for* $\{a_1, \ldots, a_n\}$ *is defined as* $\mathsf{F}_{\{a_1,\ldots,a_n\}} := \mathsf{F}_{a_1}\ ;\ \ldots\ ;\ \mathsf{F}_{a_n}$.

The state filter $\mathsf{F}_{\mathcal{A}}$ allows one to check whether all of the state assertions of $\mathcal{A}$ hold in a given state by sequentially testing the state filters that are associated with the assertions in $\mathcal{A}$. More formally,

Proposition 5.6. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory. Let $\mathcal{A}$ be a safety policy. Let t be a term in $\tau(\Sigma)$. Then,*

$$[\![\mathsf{F}_{\mathcal{A}}]\!](t) = \begin{cases} \emptyset & \text{if } \exists\, a \in \mathcal{A} \text{ s.t. } a \text{ does not hold in } t \\ \{t\} & \text{otherwise} \end{cases}$$

Proof. Let $\mathcal{A} = \{a_1, \ldots, a_n\}$, $n \geq 0$, be a safety policy. The proof proceeds by induction on n.

$n = 0$. $\mathcal{A}$ is the empty set. Hence, the proposition vacuously holds.

$n > 0$. We have $\mathcal{A} = \{a_1, \ldots, a_n\}$. By Definition 5.5, $\mathsf{F}_{\mathcal{A}} = \mathsf{F}_{a_1}$; ... ; F_{a_n}. Let t be a term in $\tau(\Sigma)$. We distinguish two cases: (i) all of the state assertions $a_1, \ldots, a_n$ hold in t; (ii) there exists a state assertion $a \in \{a_1, \ldots, a_n\}$ that does not hold in t

Case *i*. Since $a_1, \ldots, a_n$ hold in t, $a_1, \ldots, a_{n-1}$ hold in t. Hence, by inductive hypothesis $[\![\mathsf{F}_{\{a_1,\ldots,a_{n-1}\}}]\!](t) = \{t\}$. Also, by Proposition 5.3, $[\![\mathsf{F}_{\{a_n\}}]\!](t) = \{t\}$ because a_n holds in t. Then

$$\begin{aligned} [\![\mathsf{F}_{\mathcal{A}}]\!](t) &= \bigcup_{t' \in [\![\mathsf{F}_{\{a_1,\ldots,a_{n-1}\}}]\!](t)} [\![\mathsf{F}_{\{a_n\}}]\!](t') && \text{(by Definition 3.2)} \\ &= [\![\mathsf{F}_{\{a_n\}}]\!](t) && \text{(by inductive hypothesis)} \\ &= \{t\} && \text{(by Proposition 5.3)} \end{aligned}$$

Case *ii*. There exists $a \in \{a_1, \ldots, a_n\}$ such that a does not hold in t. If $a \in \{a_1, \ldots, a_{n-1}\}$, then by inductive hypothesis $[\![\mathsf{F}_{\{a_1,\ldots,a_{n-1}\}}]\!](t) = \emptyset$. Hence,

$$\begin{aligned} [\![\mathsf{F}_{\mathcal{A}}]\!](t) &= \bigcup_{t' \in [\![\mathsf{F}_{\{a_1,\ldots,a_{n-1}\}}]\!](t)} [\![\mathsf{F}_{\{a_n\}}]\!](t') && \text{(by Definition 3.2)} \\ &= \emptyset && \text{(by inductive hypothesis)} \end{aligned}$$

If $a = a_n$, then we have

$$
\begin{aligned}
[\![\mathsf{F}_{\mathcal{A}}]\!](t) &= \bigcup_{t' \in [\![\mathsf{F}_{\{a_1,\ldots,a_{n-1}\}}]\!](t)} [\![\mathsf{F}_{\{a_n\}}]\!](t') && \text{(by Definition 3.2)} \\
&= [\![\mathsf{F}_{\{a_n\}}]\!](t) && \text{(by inductive hypothesis)} \\
&= \emptyset && \text{(by Proposition 5.3)}
\end{aligned}
$$

□

Corollary 5.7. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory. Let $\mathcal{A}$ be a set of state assertions. Let t be a term in $\tau(\Sigma)$. Then, $[\![\mathsf{F}_{\mathcal{A}}]\!](t) = \{t\}$ iff t is safe w.r.t. $\mathcal{A}$.*

Proof. Direct consequence of Proposition 5.6 and the definition of safe state w.r.t. a set $\mathcal{A}$ of state assertions. □

5.2. Integrating State Assertions into Path Strategies

In Section 5.1, we showed how state assertions can be checked on a system state by executing the corresponding state filter on it. In other words, we have an effective methodology that allows one to establish if a given state is safe w.r.t. a given safety policy.

Given a path strategy P, we can combine state filters with P to build a *safe* version of P that drives system executions only through safe states. For this purpose, we define the following auxiliary transformation.

Definition 5.8. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and let $\mathcal{A}$ be a safety policy. Let $\mathsf{P}, \mathsf{P}_1, \mathsf{P}_2 \in PStr(\mathcal{R})$ be path strategies. We define the path transformation safe(P) as follows:*

$$
safe(\mathsf{P}) = \begin{cases}
\mathtt{l};\mathsf{F}_{\mathcal{A}} & \textit{if } \mathsf{P} := \mathtt{l},\ \textit{with } \mathtt{l} \in \textit{labels}(R) \\
\mathtt{all};\mathsf{F}_{\mathcal{A}} & \textit{if } \mathsf{P} := \mathtt{all} \\
safe(\mathsf{P}_1)^{\bullet} & \textit{if } \mathsf{P} := \mathsf{P}_1^{\bullet},\ \textit{with } \bullet \in \{*, +, !\} \\
safe(\mathsf{P}_1) \circ safe(\mathsf{P}_2) & \textit{if } \mathsf{P} := \mathsf{P}_1 \circ \mathsf{P}_2,\ \textit{with } \circ \in \{;\, , |\}
\end{cases}
$$

Roughly speaking, the transformation *safe*(P) attaches to every rule application (which can be triggered by a rule label `l` or by the operator `all`) a state filter that checks the safety of the state that results from applying the considered rewrite rule.

Definition 5.9. *Let $\mathcal{A}$ be a safety policy and let* P *be a path strategy. Then, a* path-safe *strategy w.r.t. $\mathcal{A}$ is a strategy* $\overline{\mathsf{P}}(\mathcal{A})$ *defined as* $\overline{\mathsf{P}}(\mathcal{A}) := \mathsf{F}_{\mathcal{A}}; \mathit{safe}(\mathsf{P})$.

Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, a path-safe strategy $\overline{\mathsf{P}}(\mathcal{A})$ explores the search space of $\mathcal{R}$ in the following way: given an initial state t, first the state filter $\mathsf{F}_{\mathcal{A}}$ is executed on t to check whether t is safe w.r.t. $\mathcal{A}$, and then $\mathit{safe}(\mathsf{P})$ is applied to guide the execution only through those states computed by the strategy P that are safe w.r.t. $\mathcal{A}$. This way, we guarantee that any computation that has been generated by means of the strategy $\overline{\mathsf{P}}(\mathcal{A})$ only includes safe states. Furthermore, unsafe computations are completely removed from the search space of $\mathcal{R}$.

