UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dept. of Computer Systems and Computation

Safety enforcement via programmable strategies in Maude

Master's Thesis

Master's Degree in Software Systems Engineering and Technology

AUTHOR: Galán Pascual, Daniel

Tutor: Alpuente Frasnedo, María

Experimental director: SAPIÑA SANCHIS, JULIA

ACADEMIC YEAR: 2021/2022

Acknowledgements

This MSc. thesis has been developed through a research contract with the Extensions for Logic Programming (ELP) research group, which is part of the Valencian Research Institute for Artificial Intelligence (VRAIN) at the Universitat Politècnica de València.

The work developed in this thesis and its underlying research stands on the foundations laid by María Alpuente, Santiago Escobar and Julia Sapiña, from the ELP group, as well as Demis Ballis, from the Department of Mathematics and Computer Science at the University of Udine, Italy, whose joint efforts have contributed immensely to the Maude ecosystem and the logic programming community.

It is also important to thank all the MITSS professors who, with their renowned education in logic and formal methods, make possible year after year the master's degree that, in my case, this thesis proudly concludes.

The research activities associated to this work have been partially funded by the Generalitat Valenciana under grant PROMETEO/2019/098.

Resum

L'objectiu d'aquest treball és formalitzar un mecanisme general per a garantir la seguretat de les computacions en lògica de reescriptura. La tècnica proposada consisteix en una transformació de programes que garanteix que el sistema transformat satisfà els invariants de seguretat definits per l'usuari, evitant així que el sistema aconsegueixi estats considerats insegurs. La transformació és genèrica i no depèn d'estratègies ad hoc, sinó que aquestes es defineixen com a assercions que es tradueixen de manera automàtica al llenguatge d'estratègies de Maude, recentment definit. La tècnica proposada s'ha implementat en una eina anomenada STRASS i la seva eficiència i escalabilitat s'han avaluat empíricament amb excel·lents resultats que demostren la seva viabilitat a la pràctica.

Paraules clau: lògica de reescriptura, Maude, llenguatge d'estratègies, seguretat, síntesi de controladors, transformació de programes

Resumen

El objetivo de este trabajo es formalizar un mecanismo general para garantizar la seguridad de las computaciones en lógica de reescritura. La técnica propuesta consiste en una transformación de programas que garantiza que el sistema transformado satisface los invariantes de seguridad definidos por el usuario, evitando así que el sistema alcance estados considerados inseguros. La transformación es genérica y no depende de estrategias ad-hoc, sino que éstas se definen como aserciones que se traducen de forma automática al lenguaje de estrategias de Maude, recientemente definido. La técnica propuesta se ha implementado en una herramienta llamada STRASS y su eficiencia y escalabilidad se han evaluado empíricamente con excelentes resultados que demuestran su viabilidad en la práctica.

Palabras clave: lógica de reescritura, Maude, lenguaje de estrategias, seguridad, síntesis de controladores, transformación de programas

Abstract

This work aims to formalize a general mechanism for safety enforcement in rewriting logic computations. The proposed technique consists of a program transformation that guarantees that the transformed system satisfies user-defined invariants, thus preventing the system from reaching states considered insecure. The transformation is generic and does not depend on ad-hoc strategies but relies on the newly defined Maude strategy language. The proposed technique has been implemented in a tool called STRASS and its efficiency and scalability have been empirically evaluated with excellent results which demonstrate that the technique pays off in practice.

Key words: rewriting logic, Maude, strategy language, safety, controller synthesis, program transformation

Contents

List of Figures

List of Tables

List of Code Listings

CHAPTER 1

Introduction

Our society increasingly depends on software in order to deliver its benefits and to achieve its goals. Year after year, from industrial equipment to personal vehicles, most technology around us is driven by software instructions that can be uninterruptedly executed for hundreds or even thousands of hours. This automation often comprises complex, concurrent and reactive systems which need to be aware of its environment and react to it in real time. As automated systems embrace more important roles in our society, the risks involved in possible malfunctions become more evident and critical: systems guided by defective software may lead to monetary loss, environmental damage or even harm to human life.

Complex, concurrent systems can be encoded in many different programming languages, of which Maude is one of them. In the case of Maude, such concurrent systems can be encoded as rewrite theories that describe, means of rules, the transitions between system states, which are encoded as terms of an algebraic datatype. Just like in any other programming language, constructs such as rewrite rules are written and added to the program's source code by programmers and domain experts. Those human agents may incur in mistakes arising from a bad understanding of details and requirements that are inherent to the problem domain, or from a deficient encoding of the expected semantics. Software specification errors can manifest in different ways and with a wide range of consequences and severity levels. Given the importance of properties such as correctness and completeness of models and programs, several approaches have been developed in order to (semi-)automatically ensure that existing programs have those desirable properties. One of such approaches is model checking [13], which is available for Maude programs [15, Chapters 11-12].

Another important property besides software correctness is *safety* of the system. We say a program is *safe* when it cannot accidentally cause harm to its environment, because it passively or actively remediates or mitigates possible hazards that can be or derived from its operation. One way to approach the problem of ensuring the safety of a software system is *invariant enforcement*. Invariants are logic assertions that specify some constraints that must always hold in a given software, e.g., the fact that an elevator must never move with its doors open is an invariant in the domain of elevator controllers. Invariant enforcement refers to the set of techniques that pursue the guarantee that some given invariants actually hold in a certain program. There are several approaches to invariant enforcement: static software verification techniques can ensure the compliance of the program w.r.t. some invariants without executing the program, while some dynamic techniques ensure the invariants are not violated while said program is running. Static software verification techniques are varied but sometimes cannot contend with some properties, or may over- or under-approximate results, while dynamic techniques may offer precise results that are closer to the real behavior of the system at the cost of completeness or run time performance. This is especially important for the complex systems often found in safety-critical scenarios [19].

