STRASS User Guide

MARÍA ALPUENTE, SANTIAGO ESCOBAR, JULIA SAPIÑA, DANIEL GALÁN

VRAIN, Universitat Politècnica de València
{alpuente,sescobar,jsapina,dgalan}@upv.es

DEMIS BALLIS

DMIF, University of Udine
demis.ballis@uniud.it

September 2022

What is STRASS?

STRASS is an automatic safety enforcement tool for Maude programs that can take a potentially unsafe program as input and automatically transform it into a new program that is safe w.r.t. a user provided specification denoting the notion of safety for the modeled domain.

STRASS supports all of the features that are supported by Core Maude v3.2.1, including its recently defined strategy language. This means that STRASS can deal with non-deterministic and non-terminating rewrite theories that allow concurrent and complex systems to be modeled.

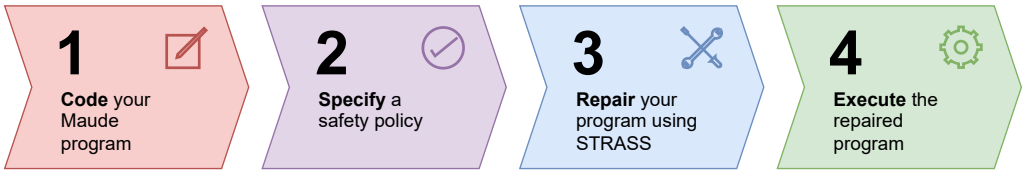


Fig. 1. Graphical outline of the STRASS usage process

Consider you have a (conditional) rewrite theory $\mathscr{R}$ whose rules define the concurrent rules of a system. It is possible that the program contains mistakes, or that it is more general than required, and these two facts might lead to anomalous, unsafe computations that:

i) may reach potentially hazardous system states and also
ii) may include meaningless or unsafe transition sequences.

These two varieties of abnormal computations are respectively related to two *levels* of safety control: safety w.r.t. states (i), and safety w.r.t. transitions (ii).

Manually checking the safety of a Maude program would require complete comprehension of the implementation alongside the relevant semantics of the Maude language, and this process is likely to be slow and error prone.

Is there a way to automatically guarantee safety by simply leveraging my domain knowledge?

By using STRASS, it is possible to automatically correct a potentially unsafe program $\mathscr{R}$ by transforming it into a new program $\mathscr{R}'$ that is safe w.r.t. a *safety policy* $\mathscr{A}$. The safety policy is a specification that you must provide to express the notion of safety through a series of invariants and strategy definitions, as described in the upcoming sections.

The overall process of using STRASS (Fig. 1) is simple: first, model your system as a suitable Maude program, and then provide your safety policy for the system. These two components can be fed to STRASS so that it produces a repaired program that can be retrieved, stored, and executed like any other Maude program.

STRASS can be particularly useful for controller synthesis, where sophisticated rewrite theories can unfold wide computational spaces comprising many possible system states and transitions which may be potentially undesirable in the corresponding domain.