The soundness of the proposed transformation methodology is stated in the following proposition.

Proposition 5.10. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, and let $\mathcal{A}$ be a safety policy. Let* $\overline{\mathsf{P}}(\mathcal{A})$ *be a path-safe strategy for $\mathcal{A}$. Then,*

i. *Let $t \in \tau(\Sigma)$. Every computation in $\mathcal{T}_{\mathcal{R}}^{\overline{\mathsf{P}}(\mathcal{A})}(t)$ is also a computation in $\mathcal{T}_{\mathcal{R}}(t)$;*

ii. *For every computation $t_1 \rightarrow_{R,E} t_2 \rightarrow_{R,E} \ldots \rightarrow_{R,E} t_n$ in $\mathcal{T}_{\mathcal{R}}^{\overline{\mathsf{P}}(\mathcal{A})}(t_1)$, it holds (a) the states t_i, $i = 1, \ldots, n$, are safe w.r.t. $\mathcal{A}$; and (b) $t_n \in [\![\mathsf{P}]\!](t_1)$.*

Proof. Claim i trivially holds since $\mathcal{T}_{\mathcal{R}}^{\overline{\mathsf{P}}(\mathcal{A})}(t)$ includes a subset of the computations of $\mathcal{T}_{\mathcal{R}}(t)$ by Definition 3.3. Let us prove Claim $ii.a$ and Claim $ii.b$.

($ii.a$) Let $\mathcal{C} = (t_1 \rightarrow_{R,E} t_2 \rightarrow_{R,E} \ldots \rightarrow_{R,E} t_n)$ be a computation in $\mathcal{T}_{\mathcal{R}}^{\overline{\mathsf{P}}(\mathcal{A})}(t_1)$, where $\overline{\mathsf{P}}(\mathcal{A}) = \mathsf{F}_{\mathcal{A}}; \mathit{safe}(\mathsf{P})$. Then, by Definition 3.3, we have

$$t_n \in [\![\overline{\mathsf{P}}(\mathcal{A})]\!](t_1) = [\![\mathsf{F}_{\mathcal{A}}; \mathit{safe}(\mathsf{P})]\!](t_1).$$

Now, we proceed by contradiction and we assume that there exists a state t_i in $\mathcal{C}$ such that t_i is not safe w.r.t. $\mathcal{A}$. Hence, by Proposition 5.6, $[\![\mathsf{F}_{\mathcal{A}}]\!](t_i) = \emptyset$. We distinguish two cases: t_i is the initial state t_1, and t_i is any state in $\{t_2, \ldots t_n\}$.

$t_i = t_1$**.** In this case, $[\![\mathsf{F}_{\mathcal{A}}]\!](t_1) = \emptyset$; hence, by Definition of the semantics of the sequential operator ; (see Definition 3.2)

$$[\![\overline{\mathsf{P}}(\mathcal{A})]\!](t_1) = [\![\mathsf{F}_{\mathcal{A}}; \mathit{safe}(\mathsf{P})]\!](t_1) = \emptyset$$

which leads to a contradiction since we assumed $t_n \in [\![\overline{\mathsf{P}}(\mathcal{A})]\!](t_1)$.

$t_i \in \{t_2, \ldots, t_n\}$. In this case, the unsafe state t_i has been generated by the rewrite step $t_{i-1} \rightarrow_{R,E} t_i$ which has been triggered by some rule label `l` or the operator `all` in the path strategy P. Now note that every occurrence of `l` and `all` in $\mathit{safe}(\mathsf{P})$ is followed by an application of the state filter $\mathsf{F}_{\mathcal{A}}$, that is, `l`;$\mathsf{F}_{\mathcal{A}}$ and `all`;$\mathsf{F}_{\mathcal{A}}$. Since t_i is not safe w.r.t $\mathcal{A}$, in both cases, we have

$$[\![\mathtt{l}\ ;\ \mathsf{F}_{\mathcal{A}}]\!](t_{i-1}) = \emptyset$$

$$[\![\mathtt{all}\ ;\ \mathsf{F}_{\mathcal{A}}]\!](t_{i-1}) = \emptyset$$

Hence, t_i is not reachable from t_{i-1} using `l` ; $\mathsf{F}_{\mathcal{A}}$ or `all` ; $\mathsf{F}_{\mathcal{A}}$, which leads to a contradiction because we assumed $t_{i-1} \rightarrow_{R,E} t_i$.

(ii.b) Let $\mathcal{C} = (t_1 \rightarrow_{R,E} t_2 \rightarrow_{R,E} \ldots \rightarrow_{R,E} t_n)$ be a computation in $\mathcal{T}_{\mathcal{R}}^{\overline{\mathsf{P}}(\mathcal{A})}(t_1)$. Thus, $t_n \in [\![\overline{\mathsf{P}}(\mathcal{A})]\!](t_1)$.

By Claim *ii.a*, we know that each t_i in $\mathcal{C}$ is safe w.r.t $\mathcal{A}$. Hence, by Proposition 5.6, $[\![\mathsf{F}_{\mathcal{A}}]\!](t_i) = \{t_i\}$, for $i = 1, \ldots, n$ which directly implies that

$$[\![\mathtt{l}\ ;\ \mathsf{F}_{\mathcal{A}}]\!](t_i) = [\![\mathtt{l}]\!](t_i) \quad (1)$$
$$[\![\mathtt{all}\ ;\ \mathsf{F}_{\mathcal{A}}]\!](t_i) = [\![\mathtt{all}]\!](t_i) \quad (2)$$

By (1) and (2) and Definition 5.8, it is straightforward to show that $\mathit{safe}(\mathsf{P}) = \mathsf{P}$ (trivial structural induction on P). Finally,

$$\begin{aligned} t_n \in [\![\overline{\mathsf{P}}(\mathcal{A})]\!](t_1) &= [\![\mathsf{F}_{\mathcal{A}}\ ;\ \mathit{safe}(\mathsf{P})]\!](t_1) && \text{(by Definition 5.9)} \\ &= [\![\mathit{safe}(\mathsf{P})]\!](t_1) && \text{(by } [\![\mathsf{F}_{\mathcal{A}}]\!](t_1) = \{t_1\}) \\ &= [\![\mathsf{P}]\!](t_1) && \text{(by } \mathit{safe}(\mathsf{P}) = \mathsf{P}) \end{aligned}$$