In this thesis, we explore a dynamic safety enforcement technique that can ensure programs written in the Maude language don't deploy derivations where invariants are violated, acting as a sentinel of safety for the program computations. This work is based on the newly defined Maude strategy language [22], which enables fine control on the application of rules, hence imposing an external, programmable control layer on top of an existing program. By relying on the Maude strategy language, we have have defined a new, logical specification language that allows safety invariants to be defined and overimposed to a given Maude program. We have developed a program transformation that translates such a "external" safety policy specification into a "internal" Maude strategy module that lays on top of the original program. Moreover, our safety specification language is rich enough to not only reason about the safety of states but also safety the transitions. One important advantage of our approach is that safer, more refined versions of a program can be incrementally built without any programming effort, enabling the inexperienced user to largely forget about Maude syntax and semantics.

The technique presented here improves existing preliminary approaches for safety enforcement in Maude programs [8, 9]. Our proposal leverages the programmable Maude strategy language and so avoids depending on ad-hoc, hard-coded safety checks that are intertwined with the existing code, which could negatively impact tracing and maintenance [21]. The generated strategies are integrated alongside the original program but kept separate in its own module, allowing for incremental refinement to be built by simply appending new logical constraints, and also avoiding the need to rely on an external, expensive system monitor. On top of that, the techniques presented in [8, 9] only apply to *topmost* rewrite theories (i.e., programs where terms can only be rewritten at the root position) while the proposed technique at hand covers a wider class of programs that are not necessarily topmost, and that optionally make use of advanced Maude language features previously unsupported.

A friendly, user-facing tool called STRASS, which is publicly available as a web application, efficiently implements this framework. The efficiency and scalability of its implementation has been empirically evaluated by using a series of benchmarks with excellent experimental results.

The structure of this thesis is as follows. Chapter 2 covers the preliminaries by presenting the Maude programming language alongside its underlying rewriting logic fundamentals and its strategy language. Chapter 3 introduces the new MSPS language that is specifically devised for specifying the required safety policy for a given Maude program, as well as its related program transformation technique for automatic safety enforcement. Chapter 4 presents the STRASS tool and provides technical details on its implementation. Chapter 5 provides an empirical evaluation of the efficiency and scalability of STRASS. Finally, Chapter 6 concludes this thesis by presenting related work, determining the future work to be explored and offering some discussion and insight of the obtained results.

CHAPTER 2
The Maude Programming Language

Maude is a declarative, multi-paradigm and high performance programming language and system based in rewriting logic [15, 20]. Maude integrates concurrent, functional and logic programming, and it also includes features from other paradigms, such as object-oriented programming. Maude acts simultaneously as a general-purpose programming language and as a formal logic framework, allowing the user to define and run formal specifications that may be subject to a broad range of formal methods without giving up properties such as executability or I/O capabilities.

Given the formal properties of rewriting logic, as introduced in the subsequent sections, Maude is useful as a semantic and methodological framework for defining the semantics of a broad range of languages and formal systems, and is especially capable of handling modeling and the execution of complex concurrent, reactive systems. Furthermore, its extensive meta-programming capabilities have paved the way for a considerable tooling ecosystem, including theorem provers [14], debuggers and code optimizers [10], and cryptographic protocol analyzers [17], among many others.

The following sections provide a brief introduction to the Maude programming language alongside its underlying rewriting logic basis. The considered version is 3.2.1 [15], the most recent at the moment of writing this work. More information and resources about the Maude language may be found at `http://maude.cs.illinois.edu`.

2.1 Rewriting Logic Fundamentals

Rewriting logic (RWL) [20] is a many-sorted (i.e., typed) logic that is based on membership equational logic. RWL empowers the user to describe the evolution of a concurrent system in a simple, expressive and effective manner: states are represented as *terms*, *equations* allow those terms to be simplified, and *rewrite rules* define transitions that cause those states to change. Similarly to other kinds of logic, rewriting logic can be seen as a model and framework for unambiguously encoding some knowledge about our world, as well as reasoning about it by inferring conclusions using a set of rules.

The series of statements that assert knowledge we know to be true are called *axioms*. A *theory* is a set of statements that are subject to be proven before they can be considered to be true or false. A *formal system* is an abstract structure containing a notation and a set of rules (called the *logical calculus* of the system) that can be used to infer which theorems are true or false given some axioms. Rewriting logic has been efficiently implemented in the programming language and system Maude, which is introduced in the following sections.

Consider a collection of typed *operator* symbols called a *signature* and denoted Σ, where each operator has some arity and may be prefix, suffix or mixfix. The recursive valid[1] applications of these operators form *terms*. For instance, the term `0 + 1` is an example of the application of the binary operator `+` to `0` and `1`. In turn, `0` and `1` are also operators, of arity 0, as they do not take any input. Such 0-arity operators are called *constants*. Terms are the fundamental means of encoding knowledge and may represent a wide variety of system states or data. By defining new operators, the user is able to define a personalized grammar that can be used to form terms of any shape and for any purpose.