Underlying technique at a glimpse

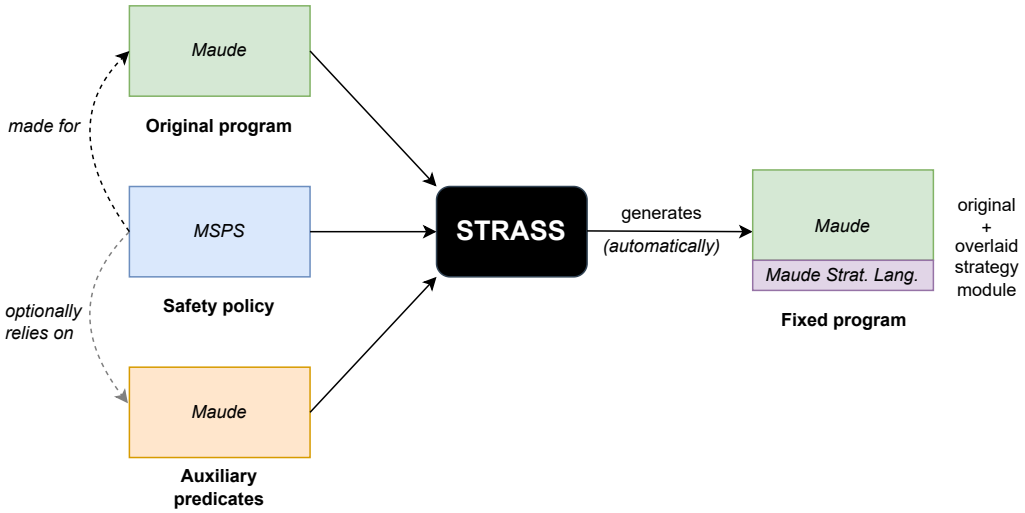


Fig. 2. High-level view of the inputs and outputs of STRASS

The correction technique implemented by STRASS is based on the newly defined Maude strategy language [**?**], which enables fine control on the application of rewrite rules, hence imposing an external, programmable control layer on top of an existing program. STRASS applies a dedicated program transformation that translates the "external" user provided specification into an "internal" Maude strategy module that lays on top of the original program. This overlaid strategy module drives the program execution so that bad runs are dropped out while good runs are kept. In other words, the transformation implemented in STRASS enjoys the following properties:

- **Completeness**: all safe computations in the original program $\mathscr{R}$ are kept in $\mathscr{R}'$, while all unsafe computations are dropped out.
- **Correction**: all computations in the fixed program $\mathscr{R}'$ are also computations in $\mathscr{R}$, i.e., no spurious computations are introduced.

As shown in Fig. 2, STRASS takes as input a Maude program and a safety policy that is encoded in a dedicated language called Maude Safety Policy Specification Language (MSPS), and returns a new Maude program that is safe w.r.t. the provided safety policy. Optionally, and for the user's convenience, a set of *auxiliary predicates* for defining the safety policy may be provided. These predicates are a series of Maude definitions (e.g., operators and equations) that are only brought into the scope of the safety policy.

Features of STRASS

The key points of the STRASS tool are summarized as follows:

- Efficient and easy to use: a program may be fixed with only three steps using a friendly assistant-style user interface.
- Fast and performant: most programs can be fixed in milliseconds of transformation time.
- 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.

Defining a safety policy in STRASS

A safety policy for $\mathscr{A}$ for a rewrite theory $\mathscr{R}$ can contain *state assertions* and *path strategies* that constrict computation paths to follow a series of actions (i.e., rule applications). These constructs cover the two levels of safety presented: safety w.r.t. states and safety w.r.t. transitions, respectively. Safety policies accepted by STRASS are expressed in the Maude Safety Policy Specification Language (MSPS), and are encoded as text containing one statement per line according to the following Extended Backus–Naur grammar:

$$
\begin{array}{rcl}
\langle \text{policy} \rangle & ::= & \langle \text{statement} \rangle \ \textit{newline}? \mid \langle \text{statement} \rangle \ \textit{newline} \ \langle \text{policy} \rangle \\
\langle \text{statement} \rangle & ::= & \langle \text{state-assrt} \rangle \mid \langle \text{path-strat} \rangle \\
\langle \text{state-assrt} \rangle & ::= & \textit{term} \ '\texttt{\#}' \ \textit{term} \\
\langle \text{path-strat} \rangle & ::= & '\texttt{path for}' \ \textit{identifier} \ '\texttt{:}' \ \textit{strategy}
\end{array}
$$

where *term* is a valid Maude term, *identifier* is a valid Maude identifier, and *newline* is a terminal symbol that specifies the special end-of-line character.

A state assertion (a.k.a. system assertion) consists of a pair $\Pi\#\varphi$ that specifies a generic safety property of the software system encoded by $\mathscr{R}$, 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 in all of the system states that match (modulo equations and axioms) the state template Π. Formulas are built using the usual Boolean operators, where the truth values are given by the formulas `true` and `false`, and the usual conjunction (`and`), disjunction (`or`), and negation (`not`) are used to express composite properties. Variables in the formulas are not quantified. Besides Boolean operators, φ may include Maude built-in operators as well as user-defined auxiliary predicates/functions that can be equationally specified.

Example 1
If `op (_,_) : Nat Nat -> NatPair` is a constructor of pairs of natural numbers, then the state assertion `(N:Nat, M:Nat) # N:Nat < M:Nat` indicates that, in all pairs, the first component must be strictly less than the second component. Thus, for instance, the pair $(1,2)$ is considered safe, but the pair $(3,3)$ is considered unsafe because $3 \not< 3$. Similarly, all states that contain the pair $(3,3)$ as a subterm are also deemed entirely unsafe, even if they contain other operators of unrelated sorts. The safe program computed by STRASS will never return a trace containing a state that includes an unsafe element such as $(3,3)$, according to the provided policy.