□

Example 5.11. *Consider again the Maude program $\mathcal{R}_{\mathtt{DAM}}$ of Example 2.1, the path strategy* `alt` *of Example 3.4, and the safety policy $\mathcal{A}_{\mathtt{DAM}}$ of Example 4.3. Then, the path-safe strategy* `safe-alt` *is obtained by applying the transformation $\overline{\mathsf{alt}}(\mathcal{A}_{\mathtt{DAM}})$ to* `alt`*:*

$$\mathtt{safe\text{-}alt} = \overline{\mathsf{alt}}(\mathcal{A}_{\mathtt{DAM}}) = \mathsf{F}_{\mathcal{A}_{\mathtt{DAM}}} ; \mathit{safe}(\mathtt{alt}) = \mathsf{F}_{\mathcal{A}_{\mathtt{DAM}}} ; (\overline{(\mathtt{all\text{-}(volume)}}; \mathsf{F}_{\mathcal{A}_{\mathtt{DAM}}} ; \mathtt{volume} ; \mathsf{F}_{\mathcal{A}_{\mathtt{DAM}}})*$$

Note that, in this specific case, we can even optimize `safe-alt` *by replacing* $\overline{\texttt{all-(volume)}}$ *with* `all-(`$\overline{\texttt{volume}}$`)`*, thereby producing a strategy that contains less state filters. Indeed,* $\overline{\texttt{all-(volume)}}$ *executes the state filter* $\mathsf{F}_{\mathcal{A}_{\mathtt{DAM}}}$ *after each application of an aperture command rule, which is useless since such rules do not change the water volume. Therefore,* $\mathsf{F}_{\mathcal{A}_{\mathtt{DAM}}}$ *can be safely executed only once after applying the* `volume` *rule. Simple optimizations of this kind are automatically carried out by the* STRASS *system. Now consider the initial state* t *of Example 3.6, i.e.,* `{[s1,close] [s2,close] [s3,close] | 49989000| 0}`. *By executing*

```
srew [2] [s1,close] [s2,close] [s3,close] | 49989000 | 0  using safe-alt .
```

we explore the first two states that are reachable from t *using the* safe-alt *strategy:*

```
Solution 1
rewrites: 88 in 0ms cpu (0ms real) (265060 rewrites/second)
result State: {[s1,open3] [s2,open1] [s3,close] | 49999000 | 5}
Solution 2
rewrites: 88 in 0ms cpu (0ms real) (244444 rewrites/second)
result State: {[s1,open3] [s2,close] [s3,open1] | 49999000 | 5}
```

Note that the two solutions computed by `safe-alt` *reach safe states where the water volume is lower than 50 million* m^3*. In contrast, the solutions yielded by the unrestricted path strategy* `alt` *can reach such a critical threshold, thus violating the state assertion* `a1` *of* $\mathcal{A}_{\mathtt{DAM}}$ *as shown in Example 3.6.*

6. The STRASS System

The safety enforcement methodology defined in this paper has been efficiently implemented in a Maude tool called STRASS (*STRategy-based Automatic Safety aSsurance tool*). The tool consists of three modules: a web client, a RESTful web service, and the *core* module that contains the domain logic that is responsible for running the proposed safety enforcement technique. The core has been implemented in Maude itself by using Maude's meta-level capabilities. The RESTful Web service has been written in Java. The web client is provided with an intuitive graphical user interface and is based on AJAX. It has been developed using the programming language Svelte, which gets statically compiled to HTML, CSS, and JavaScript. The implementation consists of about 800 lines of Java source code, 1200 lines

of Maude[5] code, and 2000 lines of combined HTML, CSS, TypeScript and Svelte.

The STRASS tool is publicly available together with a number of examples at `http://safe-tools.dsic.upv.es/strass`. Also a starting guide that contains a complete description of all of the settings and detailed examples of tool usage can be found at: `http://safe-tools.dsic.upv.es/strass/download/StartingGuide.pdf`.

6.1. Implementation of STRASS

Given a safety policy $\mathcal{A}$ to be enforced on a Maude program $\mathcal{R} = (\Sigma, E, R)$, the core logic of STRASS is implemented in five phases that are executed in order: (i) the meta-level ascent phase, where the input program $\mathcal{R}$ is transformed into rich meta-level data structures using Maude built-in functions; (ii) the static analysis phase, where the (ascended) input program is analyzed to extract some relevant sorting information that is described below; (iii) the policy parsing phase, where the safety policy $\mathcal{A}$ is checked using a dedicated parser); (iv) the generation phase, where the synthesized strategy module is generated at the meta-level; and, lastly, (v) the meta-level descent phase, where the generated constructs are converted into a string of pretty-printed, executable source code.

Phases (i), (ii), and (iv) are implemented in the standard way using Maude's meta-level capabilities. With regard to phase (iii), STRASS implements a dedicated parser that takes as input: (a) the signature Σ of the input program, (b) the rule labels of the input program, (c) the provided auxiliary predicates, and (d) the set of auxiliary, internal definitions that are generated automatically during phase (ii). Different transient parsing contexts are generated that can be used depending on the particular expression to be parsed. State assertions and path strategies are analyzed in phase (iii). We first check that any formula φ in a state assertion $\Pi \# \varphi$ only contains operators of $\mathcal{R}$ and user-defined auxiliary predicates, while the pattern component Π is a constructor term of $\mathcal{R}$. Next, we check that any label that appears in a path strategy belongs to the rule labels of $\mathcal{R}$. This parser also leverages Maude meta-level features to support native Maude features such as mixfix operators and user-defined operator precedence. The parser has been endowed with the capability to track line numbers, allowing for

[5]We use a developer version of Maude implemented in C++ called *Mau-Dev* [9]

accurate error messages.

As for phase (v), it creates a new strategy module that contains all of the automatically generated definitions so that the original program modules get untouched. In Maude, named strategy definitions are encoded using two different constructs[6], one using the keyword `strat` to declare the strategy name alongside its type signature, and the other using `sd` to assign a concrete strategy expression to said name [7, Section 10.2]. State assertions $\Pi \# \varphi$ in $\mathcal{A}$ are encoded as state filter strategies $\mathsf{F}_{\Pi\#\varphi}$ which are then wrapped in named strategy definitions as follows:

$$\texttt{strat s}i \; \texttt{:} \;\; \texttt{@} \; ls(\Pi) \;\; \texttt{.}$$
$$\texttt{sd s}i \; \texttt{:=} \; \mathsf{F}_{\Pi\#\varphi} \;\; \texttt{.}$$

where i specifies an automatically generated index that ensures that the strategy name is unique. The sort that appears after the `@` symbol (or subject sort) indicates the sort of terms to which the named strategy applies. In this case, the least sort of the pattern component of the state strategy, $ls(\Pi)$, is used to satisfy the type signature.

Since the synthesized strategies should be applicable to input terms of any sort at run time, strategies are specialized w.r.t. sorts (i.e., monomorphized) into a series of most specific strategy definitions at transformation time. Generally, monomorphization is an automatic process that is conceptually contrary to generalization, in which polymorphic constructs of some language (usually functions) are replaced by many monomorphic, specialized instances until the result is determinate enough for execution purposes. The purpose of the monomorphization transformation is twofold: it allows a legal strategy module to be generated that satisfies the Maude type checker; and it additionally leverages the sort hierarchy to enable useful optimizations.