Terms may be represented as trees where a node is an operator and its children are the different operators and variables it is applied to. A *position* w of a term t is a hierarchical sequence of natural numbers that addresses a *subterm* of t by indicating how to traverse its tree until a certain node is found. Let $t_{|w}$ denote the subterm of t at position w. The set of valid positions of a term t is denoted by $Pos(t)$. The special position λ denotes the root position of a term tree, such that $t_{|\lambda} = t$. For example, $((1 + (2 - 0)) * 3)_{|1.2}$ is $2 - 0$. The *subterm replacement* operation is denoted $t[t']_w$ and results in the term t where $t_{|w}$ has been replaced by t'. For example, $((1 + (2 - 0)) * 3)[7]_{1.2} = (1 + 7) * 3$. We use $root(t)$ to denote the symbol occurring at the top of t, e.g., $root($`1 + 2`$)$ is the binary operator `+`.

Recall that rewriting logic is a *many-sorted* logic: terms, operators, variables and other elements of the language are typed. Types classify values into several homogeneous and different sets, instead of considering all data to be part of a unique heterogeneous universe. Similarly to other statically and strongly typed languages, Maude enforces this type safety by using conventional type-checking techniques. In contrast to conventional languages, rewriting logic and Maude consider two varieties of *types*: *sorts* and *kinds*.

Sorts correspond to named sets of possible values and are analogous to the traditional concept of type as presented in many other languages. Types on operators are coherent with their arity and specify their domain and image. For instance, if `Nat` is a sort referring to all natural numbers and the operator + has signature `Nat Nat` $\to$ `Nat`, we are denoting this operator takes two natural numbers to produce a third potentially different one. We use S to denote the set of all sorts of a program. Sorts have a *subsort* relation $<$, defining typical subtype polymorphism. This relation is one of partial order and is encodable as a poset $(S, <)$. Let $s, u \in S$ be two sorts; we say s is a subsort of u (or u is a supersort of s), denoted $s < u$, if an only if all valid values of sort s are also valid values of sort u. Each term t is classified using one *least sort*, denoted $ls(t) \in S$, and corresponding to the maximal return sort of the operator symbol at the root of t, $root(t)$. We say the term t has sort $s \in S$ if $s \leq ls(t)$, and we denote this fact as $t :: s$.

A *kind* of a set of one or more sorts $\{s\}_{s \in S}$ is denoted $[\{s\}_{s \in S}]$ and is a type that corresponds to the maximal of all sorts $s \in S$ in the poset $(S, <)$. Thus, kinds act as equivalence classes where two sorts are grouped in the same equivalence class if they belong to the same connected component. In Maude, kinds are implicitly defined and do not have their own name; instead, they are represented by enclosing one or more sorts of their class in square brackets, like this: `[Nat, String]`. The previously introduced `::` notation may be naturally lifted to kinds: we say $t :: [\{s_1, \ldots, s_n\}]$ for a term t if and only if $(\exists s_i \in \{s_1, \ldots, s_n\})\ s_i \leq ls(t)$.

A *variable* is a special symbol that conceptually acts as a placeholder for any term. Consider a family of typed variables $\mathcal{V} = \{\mathcal{V}_s\}_{s \in S}$. We use $\tau(\Sigma, \mathcal{V})$ to denote the *term*

[1]This is, applications which respect the arity of each operator.

algebra induced by $\mathcal{V}$ a d Σ, this is, the set of all valid *terms* that may be constructed using operators in Σ and variables in $\mathcal{V}$. Terms without variables are called *ground terms*, and its algebra is denoted $\tau(\Sigma)$. The set of variables that appear in a term t is denoted by $Var(t)$. For example, `1 +` X is a valid term in $\tau(\Sigma, \mathcal{V})$ for the aforementioned signature if X has sort `Nat`.

A *substitution* $\sigma \equiv \{x_1/t_1, \ldots, x_n/t_n\}$ is a mapping from variables in $\mathcal{V}$ to terms in $\tau(\Sigma, \mathcal{V})$ that is equivalent to the identity relation for all variables except the finite set of variables $\{x_1, \ldots, x_n\}$ it replaces. The application of a substitution σ to a term t is denoted by the juxtaposition $t\sigma$. A term t is an *instance* of another term t' if there exists a substitution σ such that $t = t'\sigma$. The *composition* of two substitutions $\sigma\phi$ is the substitution such that $t\sigma\phi$ is equivalent to applying ϕ to the result of $t\sigma$. For instance, given the substitution $\sigma = \{X/7\}$ then `((1 +` X`) +` Y`)`σ results in `(1 + 7) +` Y.

Equations have the form $[\ell]\colon l = r$, where ℓ is an optional label and $l, r \in \tau(\Sigma, \mathcal{V})$ are terms. Equations describe how terms may be deterministically simplified in a process called *equational simplification* or equational reduction, being analogous to the concept of function definition in other languages. In Maude, equations are oriented from left to right on application: the left hand side l is a term that may contain variables, conceptually acting as a *pattern* which specifies the family of terms that will be affected by the equation, while the right hand side r indicates the result of its application on a term. Maude looks for subterms that *match* these left hand side patterns by using a mechanism called *pattern matching*. Conditional labeled equations generalize equations and have the form `[`ℓ`] :` $l = r$ `if` C where ℓ is an optional label; $l, r \in \tau(\Sigma, \mathcal{V})$ are (potentially non-ground) terms and C is a condition that can be interpreted as a Boolean expression whose normal form is a truth value. Formally, we say a term t *reduces* to t' using a (left-to-right oriented) equation $(l \overset{\to}{=} r) \in \vec{E}$ if and only if there exists a position w and a substitution σ such that $t_{|w} = \sigma l$ and $t' = t[\sigma r]_w$. This one-step reducing relation is denoted $t \overset{w}{\to}_{\vec{E}} t'$ (or simply $t \to_E t'$) and is henceforth called an *equational step*. Each instance $l\sigma$ of an equation $l = r$ is called a *redex*, which stands for **red**ucible **ex**pression. If C is not empty, the equation only applies to a redex when σC is ground and satisfied for the corresponding instance σl. We use E to denote the set of equations in a program.