When several state assertions are specified, they can be considered to be implicitly joined with a logical conjunction: given a safety policy $\mathscr{A}$ containing state assertions and a state t, t is *safe* w.r.t. $\mathscr{A}$ iff all state assertions in $\mathscr{A}$ hold in t. In the case when no subterm of t matches Π, $\Pi\#\varphi$ trivially holds in t, which roughly speaking means that "irrelevant" state assertions are ignored in this implicit union.

Even if two states t and t' are safe, what may be unsafe is the way the overall transition has taken place. An approach that is purely based on state assertions is unable to reason about the admissible transitions between safe states. This is the reason why a second kind of statement is considered in MSPS: path strategies.

A path strategy statement enables the user to attach a strategy expression to the safety policy while relating it to a sort in $\mathscr{R}$ and to intuitively claim that all terms of sort s must evolve as indicated by the accompanying strategy. This associated sort enables a conceptual filtering such that only terms representing (relevant) system states are affected: for example, adding a path strategy for a sort `State` will not affect how standalone terms of sort `NatList` may evolve. However, a path strategy may cover all the subterms of the relevant terms: subterms of sort `NatList` that are contained *inside* a term of sort `State` may be controlled if the relevant path strategy for `State` indicates so. Only one path strategy may be declared per each sort. The syntax of path strategies is `path for` s `:` ζ where s is the sort that this path strategy is being associated to and ζ is a strategy expression.

Strategy expressions ζ in path strategies comprise a subset of Maude's own strategy language [**?**] and follow this grammar:

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

For the user's convenience, STRASS supports an additional custom operator that is especially useful when expressing safety policies: the *all except L* operator, denoted `all-(`L`)` in MSPS, where L is a comma-separated list of rule labels. In programs where some rule acts as a time advancement or where there exists an evident asymmetry between certain groups of rules, it may be useful to specify that we wish to apply all possible rules except a certain set of excluded rules. This operator performs exactly this function, acting as a restricted counterpart of the built-in `all` strategy constructor. The new operator `all-(`L`)` is defined as a macro that expands to Maude's strategy language as follows:

$$\texttt{all-(}L\texttt{)} = \ell_1\ \texttt{|}\ \ldots\ \texttt{|}\ \ell_n \qquad \text{where } \{\ell_1,\ldots,\ell_n\} = \mathit{labels}(R) \setminus L$$

where $\mathit{labels}(R)$ is the set of all of the rule labels in the rewrite rule set R.

Let us illustrate the MSPS language by means of an example.

A simple Maude model for a space satellite

The following program models in Maude a satellite that orbits Earth and can rotate on itself in order to extract scientific information from any point of interest located in space. These data are stored in an on-board memory and are eventually sent to mission control when the probe is facing Earth. Since the satellite orbits our planet, it is visible from mission control only during certain periodic windows of time and is occluded by the mass of the planet for the rest of the time. Attempting to communicate when the probe is not visible from mission control is fruitless, even if it is properly pointing towards Earth. The satellite is also equipped with scientific equipment that can analyze the collected data on the fly, and this equipment may turn on or off dynamically according to certain needs and conditions, such as the ones posed by energy management requirements.

The controller keeps track of the system state by retaining three values that are respectively associated with three state variables called `pm` (pointing mode), `gv` (ground visibility), and `is` (instrument status). Each variable stores one value at a time from a predetermined and finite selection of allowed values, e.g., the variable `gv` may equal either `visible` or `notVisible`. Variables may change their value, but not all value transitions are allowed. The state variables, their associated values, and their enabled transitions are summarized in Fig. 3. The controller also contains a series of synchronized clocks that advance in lockstep, tracking how long each state variable has held its current value.

Let us consider an example instance t of the probe's state:

```
t = { pm: 12 earth,  gv: 28 notVisible,  is: 7 idle }
```

In this state t, the pointing mode (`pm`) indicates that the probe is looking towards Earth and has been keeping this rotation configuration for 12 time units; also, the probe has not been visible from mission control for 27 time units, as denoted by the value of the `gv` variable; and finally,