In our implementation, each state filter $\mathsf{F}_{\mathcal{A}}$ is translated into the set $\{\mathsf{F}^s_{\mathcal{A}} \mid s \text{ is a maximal sort in } (S, <) \text{ of } \mathcal{R}\}$. Each monomorphized definition $\mathsf{F}^s_{\mathcal{A}}$ is assigned the name "s`-state`". For instance, the strategy named `System-state` checks all of the state assertions for terms of sort `System` (or a subsort of it). The resulting source code follows this scheme:

$$\texttt{strat } s_1\texttt{-state :} \;\; \texttt{@} \; s_1 \;\; \texttt{.}$$

[6]These strategy constructs can be considered analogous to the standard `op` and `eq` definitions, respectively.

```
   sd s1-state := F_A^s1 .
                  ...
strat sn-state :  @ sn .
   sd sn-state := F_A^sn .
```

$$
\begin{array}{r}
\texttt{sd}\ s_1\texttt{-state}\ \texttt{:=}\ F_{\mathcal{A}}^{s_1}\ \texttt{.} \\
\dots \\
\texttt{strat}\ s_n\texttt{-state}\ \texttt{:}\ \ \texttt{@}\ s_n\ \texttt{.} \\
\texttt{sd}\ s_n\texttt{-state}\ \texttt{:=}\ F_{\mathcal{A}}^{s_n}\ \texttt{.}
\end{array}
$$

where $s_1, \ldots, s_n$ are maximal sorts in $\mathcal{R}$. An analogous monomorphization process is also applied to path-safe strategies, which are similarly assigned the name "s-`path`".

For the sake of the user's convenience, given the path strategy `P`, STRASS also computes the path-safe strategies corresponding to the transitive and reflexive closure of `P` (i.e., $\texttt{P}^*$), the transitive closure of `P` (i.e, $\texttt{P}^+$), and the normalizing strategy `P!`. Hence, the user only has to provide `P` to get a safe program w.r.t. `P`, $\texttt{P}^*$, $\texttt{P}^+$, `P!`, and the safety policy $\mathcal{A}$. This effectively enables different run-time techniques for the exploration of the safe computation space without having to execute the correction technique more than once. The strategies $\texttt{P}^*$, $\texttt{P}^+$, `P!` are also monomorphized and are respectively assigned the names "s`*`", "s`+`", and "s`!`", for each one of the corresponding sorts s in $\mathcal{R}$.

Let us finally describe two useful optimizations of our framework that are implemented by STRASS. The first one is an *inlining* transformation that expands strategy calls and then simplifies the resulting strategy expressions. For example, consider the strategy definitions:

```
sd s-state := idle .
sd s• := s-state ; ((all ; s-state) •) .
```

First, both calls to s-`state` are expanded to `idle`

```
sd s-state := idle .
sd s• := idle ; ((all ; idle) •) .
```

Then, since `idle` is the identity element of the strategy constructor `_ ; _`, by applying (right and left) identity axioms, `idle` is dropped

```
sd s-state := idle .
sd s• := all • .
```

We then say the strategy s-`state` has been *inlined into* the strategy $s\bullet$, where $\bullet \in \{\texttt{*}, \texttt{+}, \texttt{!}\}$.

The second optimization, called *sort-dependence filter erasure*, removes the calls to state filters $\mathsf{F}_{\Pi\#\varphi}$ in $\mathsf{F}^s_{\mathcal{A}}$ when it is possible to statically ensure that the state assertion $\Pi\#\varphi$ vacuously holds because Π does not match any possible subterms of a term of sort s. The following notion of $s$ort graph allows us to better describe this optimization. Roughly speaking, the sort graph models the idea that terms of a certain sort can be "constructed upon" subterms of different sorts. More formally, given a sort s of the signature Σ, consider the set of sorts of all possible subterms of any term of sort s, according to Σ. Given the signature Σ, the sort graph of Σ can be decided as a reachability problem in the rewrite theory $SortGraph(\Sigma) = (\Sigma \cup \{\top\}, \emptyset, R)$, where the rules of R are simply obtained by orienting (right to left) each operator declaration $f : s_1, \ldots s_n \to s$ in Σ as the set of rewrite rules $s : \top \Rightarrow s_i : \top$, with i in $1, \ldots, n$, where $\top$ conceptually represents a universal supersort of all sorts in Σ. That is, $R = \{s : \top \Rightarrow s_i : \top,\ \text{with } i \in 1, \ldots, n \mid f : s_1, \ldots s_n \to s \in \Sigma\}$; hence, for any sort s of Σ, the normal forms of s in the rewrite theory $SortGraph(\Sigma)$ deliver all possible sorts of the subterms of any term of sort s.

For instance, if the sort `System` does not simplify to `Nat` in the rewrite theory $SortGraph(\Sigma)$ (i.e., no term of sort `System` can contain a natural number as subterm), then $\mathsf{F}^{\mathtt{System}}_{\mathcal{A}}$ may safely omit the state strategies that are only relevant to natural numbers. For the purposes of the implementation, one of the specific steps performed by the static analysis phase (ii) is computing and encoding the sort graph of Σ as a data structure called the sort dependency map (SDM), which associates each sort s of Σ with the set of its reachable sorts in the rewrite theory $SortGraph(\Sigma)$ as follows:

$$SDM(S) = \{s' \mid s \Rightarrow^+ s' \text{ in } SortGraph(\Sigma)\}$$

where $\Rightarrow^+$ denotes the transitive closure of the rewriting relation $\Rightarrow$.

Finally, since the generated strategies may need to invoke the auxiliary predicates in φ at run time, the auxiliary predicates provided by the user are directly inserted into the synthesized strategy module, effectively exposing them to all of the generated strategies but not to the original program.

6.2. Features

The key points of the STRASS tool are as follows:

- Efficient and easy to use: a program may be fixed with only three steps using a friendly assistant-style user interface (Steps 1–3 of Figure 4).

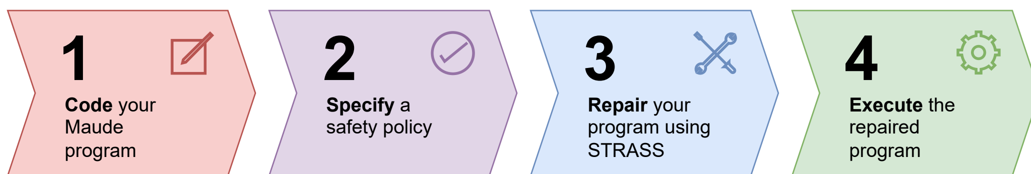

Figure 4: Graphical outline of the STRASS usage process

- Fast and performant: most programs can be fixed in milliseconds. Program transformation time is often negligible.
- Support for an extremely wide variety of complex, multi-module Maude programs making use of advanced language features.
- A set of representative examples can be selected from a drop-down list.
- Built-in editor featuring advanced text editing capabilities such as multiple cursor editing and code unfolding.
- Complete syntax highlighting for both Maude and our domain-specific policy specification language.
- Detailed error messages with in-editor hints and markings.
- Automatic, transparent optimizations are applied to simplify the generated strategy module.
- Ability to upload Maude source code files from the local storage.
- All artifacts and a set of representative benchmarks are publicly available at the STRASS website for download.