A (conditional) *equational theory* $\mathcal{E}$ is a pair $(\Sigma, E_0 \cup Ax)$ (from now simply denoted as $\mathcal{E} = (\Sigma, E)$) where Σ is a signature, E_0 is a set of (conditional) equations (which are implicitly oriented from left to right) and Ax is a collection of equational axioms that are distinguished equations that can be associated with binary operators in Σ. Such equational axioms often represent well-known mathematical properties such as associativity, commutativity and unit element. Terms in $\mathcal{E}$ are semantically equal (or E-equal) when both can be reduced to a common normal (i.e., irreducible) form. This induces a congruence relation on the term algebra $\tau(\Sigma, \mathcal{V})$ that is denoted $=_E$. For example, `1 + 1` $=_E$ `2 + 0` if E defines the usual addition of natural numbers, as both have the same normal form `2`.

In Maude, equational theories are assumed to be Church-Rosser [16] (i.e., *confluent* and *sort-decreasing*) and *terminating* [15]. Roughly speaking, a theory is confluent if a term has a unique normal form regardless of the order in which equations are applied. A theory is terminating if its execution always halts, i.e., it finds a normal form for all possible "input" terms.

We consider a natural partition of the signature Σ as $\Sigma = \mathcal{D} \uplus \Omega$, where Ω is the set of *constructor symbols* and $\mathcal{D}$ are the *defined symbols*. Constructor symbols are used to define irreducible data values, whereas defined symbols are always evaluated via equational simplification. Terms in the algebra $\tau(\Omega, \mathcal{V})$ are called *constructor terms*.

Rewrite rules are the non-confluent and/or non-terminating counterpart of equations, and share a similar form $[\ell] : l \Rightarrow r$, where ℓ is an optional label and $l, r \in \tau(\Sigma, \mathcal{V})$ are terms. Conditional rewrite rules have the form `[`ℓ`] :` $l \Rightarrow r$ `if` C where, similarly as before, ℓ is an optional label; $l, r \in \tau(\Sigma, \mathcal{V})$ are terms and C is a condition. We say a term t *rewrites* to t' using $(l \Rightarrow r \texttt{ if } C) \in R$, denoted $t \xrightarrow{w}_{\mathcal{R}} t'$ (or simply $t \rightarrow_{\mathcal{R}} t'$), if and only if $\exists w, \sigma$ such that $t_{|w} = \sigma l \ \wedge \ t' = t[\sigma r]_w$ and $C\sigma$ holds. This relation is a (conditional) *rewriting step*. We use R to denote the set of rewrite rules in a program.

A (conditional) *rewrite theory* $\mathcal{R}$ is a triple $(\Sigma, E_0 \cup Ax, R)$ (or simply denoted $\mathcal{R} = (\Sigma, E, R)$) where R is a set of (conditional) rewriting rules, Σ is a signature, E_0 is a set of (conditional) equations and Ax is a collection of equational axioms that can be associated with binary operators in Σ. The components $(\Sigma, E_0 \cup Ax)$ constitute the *underlying equational theory* of $\mathcal{R}$. In Maude, rewriting theories are subject to fewer constraints than equational theories, as rules may model concurrent computations that are not terminant or confluent. However, the underlying equational theory is still subject to all constraints relevant to equational theories.

When a term t cannot be reduced anymore with the equations in E (resp., the rules in R), we say that that t is a *E-normal form* (resp., R-normal form). Given a relation $(\rightarrow_\bullet) \subseteq \tau(\Sigma) \times \tau(\Sigma)$ (e.g., the defined reduction and rewriting relations $\rightarrow_E$ and $\rightarrow_R$), let $\rightarrow_\bullet^*$ and $\rightarrow_\bullet^+$ be the transitive (and in the case of $\rightarrow_\bullet^*$ also reflexive) closures, following the typical definitions of zero-or-more steps and one-or-more steps, respectively. The reduction of a term to its normal form w.r.t. $\rightarrow_\bullet$ is called *normalization* and is denoted $t\downarrow_\bullet$.

In a (conditional) rewrite theory $\mathcal{R} = (\Sigma, E_0 \cup Ax, R)$, computations evolve by rewriting terms using the *equational rewriting* relation $\rightarrow_{R,E_0 \cup Ax}$ (or simply $\rightarrow_{R,E}$), which applies the rewriting rules in $\mathcal{R}$ *modulo* the underlying equational theory $\mathcal{E} = (\Sigma, E_0 \cup Ax)$. Given a term t, one redex is chosen following a certain strategy and then a corresponding rewriting rule is applied. Directly after that, the reached term is E_0-normalized modulo the axioms Ax. Formally, $t_i \xrightarrow{r}_{R,E_0 \cup Ax} t_{i+1}$ (or simply $t_i \rightarrow_{R,E} t_{i+1}$) if $t_i \rightarrow_R t'_i$ using rule r and $t_{i+1} = t'_i \downarrow_{E_0 \cup Ax}$. This relation is called a *Maude step*. This step is then repeated forming a sequence $t_0 \rightarrow_{R,E_0 \cup Ax} t_1 \rightarrow_{R,E_0 \cup Ax} \cdots \rightarrow_{R,E_0 \cup Ax} t_n$ where each t_i is called a *system state*. Note that each state is irreducible w.r.t. the equational theory $\mathcal{E}$.