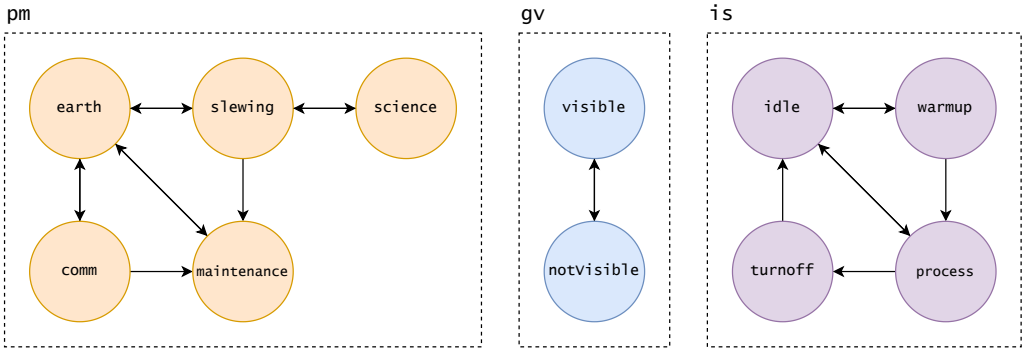


Fig. 3. Representation of the enabled transitions for state variables of the space probe

the on-board instruments (`is`) have been idling for 7 time units.

Further temporal restrictions exist when a variable attempts to change its value. In fact, each value has a minimum and maximum time bound that must be assigned to a variable. Once a variable has acquired a value, it must remain "locked" until the associated clock has passed the minimum bound, and then it must change before the clock reaches the maximum bound. When a variable changes its value, the related clock resets to zero time units. The time bounds for the different values are summarized in Table 1.

Variable	Value	Min. duration	Max. duration
pm	earth	1	$+\infty$
	slewing	30	30
	science	36	58
	comm	30	50
	maintenance	90	90
gv	visible	60	100
	notVisible	1	100
is	idle	1	$+\infty$
	warmup	5	5
	process	5	5
	turnoff	5	5

Table 1. *Duration bound restrictions for state values of the space probe*

We can use the MSPS language to specify a safety policy $\mathscr{A}_{\text{SATELLITE}}$ encoding representative domain constraints. Note that, even though here some statements use more than one line due to page space constraints, each statement should only occupy one line in the corresponding MSPS source code:

(1) `T:Time # T:Time >= 0`

(2) `{ pm: Tpm:Time comm, gv: Tgv:Time GV:GroundVisibility,`

```
     is: Tis:Time IS:InstrumentStatus } # GV:GroundVisibility == visible

(3) { pm: Tpm:Time maintenance, gv: Tgv:Time GV:GroundVisibility,
     is: Tis:Time IS:InstrumentStatus } # IS:InstrumentStatus == idle

(4) { pm: Tpm:Time maintenance, gv: Tgv:Time notVisible,
     is: Tis:Time IS:InstrumentStatus } # false

(5) { pm: Tpm:Time science, gv: Tgv:Time notVisible,
     is: Tis:Time IS:InstrumentStatus } # Tis:Time <= 2 * max(duration(process))
```

Also, the following path strategy can be provided as an additional statement:

```
(6) path for State : (all-(advance-time) | idle) ; advance-time
```

Finally, this safety policy relies on the following auxiliary operator `max` that is used in statement (5):

```
var Tmin Tmax : Time .

op max : TimeInterval -> Time .
eq max([Tmin, Tmax]) = Tmax .
```

Statement by statement, the safety policy at hand imposes the following constraints on the system's domain:

Statement 1. **Time values (e.g., durations) must be either zero or positive.**
It may be common for Maude programs to contain definitions that are broader than what is actually required for a certain scenario, especially when importing already existing modules for the purpose of code reuse. We simulate this situation in this example: the auxiliary module `TIME` contains some generic, program independent definitions for the concept of time, using (potentially negative) integers as time values. In this case, however, we only wish to use the domain of naturals, being zero the first valid time instant. We thus use this assertion to match any time value and explicitly indicate it must be equal to or greater than 0.

Statement 2. **The satellite may only communicate with mission control in a time window where both can see each other.**
The space probe at hand communicates periodically with Earth, but in order for this communication to occur, the probe and the antennas located at mission control must be visible from each other. This happens during certain windows of time, as the probe rotates and is sometimes occluded by our planet. Two conditions must align simultaneously: (i) the probe must be pointing towards Earth (`pm = earth`); and (ii) the probe must be visible from mission control (`gv = visible`). We encode this pair of conditions using the pattern to match states where the satellite attempts to communicate, while the formula enforces ground visibility.

Statement 3. **During maintenance operations, the on-board scientific instruments must remain inoperative.**

In order to ensure the safety of on-board instruments, they must not execute any task during the entire time when maintenance operations take place. We encode this restriction in a way that is similar to the previous one: the pattern matches a maintenance situation, and then the condition enforces that the instruments must be idling.