6.3. Experimental Results

To experimentally evaluate the performance and the outcomes of the STRASS system, we coupled several Maude programs with safety policies, and we used STRASS to generate the corresponding safe versions. We benchmarked STRASS on the following collection of Maude programs, which are all available and fully described within the STRASS web platform: (i) *Blocks World*, a Maude encoding of the classical AI planning problem that consists of setting one or more vertical stacks of blocks on a table using a robotic arm; (ii) *Containers*, a Maude specification that models the cargo manipulation

in a container terminal; (iii) *Dam Controller*, the controlling system used as our leading example; (iv) *Maze*, a non-deterministic maze exploration algorithm; (v) *Space invaders*, a Maude specification of a classic videogame of the 70's; and (vi) *Satellite Controller*, a realistic model of the behavior of an unmanned space probe which can rotate on itself, extract scientific information from points of interest and communicate with mission control under certain visibility conditions, where durations of the different activities are measured and coordinated using a series of clocks.

The primary objective of this experimental evaluation aims at comparing the performance of the computed safe programs w.r.t. the original unsafe counterparts both in terms of space and time. More precisely, we have considered four assertions per benchmark. We have generated three computation trees of increasing depths (in the range [5, 50]) for each input program and its corresponding strategy-driven version computed by STRASS. We have then compared the size of the computation trees and the time needed to produce them. Note that, for the original programs, the resulting search space may contain unsafe states, while the transformed version only contains safe states by construction.

All of the experiments were conducted on an Intel Xeon Silver 4215R 3.3GHz CPU with 378GB of RAM. Table 1 summarizes our results. For each benchmark, we considered the original program $\mathcal{R}$ and its safe version $\mathcal{R}_S$ computed by STRASS. The generation time of $\mathcal{R}_S$ for each considered benchmark is negligible (less than 0.1 milliseconds in all cases). Column *Depth* sets a bound to the depth of the computation trees that are generated for $\mathcal{R}$ and $\mathcal{R}_S$, while $Size_{\mathcal{R}}$ and $Size_{\mathcal{R}_S}$ respectively measure the sizes of the search spaces for $\mathcal{R}$ and $\mathcal{R}_S$ as the number of states in their corresponding computation trees. Execution times for the generation of the computation trees of depth n are respectively shown in Columns $T_{\mathcal{R}}$ and $T_{\mathcal{R}_S}$ and are measured in milliseconds. We set a timeout of 60 minutes for the generation of the computation trees that is only overrun in the case of *Maze*. We recorded the total speedup $T_{\mathcal{R}}/T_{\mathcal{R}_S}$ in Column *Speedup*. It is worth noting that we chose to measure the complexity of our experiments in terms of the size of the generated program search space instead of using traditional program size indicators such as lines of code or number of equations and rewrite rules. This is because program size is not a meaningful indicator in our context due to the highly, purely declarative nature of Maude, which allows one to formalize extremely complex systems in a highly compact and concise way. For instance, the satellite controller implements a detailed unmanned space

$\mathcal{R}$	*Depth*	$Size_{\mathcal{R}}$	$Size_{\mathcal{R}_S}$	$T_{\mathcal{R}}$	$T_{\mathcal{R}_S}$	*Speedup*
Blocks World	30	511,921	1,205	9,668	40	241.70
	40	2,004,332	2,568	41,384	87	475.68
	50	5,841,540	4,689	139,198	171	814.02
Containers	15	350,391	53,624	40,967	7,635	5.37
	20	1,465,829	88,097	166,614	13,289	12.54
	25	4,172,116	122,538	549,430	19,518	28.15
Dam Controller	15	139,948	220	4,198	14	299.86
	30	2,271,930	505	82,845	30	2,761.50
	45	9,581,406	790	392,157	48	8,169.94
Maze	4	196	23	8	2	4
	6	133,225	78	59,553	9	6,617
	8	>3,154,238	303	>1,236,722	53	>23,334.38
Space Invaders	15	518,379	88,680	21,679	4,985	4.35
	20	1,797,799	268,115	120,930	19,609	6.17
	25	5,024,516	720,649	515,246	65,345	7.89
Satellite	40	648,965	8,585	35,776	1,508	23.72
	45	1,687,605	9,924	216,781	3,097	70.00
	50	3,496,645	11,289	894,733	6,308	141.84

Table 1: Experimental results of the safe enforcement technique.

probe orbiting Earth in just 23 equations and 16 rewrite rules.

Our figures show impressive results. For all of the conducted experiments, the proposed transformation achieves a dramatic reduction of the original search space and its corresponding generation time by actively pruning unsafe states from the computation tree of the input programs. On average, the pruned search space is 2137 times smaller than the original one and is computed 2364 times faster. One of the worst speedups is achieved by *Space invaders* with depth 15, yet we narrow the original search space from 518,379 states to 88,680 states, thereby obtaining a 5.84x smaller computation tree that is generated 4.35 times faster. The best performance is achieved by executing STRASS on *Maze*. In this case, the generation of a computation tree of depth 8 times out after 60 minutes delivering an incomplete search space of 3,154,238 states. In contrast, the safe program yielded by STRASS generates a complete tree of depth 8 that includes 303 safe states in just 53 ms.

STRASS supersedes and improves ATAME, the preliminary safety enforce-

ment tool based on our previous methodology defined in [4]. From a conceptual viewpoint, ATAME resorts to a bold program transformation approach that typically generates overly complex and textually large conditional program specifications, whereas STRASS neatly produces a strategy module that supplements the original program without changing its code, thereby avoiding the risk of jeopardizing the key executability properties of the code that are required for its formal analysis or verification. Also, STRASS works on a larger class of Maude programs as it can handle non-topmost rewrite theories as well as rewrite expressions in rule conditions, while ATAME cannot. Actually, ATAME cannot be applied with correctness guarantees to *Maze*, *Blocks world*, and *Space invaders* specifications since they contain either rewrite expressions or they are not topmost. In fact, ATAME can only transform *Dam controller* and *Containers* and its performance for these benchmarks is comparable to the one of STRASS, with the speedup being greater than one for *Dam controller* and less than one for *Containers*, as shown in Figure 5.