Given a term t, more than one redex may exist and more than one rule may be applied, resulting in several possible sequences of $\rightarrow_{R,E}$ steps. Because rules may not be confluent, these sequences may diverge and reach different states. The transition space of all computations in $\mathcal{R}$ stemming from some initial state t_0 can be represented as a *computation tree* denoted $\mathcal{T}_{\mathcal{R}}(t_0)$. The nodes of this tree are states and the arcs correspond to the application of rules in R.

Finally, we use $labels(E)$ (resp., $labels(R)$) to denote the set of labels collected from all equations (resp., rules), and let $label$ be the projection function that obtains the label ℓ of an equation or rule.

2.2 Functional Modules

The basic constructs of Maude consist of definitions of new operators and sorts, which allow the user to define a custom signature Σ, effectively composing a new ad-hoc, specialized grammar for expressing data and computations as terms. Definitions are led by meaningful keywords and are terminated by a space followed by a dot (`␣.`) as shown in Listing 2.1.

Operators are defined using the following syntax led by the `op` keyword:

$$\texttt{op}\ id\ \texttt{:}\ \tau_1\ \tau_2\ \ldots\ \tau_n\ \texttt{->}\ \tau_{return}\ \texttt{[}\ a\ \texttt{]}\ \texttt{.}$$

where *id* is an identifier, $\{\tau_1, \ldots, \tau_n\} \subseteq (S \cup \{[s]\}_{s \in S})$ is a whitespace-separated sequence of types (sorts or kinds) where τ_i is the type of the i-th input argument, $\tau_{return} \in (S \cup \{[s]\}_{s \in S})$ is the return type, and a is an optional set of operational attributes defining properties such as metadata, native integrations or axioms. The arity of the newly defined operator is automatically inferred and equals, as expected, the cardinality of the sequence $\{\tau_1, \ldots, \tau_n\}$. The identifier *id* may take one or more underscore symbols (_), which have a special meaning and denote conceptual "term placeholders" comparable to anonymous variables, that allow for the definition of infix and mixfix operators, e.g., see line 6 in Listing 2.1. If the attribute set a is empty, the surrounding square brackets may be omitted.

Sorts and their subsort relations can be defined using the `sort` and `subsort` keywords as seen in lines 2 and 3 of Listing 2.4. The type system defined by sorts is entirely nominal and thus sorts are represented by an identifier, e.g., `String`.

Equations and conditional equations can be defined using the `eq` and `ceq` keywords, respectively, followed by their corresponding expected syntax:

$$\texttt{eq [}\ \ell\ \texttt{] :}\ l = r\ \texttt{[}\ a\ \texttt{] .}$$
$$\texttt{ceq [}\ \ell\ \texttt{] :}\ l = r\ \texttt{if}\ C\ \texttt{[}\ a\ \texttt{] .}$$

where $l \in \tau(\Sigma, \mathcal{V})$ and $r \in \tau(\Sigma)$ are terms, ℓ is an optional identifier acting as a label, a is an optional set of equational attributes, and C is an equational condition. An equational may be left unlabelled, and in such case the surrounding square brackets and following colon are omitted as well. Similarly, if no attributes are provided, the corresponding square brackets are not present. The equational condition C is a collection of expressions connected by logical connectives that can be interpreted as a Boolean formula, and unconditional equations have an implicit trivial condition of `true`.

The posed definitions may not appear directly at the top level of Maude source code; they need to be encapsulated in an organizational unit called *module*. Maude allows three different grouping constructs at the top level, called *functional modules*, *system modules* and *strategy modules*[2]. Modules have an identifier and can be hierarchically composed by using a notion of inclusion or extension, comparable to the concept of "importing" of other languages. A Maude program can be seen as an arrangement of modules that form a directed graph of "imports".

Functional modules are logical groupings of definitions of types, operators and equations, describing at most a system executable using equational reduction and constituting a program comparable to the ones defined in conventional functional languages. Functional modules follow this syntax:

$$\texttt{fmod}\ id\ \texttt{is}\ d\ \texttt{endfm}$$

where *id* is an identifier uniquely distinguishing this module, and d is a set of definitions and statements constituting the body of the module, which include import directives, (sub)sort and operator definitions and equations. Note that an equational theory $\mathcal{E}$ is encoded in a Maude program as one or more functional modules. The number of modules has no semantic implication and is entirely a source code organization decision left to the discretion of the programmer.

[2]Full Maude also allows object-oriented modules, which are translated internally to functional or system modules and are not considered in this work.

Apart from the modules that are defined by the user, Maude includes a standard library called the *prelude*, containing a series of predefined modules carrying useful definitions for common domains, e.g., natural numbers (`NAT` module), text (`STRING`), Boolean algebra (`BOOL`), ... A distinguished modules in the prelude, which will be used heavily in our framework, is `META-LEVEL`, which contains Maude definitions about the Maude language itself, allowing for run time reflection.

Given a term t in the encoded equational theory $\mathcal{E} = (\Sigma, E)$, equational reduction of t can be requested to Maude by means of the `reduce` command [15, Section 4.9]:

`reduce in` m `:` t `.`

where m is an optional argument specifying the identifier of a module. Commands, contrary to definitions, are not part of modules and are provided directly to the Maude interpreter, which often happens interactively after the source code has been fully loaded. See Example 1 to examine a sample output of this command.