Statement 4. **The probe cannot be in maintenance mode while not visible.**
This state assertion is an example of marking undesirable states leveraging pattern matching. In this case, the pattern is enough to capture the semantics of the unsafe situation: a term having both `maintenance` and `notVisible` in their respective slots as indicated. The trivial formula `false` then explicitly forbids the family of terms captured.

Statement 5. **During scientific data collection operations, the on-board instruments will not idle for longer than double the maximum processing time allowed.**
The time dedicated to data collection operations is valuable, and hence, it is important for the on-board instruments to remain operative and to be actively used during said time frames. Thus, in case the instruments are idling, they must not spend more than $2M$ units of time in such a state, where M is the maximum processing time allowed. In order to define this assertion, we define an auxiliary operator `max` which obtains the second component of the pair (m,M) (i.e., the maximum duration bound) from a function that returns duration bounds as pairs. This auxiliary operator is not part of the original program and is hence provided as part of the auxiliary predicates of the safety policy.

Statement 6. **State variable change transitions and time advancement transitions must be intertwined.**
Transitions changing the value of the different state variables and transitions advancing time are independent and can be applied in any order and manner. In reality, time advances at regular clock cycles that allow at most one state change to occur. Hence, we use a path strategy to enforce that transitions on states must have one of the following forms: (i) no state variable changes and then time advances; or (ii) exactly one state variable changes and then time advances. After the time advancement "tick" is complete, this process repeats. Thus, a situation where two consecutive state variable changes occur with no time advancement in between is not allowed.

Note that, in the strategy expression

```
(all-(advance-time) | idle) ; advance-time
```

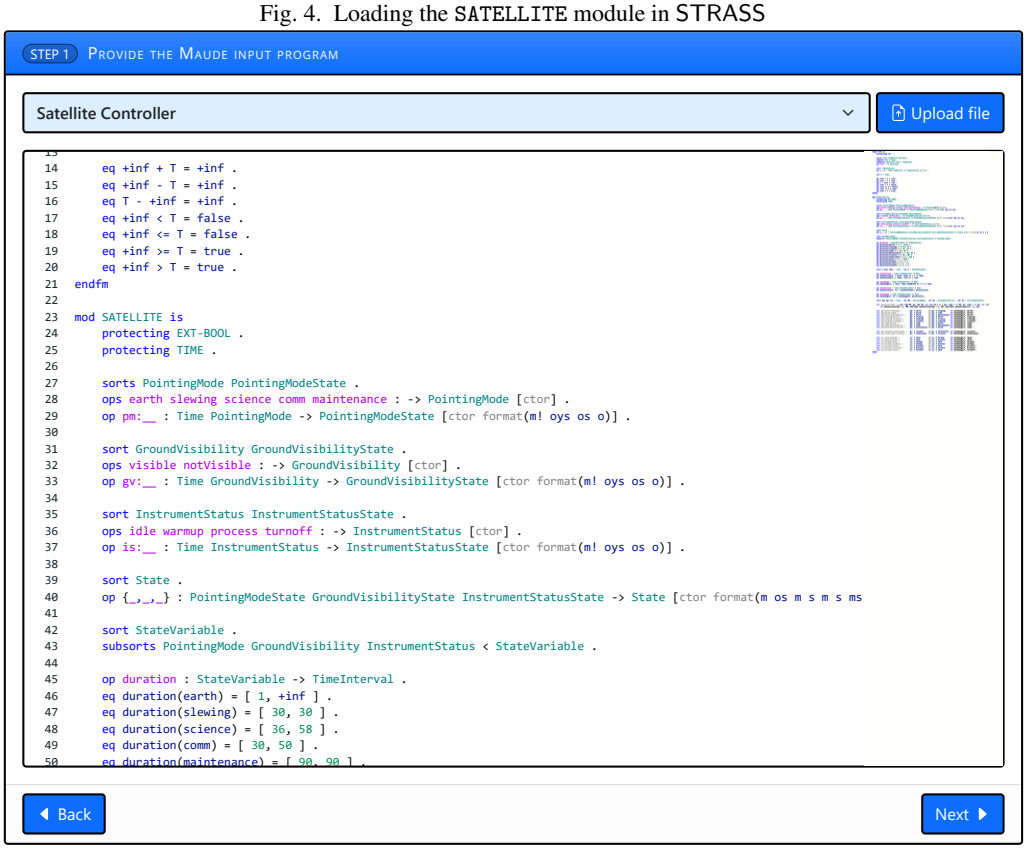
we use `all-(advance-time)` to refer to all of the program rules except for the time advancement rule, which, in this case, corresponds to all rules encoding the enabled transitions that change the state variables. The strategy sub-expression at the left of the semicolon (`;`) indicates that *at most* one of such state-changing rules should be applied, where `idle` represents no rule application. Finally, the sequential concatenation of `advance-time` indicates that the time advancement rule *must* be applied exactly once, concluding what is conceptually one clock cycle.