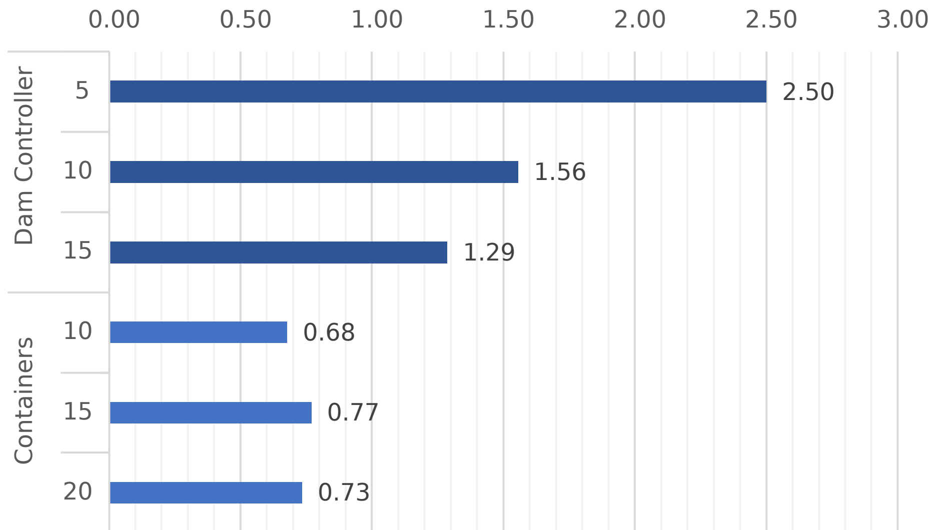

Figure 5: Speedup of STRASS w.r.t. ATAME for programs *Dam controller* and *Container*

Finally, we note that, albeit programs fixed with ATAME may exhibit a higher relative speedup for the case when the safety policy agressively prunes a high quantity of states, STRASS consistently shows less overall overhead. Actually, when we impose a vacuous safety policy (that is trivially satisfied

and never deems any state or transition as unsafe) on our benchmark programs, the overall overhead of ATAME is 1.02 whereas the overall overhead incurred by STRASS is only 0.16 across our benchmark set. We use *overhead* to denote the relative slowdown that can be attributed to the safety enforcement technique, which is calculated as $(T_{\mathcal{R}_S} - T_{\mathcal{R}})/T_{\mathcal{R}}$. Note that an overhead of zero indicates that the technique has not caused any additional execution costs.

7. STRASS to the Rescue: a Typical Safety Enforcement Session

Let us illustrate how STRASS works in practice by showing a typical safety-enforcement session for our dam controller specification of Example 2.1.

Maude programs can be uploaded in STRASS as simple `.maude` module files, written from scratch inside a dedicated edit area, or they can be selected from a preloaded collection of Maude programs that is provided with the tool for demonstration purposes. In this case, to start the tool session, we select *Dam Controller* from the preloaded example programs (see Figure 6), which encodes the dam controller $\mathcal{R}_{\texttt{DAM}}$ of Example 2.1 via the Maude module `DAM-CTRL`.

The next phase allows the user to specify the safety properties to be enforced on the input program. These properties include a path strategy `P` and a safety policy $\mathcal{A}$ that contains one or more state assertions. Recall that assertions may use logic predicates and functions that are already defined in the program or new ones that can be specified at this stage.

In this session, we input the path strategy `(all-(volume) ; volume)` and the assertions that specify the safety policy $\mathcal{A}_{\texttt{DAM}}$ of Example 4.3 together with the additional function `openSpillways`, which is used in the formalization of $\mathcal{A}_{\texttt{DAM}}$ itself (see Figure 7).

At this point, by pressing the button NEXT, STRASS automatically generates the strategy module of Figure 8 ,which encodes the path-safe strategies for the given input. Note that some strategy definitions (e.g., `Spillways` and its accompanying search strategy definitions `Spillways*`, `Spillways+`, and `Spillways!`) have been simplified by the *inlining* optimization to trivial expressions such as `idle` and `(all *)` because no state assertion or path strategy for sort `Spillways` exists in the safety policy $\mathcal{A}_{\texttt{DAM}}$.

The final outcome integrates the strategy module `DAM-CTRL-SAFE` into the original module `DAM-CTRL` (see Figure 9) yielding a safe program w.r.t.

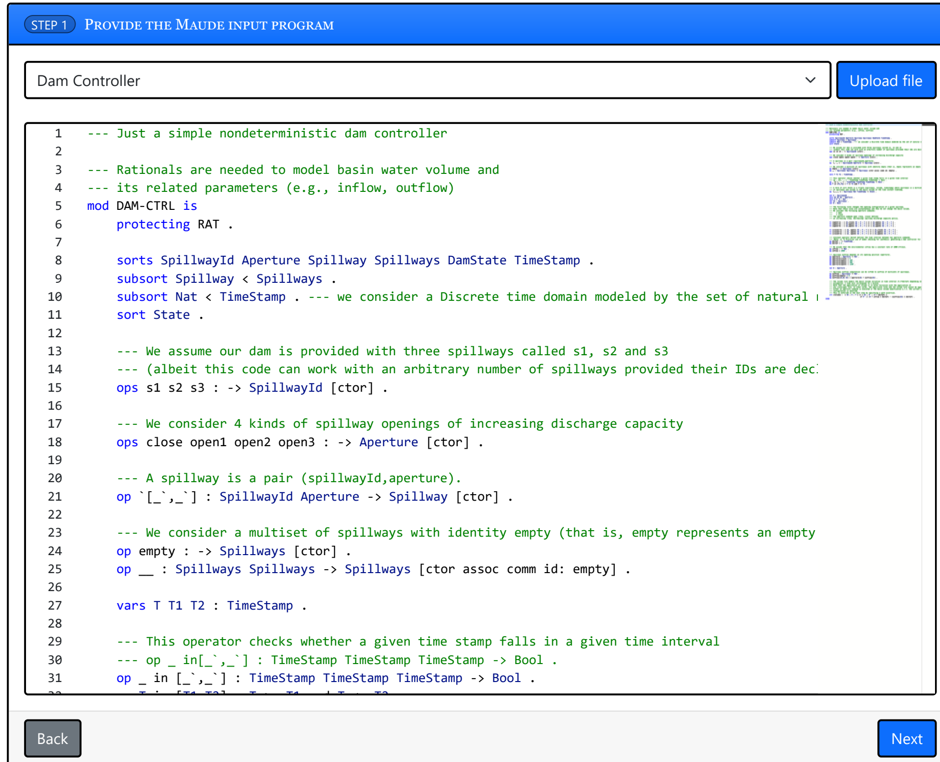

Figure 6: Loading the `DAM-CTRL` Maude module in STRASS.

$\mathcal{A}_{\texttt{DAM}}$, in which we can reproduce the safe behavior shown in Example 5.11.

8. Related work and Conclusion

The design of safety-critical and dependable systems is becoming increasingly important. We have introduced an automated transformation technique that supports the efficient enforcement of customized, strong invariant properties that are given apart from the system code in a purely declarative way. In contrast to our previous work [4], the new methodology in this paper does not modify the original rules or equations but simply enforces the assertions $\mathcal{A}$ by superimposing a control module that is automatically generated from $\mathcal{A}$. By relying on Maude's strategy language, the initial separation of concerns is preserved by the transformation, leading to corrected programs that are easier to understand and maintain.