When using Maude, the user will likely work with terms that may contain variables, which in Maude are typed and are denoted by an identifier, followed by a colon and a type. For instance, `Message:String` is an example of a variable called `Message` and of sort `String`. Specifying the type of a variable each time it appears is not convenient, especially if such variable is used many times through the source code. Thus, Maude provides a variable declaration mechanism using the `var` keyword, which appears inside a module and persistently binds the variable name to a type, allowing the use of the variable name directly without the subsequent colon and type annotation.

EXAMPLE 1

The program in Listing 2.1 illustrates a simple functional module encoded in Maude. This module, called `MY-NAT`, defines natural numbers using the axioms of Peano: a constant for zero, and a unary symbol to denote the successor of a number. Additionally, it also defines the mathematical addition operation for two given natural numbers.

```
fmod MY-NAT is
    sort MyNat .

    op 0 : -> MyNat [ctor] .         --- zero
    op s : MyNat -> MyNat [ctor] .   --- successor
    op _+_ : MyNat MyNat -> MyNat .  --- addition

    vars N M : MyNat .

    eq 0 + N = N .
    eq s(N) + M = s (N + M) .
endfm
```

Listing 2.1: Functional module encoding Peano natural numbers with addition

As seen in Listing 2.1, operators in Maude are introduced using the `op` keyword followed by their symbol and then their type signature. The `_+_` notation indicates + is an infix operator. The `sort` keyword is used to define the new sort `MyNat`. The `vars` keyword is used here as a shorthand for multiple separate `var` definitions to declare that names `N` and `M` are variables of the newly defined sort. Three dashes (`---`) mark a comment and cause the interpreter to ignore the rest of the line.

We can now ask Maude to reduce a term using the `reduce` command during an interactive session with the interpreter:

```
Maude> reduce s(0) + s(0) .
reduce in MY-NAT : s(0) + s(0) .
rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
result MyNat: s(s(0))
```

As expected, given the definitions in Listing 2.1, the term `s(0) + s(0)` is reduced to `s(s(0))`. In order to reach this result, first the equation on line 9 has been applied with substitution $\{N/\texttt{0}, M/\texttt{s(0)}\}$, and immediately after that, the equation on line 8 has removed the leftover zero with substitution $\{N/\texttt{s(0)}\}$.

Equational axioms are attributes encoded in Maude as functions that may be added to any binary operator in Σ and that automatically derive distinguished equations for algebraic laws such as associativity, commutativity, or unity of the symbols they accompany. While axioms may be represented simply as equations, the use of axioms incurs in different, more efficient semantics when it comes to pattern matching and has important performance implications. Whereas using equations to represent the aforementioned algebraic laws may increase the search space significantly or even cause a previously terminating theory to become non-terminating, Maude can leverage efficient algorithms for matching modulo axioms algorithms that normalize terms w.r.t. axioms before equations are applied. The implications of this process, which is completely transparent to the user, become evident in Example 2.

Example 2

Consider the definitions in Listing 2.1, describing natural numbers using Peano notation. The following module (incorrectly) defines the product of natural numbers by importing the previous definitions and then adding the corresponding equations in a new functional module.

```
fmod MY-NAT-PROD is
    protecting MY-NAT . --- import definitions of Example 1

    vars N M : MyNat .

    op _*_ : MyNat MyNat -> MyNat .
    eq 0 * N = 0 .
    eq N * s(M) = N + (N * M) . --- not commutative!
endfm
```

Listing 2.2: Incorrect (non-commutative) product of Peano natural numbers

However, if we try to reduce the following term corresponding to an unevaluated product we will observe Maude returns it untouched, as if it was already a normal form:

```
Maude> reduce s(0) * 0 .
reduce in NAT : s(0) * 0 .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s(0) * 0
```

This occurs because the term is indeed in normal form: pattern matching fails to match the given term with all defined equations. The newly defined product symbol, just like any other operator, strictly follows the order of its arguments and assumes no commutativity. Hence, for the right hand side of the second equation `s(M)` is matched against `0`, which always fails. Let us try to fix this by adding a new equation that encodes commutativity explicitly:

```
eq N * M = M * N .  --- causes non-termination!
```

This is not an adequate solution because it causes non-termination: products reduce over themselves infinitely, causing the program to freeze without ever yielding a result. Note that the left hand side `N * M` matches absolutely all unevaluated products with no exceptions. In order to solve this issue efficiently, Maude provides a built-in method for expressing some selected properties such as commutativity: equational axioms. Consider the program in Listing 2.3, which adds the attribute `comm` to the operator `_*_` in line 6.

```
1 fmod MY-NAT-PROD is
2     protecting MY-NAT . --- import definitions of Example 1
3
4     vars N M : MyNat .
5
6     op _*_ : MyNat MyNat -> MyNat [comm] .
7     eq 0 * N = 0 .
8     eq N * s(M) = N + (N * M) .
9 endfm
```

Listing 2.3: Functional module encoding the product of Peano natural numbers

In this program, the undertaken computation finishes with the expected result:

```
Maude> reduce s(0) * 0 .
reduce in NAT : s(0) * 0 .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: 0
```

Example 3

We may also use equational axioms to quickly and easily define a list of natural numbers. The binary associative operator `_._` allows elements to be chained together as in any semigroup. This operator also has the empty list `nil` as its unity symbol, which allows the pattern matching algorithm to insert or ignore occurrences of `nil` freely when considering the different equations. With the definitions given, an example of concrete list is `1 . 3 . 0`. Similarly, sets and multisets may be also defined by adding and removing equational axioms from the binary constructor.