Enforcing Program Safety Using STRASS

Let us illustrate how STRASS works in practice by showing a typical safety enforcement session for our satellite controller.

The tool first presents a landing page, where the user must press Start in order to proceed. Then, as a first step, the tool requires the user to provide the Maude input program. Input programs can be provided directly by using the dedicated editor area, by uploading an existing `.maude` file pressing the Upload file button, or by selecting one of the preloaded representative examples included in the tool for demonstration purposes. In this case, we select the example named *Satellite Controller* from the collection of examples (see Fig. 4), whose `SATELLITE` module encodes the satellite controller program $\mathscr{R}_{\texttt{SATELLITE}}$ described in this guide.
In the case of input programs containing several modules, STRASS assumes that the input

Fig. 4. Loading the `SATELLITE` module in STRASS

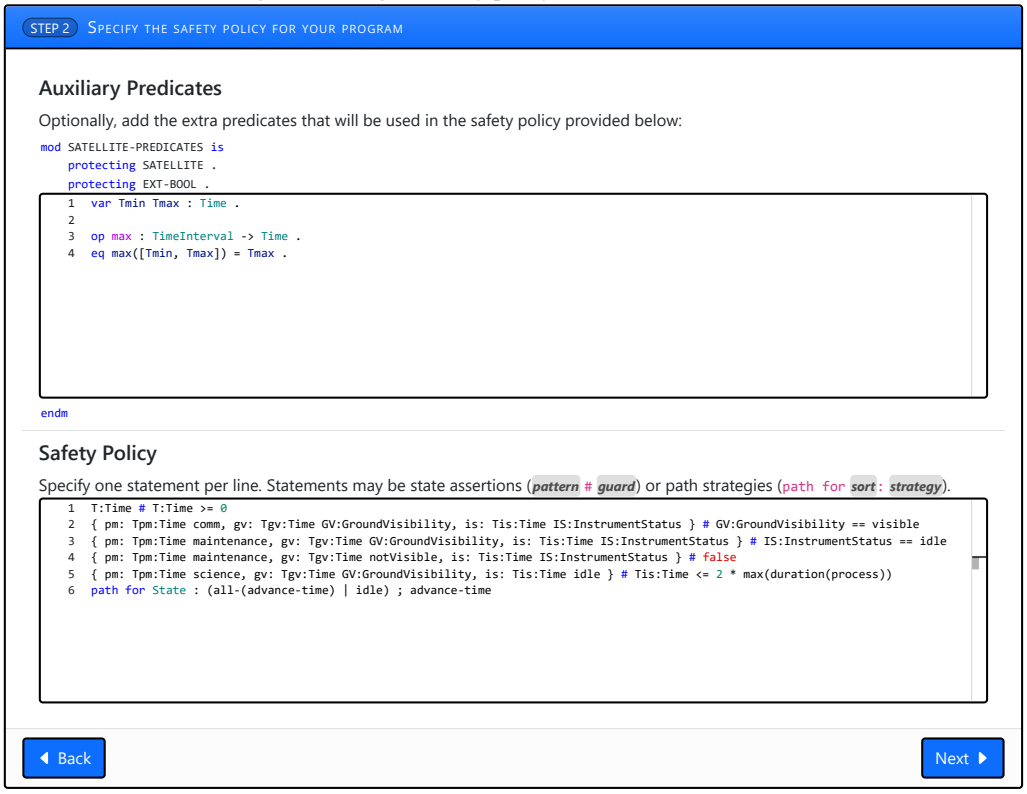


program, following the usual organization of a Maude program, has its *root module* as the last module in the source code. This root module is the top of the module import hierarchy, and, hence, it transitively imports all the other modules, containing all of the definitions in its scope.

After pressing the Next button, the following phase allows the user to specify the safety policy

in order to also enforce the optional auxiliary predicates that it may rely on (see Fig. 5). In this case, because the *Satellite Controller* example has been selected, the corresponding editing areas have already been prefilled with the safety policy $\mathscr{A}_{\texttt{SATELLITE}}$ and the `max` user-defined function presented in the previous section.