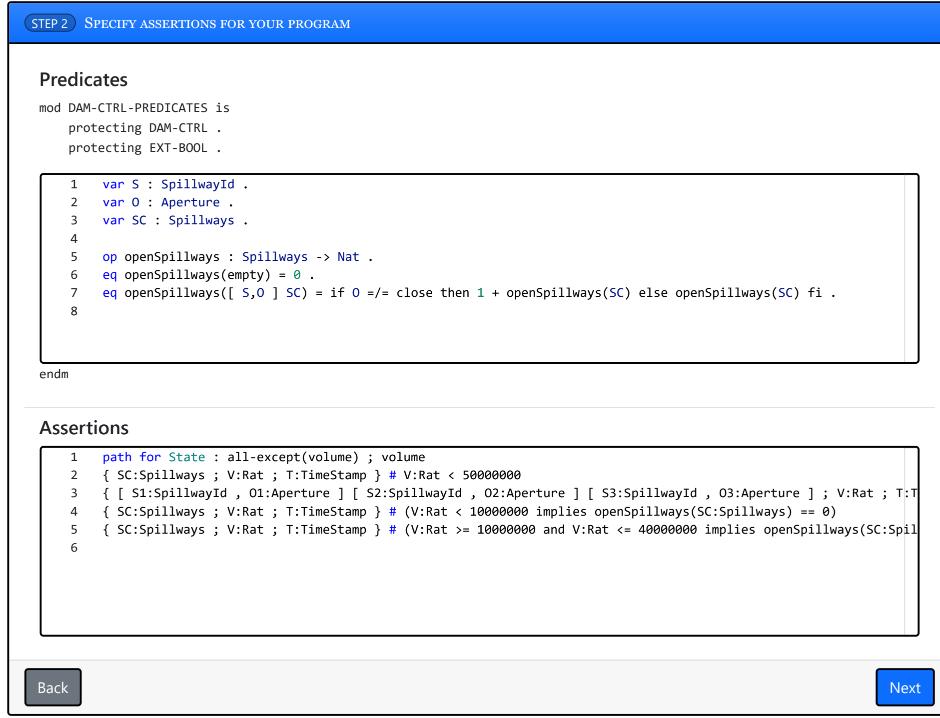

Figure 7: Loading the safety policy $\mathcal{A}_{\mathtt{DAM}}$ in STRASS.

There are also many works in the literature about the automatic enforcement of specific safety properties, either statically or dynamically, with the most traditional approaches being based on applying run-time checks (see [3, 4] and references therein). Out of all of the related approaches, the methodology presented in [2] is the closest to our work since it defines a generic strategy to impose state invariants on Maude programs that can be expressed in different logics. However, this is achieved by imposing (on top of Maude) a programmed, ad-hoc strategy that dynamically drives the system's execution in such a way that some state transitions are avoided so that every system state complies with the constraints. In contrast, our methodology is static and enforces the system assertions by transforming the program code in such a way that the imposed constraints are verified by construction without resorting to any ad-hoc strategies.

The framework for assertion-based debugging of constraint logic programs of [10] defines a program transformation that can be used for checking at runtime those assertions that cannot be decided at compile time. Similarly

```
smod DAM-CTRL-SAFE is
    protecting DAM-CTRL .
    protecting EXT-BOOL .
    op openSpillways : Spillways -> Nat .
    eq openSpillways(empty) = 0 .
    eq openSpillways(SC:Spillways[S:SpillwayId, O:Aperture]) =
        if O:Aperture =/= close then 1 + openSpillways(SC:Spillways) else openSpillways(SC:Spillways) fi .
    strat Aperture! : @ Aperture .
    strat Aperture* : @ Aperture .
    strat Aperture+ : @ Aperture .
    strat Aperture-state : @ Aperture .
    strat DamState! : @ DamState .
    strat DamState* : @ DamState .
    strat DamState+ : @ DamState .
    strat DamState-state : @ DamState .
    strat SpillwayId! : @ SpillwayId .
    strat SpillwayId* : @ SpillwayId .
    strat SpillwayId+ : @ SpillwayId .
    strat SpillwayId-state : @ SpillwayId .
    strat Spillways! : @ Spillways .
    strat Spillways* : @ Spillways .
    strat Spillways+ : @ Spillways .
    strat Spillways-state : @ Spillways .
    strat State! : @ State .
    strat State* : @ State .
    strat State+ : @ State .
    strat State-path : @ State .
    strat State-state : @ State .
    strat TimeStamp! : @ TimeStamp .
    strat TimeStamp* : @ TimeStamp .
    strat TimeStamp+ : @ TimeStamp .
    strat TimeStamp-state : @ TimeStamp .
    strat s2 : @ State .
    strat s3 : @ State .
    strat s4 : @ State .
    strat s5 : @ State .
    sd Aperture! := (all) ! .
    sd Aperture* := (all) * .
    sd Aperture+ := (all) + .
    sd Aperture-state := idle .
    sd DamState! := (all) ! .
    sd DamState* := (all) * .
    sd DamState+ := (all) + .
    sd DamState-state := idle .
    sd SpillwayId! := (all) ! .
    sd SpillwayId* := (all) * .
    sd SpillwayId+ := (all) + .
    sd SpillwayId-state := idle .
    sd Spillways! := (all) ! .
    sd Spillways* := (all) * .
    sd Spillways+ := (all) + .
    sd Spillways-state := idle .
    sd State! := (State-state) ; ((State-path) ; (((all) ; (State-path)) !)) .
    sd State* := (State-state) ; ((State-path) ; (((all) ; (State-path)) *)) .
    sd State+ := (State-state) ; ((State-path) ; (((all) ; (State-path)) +)) .
    sd State-path := (((close1-C) ; (State-state)) | ((close2-1) ; (State-state)) | ((close3-2) ; (State-state)) |
                     ((open1-2) ; (State-state)) | ((open2-3) ; (State-state)) |
                     (openC-1) ; (State-state)) ; ((volume) ; (State-state)) .
    sd State-state := (s2) ; ((s3) ; ((s4) ; (s5))) .
    sd TimeStamp! := (all) ! .
    sd TimeStamp* := (all) * .
    sd TimeStamp+ := (all) + .
    sd TimeStamp-state := idle .
    sd s2 := not(amatch {SC:Spillways ; V:Rat ; T:TimeStamp}
        s.t. V:Rat < 50000000 = false) .
    sd s3 := not(amatch {[S1:SpillwayId, O1:Aperture][S2:SpillwayId, O2:Aperture][S3:SpillwayId, O3:Aperture] ;
            V:Rat ; T:TimeStamp}
        s.t. V:Rat > 40000000 implies O1:Aperture == open3 and O2:Aperture =/= close and
            O3:Aperture =/= close or O2:Aperture == open3 and O1:Aperture =/= close and
            O3:Aperture =/= close or O3:Aperture == open3 and O1:Aperture =/= close and
            O2:Aperture =/= close = false) .
    sd s4 := not(amatch {SC:Spillways ; V:Rat ; T:TimeStamp}
        s.t. V:Rat < 10000000 implies openSpillways(SC:Spillways) == 0 = false) .
    sd s5 := not(amatch {SC:Spillways ; V:Rat ; T:TimeStamp}
        s.t. V:Rat >= 10000000 and V:Rat <= 40000000 implies openSpillways(SC:Spillways) == 2 = false) .
endsm
```

Figure 8: The strategy module DAM-CTRL-SAFE for DAM-CTRL.