2.3 System Modules

In Maude, system modules extend functional modules with the capability of containing rewrite rules, allowing the representation of rewrite theories. Because system modules are a superset of functional modules and have no *obligation* of containing rules, they can also be used to represent equational theories. A system module may import a functional

```
fmod MY-NAT-LIST is
    protecting NAT .  --- use natural number definitions from the prelude

    sort NatList .
    subsort Nat < NatList .  --- a single natural is also a (singleton) list

    op nil : -> NatList .  --- constant representing an empty list
    op _._ : NatList NatList -> NatList [ctor assoc id: nil] .  --- list constructor

    --- example: calculate the size of a list
    op size : NatList -> Nat .
    eq size(E:Nat . L:NatList) = s(size(L:NatList)) .
    eq size(nil) = 0 .
endfm
```

Listing 2.4: Functional module encoding a list of natural numbers

module, but not vice versa. Their syntax uses the keywords `mod` and `endm` instead of `fmod` and `endfm`.

Rewrite rules and conditional rewrite rules can be roughly seen as a means to represent the transitions among system states of a (typically concurrent and reactive) non-deterministic system. Similarly to equations, they are respectively encoded using the following syntax:

$$\texttt{rl [} \ell \texttt{] : } l \texttt{ => } r \texttt{ [} a \texttt{] .}$$
$$\texttt{crl [} \ell \texttt{] : } l \texttt{ => } r \texttt{ if } C \texttt{ [} a \texttt{] .}$$

where $l \in \tau(\Sigma, \mathcal{V})$ and $r \in \tau(\Sigma)$ are terms, ℓ is an optional identifier acting as a label, a is an optional set of equational attributes, and C is a rewriting condition.

When using rules, considering *the* canonical normal form of the overall system is nonsensical as the system may take one of many intrinsically different states depending on the particular sequence of choices it has traversed. For instance, a door may be open or closed and it can transition between both states, but none of them may be reduced away to the other. On the other hand, equations are used to represent deterministic computations that achieve the simplification of states towards a normal form. For instance, the term `1 + 1` can be seen as a version of 2 that has yet to be computed, i.e, another representation of the same state. Using both rules and equations together offers great expressive power and empowers the user to model complex scenarios with favorable computational performance properties.

Given an initial state, the search space of possible behaviors (sequence of rewrites) defined by the system can be explored using the `search` command [15, Section 5.4.3], which effectively explores and yields the computation tree $\mathcal{T}_{\mathcal{R}}(t)$. The simplified syntax of this command is as follows:

$$\texttt{search [}b\texttt{] in } m \texttt{ : } t_0 \texttt{ =>}\bullet\ t_n \texttt{ .}$$

where t_0 is the initial term; t_n is a term acting as a pattern used for optionally filtering the reachable states of interest; $\bullet$ is either `1`, `+`, `*` or `!`; b is an optional maximum bound of result states to yield; and m is optionally the name of a module. When using $\bullet$ = `1`, we are asking Maude to return the states reachable with strictly one Maude step that also match the pattern t_n. When using `*` (resp., `+`), we request terms reachable after zero or more (resp., one or more) steps that match the pattern t_n. Finally, when using `!`, we restrict results to normal forms that may not be reduced with more steps, and that also match the pattern t_n. Note that the pattern t_n may be a trivial term formed by one free variable and no constructors, which may be roughly seen as a null filter that vacuously allows all reachable terms. See Example 4.

EXAMPLE 4

Consider the following example in Listing 2.5, modelling a store in a country using a currency with dollars ($) and quarters of dollar (c). Those two monetary units have a conversion ratio of 1$ = 4c. A customer may have any combination of dollars and quarters, and may decide to buy items x and y (with selling prices of 1$ and 3c, respectively) at any moment, in any quantity and combination.

```
mod SHOP is
    sorts Coin Item Marking .
    subsorts Coin Item < Marking .

    op __ : Marking Marking -> Marking [assoc comm id: null] . --- juxtaposition
    op null : -> Marking .
    op $ : -> Coin .
    op c : -> Coin .
    op x : -> Item .
    op y : -> Item .

    rl [buy-x] : M:Marking $     => M:Marking x .
    rl [buy-y] : M:Marking c c c => M:Marking y .

    rl $ => c c c c .
    rl c c c c => $ .
endm
```

Listing 2.5: System module encoding a shop with dollars and quarters

The search space of a term denoting a system state, such as `$ $ c c c`, can be explored using the `search` command, yielding the terms reachable after a sequence of rule applications:

```
Maude> search [4] $ $ c c c =>+ M:Marking .

Solution 1 (state 1)
M:Marking --> $ c c c x

Solution 2 (state 2)
M:Marking --> $ $ y

Solution 3 (state 3)
M:Marking --> $ c c c c c c c

Solution 4 (state 4)
M:Marking --> c c c x x
```

Roughly speaking, note that the free variable `M` of sort `Marking` is here used to include all reachable terms with no filtering whatsoever. The `[4]` fragment is part of the syntax of the `search` command and indicates we wish to limit the response to 4 solutions, halting the exploration process early after finding this number. In this specific case, the unlimited version of this command would have yielded 16 solutions.

Another command, `rewrite`, promptly chooses an arbitrary rewrite sequence of $\mathcal{T}_{\mathcal{R}}(t)$ and yields the resulting term:

$$\texttt{rewrite [}b\texttt{] in } m \texttt{ : } t_0 \texttt{ .}$$

where t_0 is the initial term, b is an optional maximum bound of steps to perform, and m is used optionally to indicate the name of a module. If the bound b is not specified, Maude steps are applied until the normal form $t\downarrow_{R,E}$ is reached, which may result in an infinite computation.