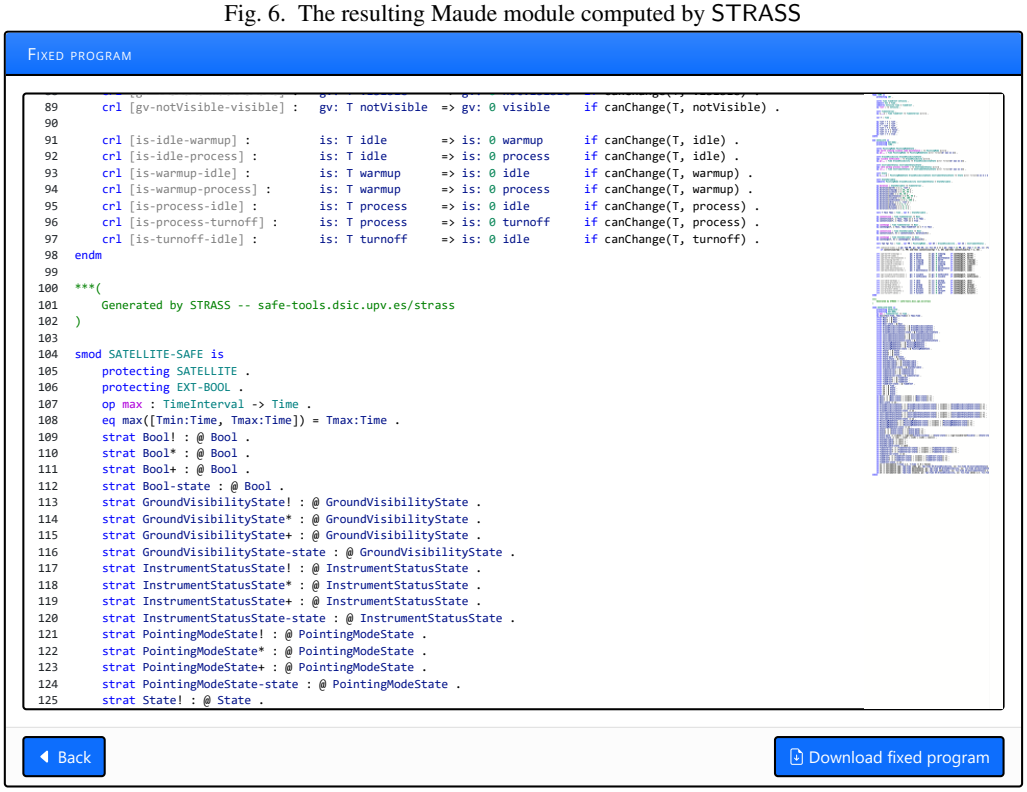
Fig. 5. Loading the safety policy $\mathscr{A}_{\texttt{SATELLITE}}$ in STRASS



After pressing Next a second time, STRASS will automatically generate the strategy module `SATELLITE-SAFE` encoding $\mathscr{A}_{\texttt{SATELLITE}}$, which is overlaid on top of the original program $\mathscr{R}_{\texttt{SATELLITE}}$, resulting in a new safe program $\mathscr{R}'_{\texttt{SATELLITE}}$ that is entirely self-contained and may be fed to a Maude interpreter instance running in the user's device. The outcome is given to the user inside the read-only dedicated text area (see Fig. 6). Using the resulting program, the user can produce safe program executions by issuing appropriate `srew` Maude commands that are driven by the synthesized strategy definitions as explained in the following section.

Executing Safe Programs

Fig. 6. The resulting Maude module computed by STRASS



The safe program computed by STRASS contains the original code of the input program plus an additional strategy module juxtaposed at the end. This new module shares the name of the root module of the original program (i.e., the last module of the original program) followed by the suffix `-SAFE`. For instance, in the case of the satellite controller, the new module will be called `SATELLITE-SAFE`.

The `search` command allows the Maude user to explore the computation space of a program. However, safe program executions require the user to employ the dedicated strategy-oriented command `srew` (*strategy rewrite*) in order to apply the control imposed by the syntehsized strategy definitions, even if said definitions are in the current scope. Accordingly, using the generated strategies, the `srew` command explores a constrained, safe version of the computation space where all states and transitions between states satisfy the given safety policy. Indeed, the strategy definitions computed by STRASS leverage the acquaintance with the `search` command to carefully expose a conceptually *safe equivalent* of the `search` command as summarized in Table 2. The user of STRASS *must* always use strategy-aware commands such as `srew` to ensure that the computed traces comply with the posed safety guarantees.