Fixed program

```
88      --- configuration is observed.
89      --- This is enforced in the next step by specifying a path assertion.
90      crl [volume] :  { SC ; V ; T } => { SC ; V' ; (T + deltaT) }
91                                      if V' := (V + inflow * deltaT) - (outflow(SC) * deltaT) .
92  endm
93
94  ***(
95      Generated by STRASS -- safe-tools.dsic.upv.es/strass
96  )
97
98  smod DAM-CTRL-SAFE is
99      protecting DAM-CTRL .
100     protecting EXT-BOOL .
101     op openSpillways : Spillways -> Nat .
102     eq openSpillways(empty) = 0 .
103     eq openSpillways(SC:Spillways[S:SpillwayId, O:Aperture]) = if O:Aperture =/= close then 1 + ope
104     strat Aperture! : @ Aperture .
105     strat Aperture* : @ Aperture .
106     strat Aperture+ : @ Aperture .
107     strat Aperture-state : @ Aperture .
108     strat DamState! : @ DamState .
109     strat DamState* : @ DamState .
110     strat DamState+ : @ DamState .
111     strat DamState-state : @ DamState .
112     strat SpillwayId! : @ SpillwayId .
113     strat SpillwayId* : @ SpillwayId .
114     strat SpillwayId+ : @ SpillwayId .
115     strat SpillwayId-state : @ SpillwayId .
116     strat Spillways! : @ Spillways .
117     strat Spillways* : @ Spillways .
118     strat Spillways+ : @ Spillways .
```

Back

Figure 9: The resulting Maude module computed by STRASS.

to our work, any meta-interpretation level is eliminated since the process of assertion checking is compiled into a transformed program which checks the assertions while running on a standard (CLP) execution system. However, the transformation of [10] does not apply to the complex rewrite theories that we consider in this work, which support inductively nested structures that may obey structural axioms such as associativity, commutativity, and unity [4].

Liquid Haskell (LH) [11] allows Haskell code to be annotated with data type invariants that complement the invariants imposed by the types with logical predicates; this allows safety properties to be enforced at compile time. A liquid type has the form $\{v : \tau | e\}$, where τ is a Hindley-Milner type and e is a Boolean expression and represents all the values u of type τ such that the expression $e[u/v]$ evaluates to true. Liquid type annotations are provided by the programmer in the input file as Haskell comments that are

ignored by GHC but are processed by LH instead. The first phase of LH uses the Haskell compiler GHC to resolve the external references, to type-check the program in the Hindley-Milner sense, and to transform it to its internal core representation. As a result, a set of type constraints is generated in the second phase, which are solved in a third phase with the help of a SMT solver. In contrast to [11], which defines constraints at the type-level, our approach specifies assertions at a specification level and uses them to statically direct a program specialization technique that produces a safe version of the input program. Then, safety checks are dynamically performed over the specialized program in the standard Maude runtime environment without resorting to external artifacts.

Also, loosely related to this work is the concept of program specialization of terminating programs based on output constraints (i.e., program post-conditions) [12]. This methodology translates the output constraints into a characterization function for the program's input that is used to guide a partial evaluation process. In contrast, we deal with non-terminating concurrent programs, and the specialization that we achieve cannot be produced by any (conventional or unconventional) partial evaluation technique for Maude programs [13]. Our technique also presents similarities with automated program correction and related problems such as code fixing and repair techniques [14] since it allows a program with incomplete specifications (given by system assertions) to be automatically fixed while keeping the transformed program as close as possible to the original one.

The proposed technique has been implemented in the prototype tool STRASS, which can be very useful for a programmer who wants to correct a program w.r.t. a preliminary version that was written with no safety concerns. To our knowledge, the assertion-based functionality for molding programs supported by STRASS is beyond the capabilities of all existing Maude tools.

References

[1] R. Rubio, N. Martí-Oliet, I. Pita, A. Verdejo, Strategies, Model checking and Branching-time Properties in Maude, J. Log. Algebraic Methods Program. 123 (2021) 100700.

[2] M. Roldán, F. Durán, A. Vallecillo, Invariant-driven Specifications in Maude, Science of Computer Programming 74 (10) (2009) 812–835.

[3] M. Alpuente, D. Ballis, J. Sapiña, Imposing assertions in Maude via program transformation, MethodsX 6 (2019) 2577–2583. doi:https://doi.org/10.1016/j.mex.2019.10.035.

[4] M. Alpuente, D. Ballis, J. Sapiña, Static Correction of Maude Programs with Assertions, Journal of Systems and Software 153 (2019) 64–85.

[5] TeReSe, Term Rewriting Systems, Cambridge University Press, 2003.

[6] J. Meseguer, Conditional Rewriting Logic as a Unified Model of Concurrency, Theoretical Computer Science 96 (1) (1992) 73–155.

[7] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, C. Talcott, Maude Manual (Version 3.2.1), Tech. rep., SRI International Computer Science Laboratory, available at: https://maude.lcc.uma.es/maude321-manual-html/maude-manual.html (2022).

[8] J. Jiang, L. Ren, Y. Xiong, L. Zhang, Inferring Program Transformations From Singular Examples via Big Code, in: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, IEEE, 2019, pp. 255–266.

[9] The Mau-Dev Website, Available at: http://safe-tools.dsic.upv.es/maudev (2022).

[10] G. Puebla, F. Bueno, M. V. Hermenegildo, Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs, in: Proceedings of the 9th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 1999), Vol. 1817 of Lecture Notes in Computer Science, Springer, 2000, pp. 273–292.

[11] N. Vazou, E. L. Seidel, R. Jhala, Liquid Haskell: Experience with Refinement Types in the Real World, in: Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014, 2014, pp. 39–51.

[12] S.-C. Khoo, K. Shi, Program Adaptation via Output-Constraint Specialization, Higher-Order and Symbolic Computation 17 (1) (2004) 93–128.

[13] M. Alpuente, A. Cuenca-Ortega, S. Escobar, J. Meseguer, Partial Evaluation of Order-sorted Equational Programs modulo Axioms, in: Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016), Vol. 10184 of Lecture Notes in Computer Science, Springer, 2016, pp. 3–20.

[14] F. Logozzo, T. Ball, Modular and Verified Automatic Program Repair, in: Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2012), Association for Computing Machinery, 2012, pp. 133–146.