Example 2
Consider again satellite controller specification. Given the initial state

```
{ pm: 40 comm, gv: 70 visible, is: 0 idle }
```

the following `search` command explores the next four states stemming from it by applying one

Safe command	**Unsafe equivalent**	**Description**
`srew` *t* `using` *S*`*`	`search` *t* `=>*` `X:`*S*	Obtain the computation traces stemming from *t* by applying rules zero or more times.
`srew` *t* `using` *S*`+`	`search` *t* `=>+` `X:`*S*	Obtain the computation traces stemming from *t* by applying rules one or more times.
`srew` *t* `using` *S*`!`	`search` *t* `=>!` `X:`*S*	Obtain the normal forms reachable from *t*.

(where S denotes the sort of the initial term t)

Table 2. *Comparison of* `srew` *and* `search` *as safe and unsafe counterparts for computation space exploration*

or more rules, as indicated by `=>+`.

```
search [4]  pm: 40 comm, gv: 70 visible, is: 0 idle  =>+ S:State .
Solution 1 S:State ->  pm: 41 comm, gv: 71 visible, is: 1 idle
Solution 2 S:State ->  pm: 0 earth, gv: 70 visible, is: 0 idle
Solution 3 S:State ->  pm: 0 maintenance, gv: 70 visible, is: 0 idle
Solution 4 S:State ->  pm: 40 comm, gv: 0 notVisible, is: 0 idle
```

The returned solutions correspond to states that fulfill basic domain restrictions such as duration bounds, but they are not necessarily safe w.r.t. the specified safety policy. The first solution corresponds to one application of the time advancement rule. The second solution arises from the transition of the pointing mode (`pm`) from `comm` to `earth`. The third solution results from the transition of `pm` from `comm` to `maintenance`. The fourth solution changes the value for the `gv` variable and resets the associated clock.

The solutions computed by the search command exposed several violations of the safety policy. Firstly, in those transitions involving the `pm` variable (i.e., Solutions 2 and 3), the time didn't advance, which violates the constraint that a clock cycle requires the intertwining of state variable changes and time advancement. Secondly, Solution 4 corresponds to an unsafe state because we explicitly forbid the communication pointing mode (`pm` = `comm`) while the probe is not visible from mission control (`gv` = `notVisible`).

Let us now explore the search space of the resulting safe program by using the `srew` command.

```
srew [4]  pm: 40 comm, gv: 70 visible, is: 0 idle  using State+ .

Solution 1
result State:  pm: 41 comm, gv: 71 visible, is: 1 idle

Solution 2
result State:  pm: 1 earth, gv: 71 visible, is: 1 idle

Solution 3
result State:  pm: 1 maintenance, gv: 71 visible, is: 1 idle

Solution 4
result State:  pm: 42 comm, gv: 72 visible, is: 2 idle
```

In this case, four states are also returned as before, but all of them do satisfy the safety policy. Note that, in Solutions 2 and 3, the state variable change is now accompanied by the corresponding time advancement. This is shown by the duration of `gv`, which increases from `70` to `71`. Furthermore, the previously unsafe Solution 4 has now been suppressed in favor of a new, different solution that is safe. This solution is the next one in the search space exploration algorithm implemented by the Maude `search` command.[1] Indeed, it would have been Solution number 5 in the previous `search` command if it were not limited to only showing the first four solutions.

The named strategy definitions shown by STRASS to the user are summarized in Table 3. The advanced STRASS user may combine those strategies freely in any context where the Maude interpreter supports the use of the Maude strategy language.

Strategy name	**Applies safety policy?**	**Semantics**
S`*`	Fully	All states reachable after zero or more rule applications.
S`+`	Fully	All states reachable after one or more rule applications.
S`!`	Fully	Normal forms reachable with any number of rule applications.
S`-path`	Path strategies only	Applies the path strategy associated to S exactly once.
S`-state`	State assertions only	Returns t if the state assertions in the policy hold in t; otherwise, it returns $\emptyset$.

Table 3. *Overview of the named strategy definitions computed by STRASS*

What do I do now?

To learn more about STRASS, visit our website at

`http://safe-tools.dsic.upv.es/strass`

[1] Maude `search` command explores the program search space by using a breadth-first visit of the associated computation tree.