As shown in Examples 1 and 4, software systems can be modeled as rewriting or equational theories which can then be encoded as a Maude program. These programs may be executed, transformed, optimized, analyzed, verified, etc., by using the Maude interpreter and the broad variety of tools that are available in the Maude ecosystem.

2.4 The Maude Strategy Language

Rule-based rewriting can be 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* [15, 22, 23]. This (sub-)language allows the user to control the rewriting process while respecting the *separation of concerns* principle since the rewrite theory is not modified in any way and could be executed according to different strategies. The strategy language is broad and contains many different strategy constructors dedicated to fulfilling different purposes. We consider a restricted, yet non-trivial, version of the strategy language that is sufficient to fulfill the specific needs described and developed in this work.

Definition 1. [22] A strategy (or strategy expression) ζ for a rewrite theory $\mathcal{R}$ is an expression that is built using the following abstract syntax.

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

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.

Strategies guide the program space exploration by specifying the collection of admissible rule applications, thus constraining the computation tree. Given the rewrite theory $\mathcal{R} = (\Sigma, E, R)$, the semantics of a strategy ζ, denoted $[\![\zeta]\!]$, is a function $\tau(\Sigma) \mapsto \mathcal{P}(\tau(\Sigma))$ that maps a term representing a system state $t \in \tau(\Sigma)$ to the set of terms that are reachable using some rules of R, where only the finite number of rule applications deemed valid by said strategy are considered [15, 22, 23]. The semantics for the different constructors of Definition 1 are summarized in Definition 2.

Definition 2. [15] Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory. Let $t \in \tau(\Sigma)$. The strategy operators of Definition 1 have the following semantics:

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

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

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

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

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

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

Roughly speaking, there is an intuitive notion of “success” associated to strategies. When a strategy returns an empty set, we say that strategy *fails*; otherwise, it *succeeds*. The `idle` strategy always succeeds and conceptually does nothing, by returning a singleton containing the same term that was provided initially, unchanged, and not causing any rule application to occur. The `fail` strategy always fails by returning an empty set. A strategy that just contains a rule label ℓ only considers applications with that rule, and thus returns the set of all terms t' such that $t \xrightarrow{r}_{R,E} t'$ [3]. The *choice* operator $\alpha \mid \beta$ returns the union of all terms returned by α and β, conceptually representing that any of the two strategies may be applied concurrently. The *sequential* operator $\alpha ; \beta$ first applies the strategy α to completion, and then sequentially applies β to all terms returned by α. The *conditional* operator $\alpha ? \beta : \gamma$ first applies strategy α and then, if said strategy succeeds, applies β sequentially to its results; otherwise, it applies γ on the initial term. The strategy `all` triggers a step $\rightarrow_{R,E}$ for all applicable rules in R.

Repetition may be expressed in the strategy language using the *closure* constructors $\alpha*$ and $\alpha+$. The former denotes the application of α zero or more times, whereas the latter requires the strategy to be applied at least once. The *negation* strategy constructor $not(\alpha) = \alpha$ `? fail : idle` fails when α succeeds, and succeeds as `idle` when α fails. The *normalisation closure* $\alpha! = \alpha* ; not(\alpha)$ applies a strategy repeatedly until it cannot be applied anymore.

A strategy ζ can be applied to a term t using Maude's `srew` command (**s**trategy **rew**rite), whose syntax is as follows:

$$\texttt{srew [}b\texttt{]}\ t\ \texttt{using}\ \zeta\ .$$

where b is an optional natural number specifying the maximum bound of ζ-controlled Maude steps to perform. If no bound is provided, the strategy will be applied to completion, until no steps can be carried out anymore. This command requests Maude to calculate and print the set of terms $[\![\zeta]\!](t)$.

The given set-based semantics describe the behavior of the strategy language based on its term results. However, it does not describe the explicit steps taken by the interpreter when expanding the computation tree under the control of a strategy [23]. Given a strategy ζ, this notion of computation tree $\mathcal{R}$ w.r.t. ζ can be formalised resorting to the small-step operational semantics described in [22] and [23, Section 4.2]. Such semantics provide a finer description of the intermediate states and present two different strategy step types: *control strategy steps* and *system strategy steps*. Control steps expand the given

[3]Note that this set will be empty (i.e., the strategy will fail) if there are no redexes in t where r can be applied.

tree t into a series of intermediate steps necessary for carrying out the strategy computation, while system steps represent the actual rewrite steps using the rules of R. Finally, the tree is compacted by removing the intermediate steps and control steps, thus yielding a tree whose branches represent rule applications and nodes represent system states in $\mathcal{R}$. The leaves of this tree form the set $[\![\zeta]\!](t)$: each term has been obtained by the application of zero or more rewrite rules (represented by control strategy steps), obeying the control imposed by ζ.

Definition 3. Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and let ζ be a strategy for $\mathcal{R}$. The *strategy-controlled computation tree* $\mathcal{T}^{\zeta}_{\mathcal{R}}(t_0)$ for the initial term t_0, w.r.t. $\mathcal{R}$ and ζ, is such that $\mathcal{T}^{\zeta}_{\mathcal{R}}(t_0) \subseteq \mathcal{T}_{\mathcal{R}}(t_0)$ and its branches represent computations of the form $t_0 \rightarrow_{R,E} t_1 \ldots \rightarrow_{R,E} t_n$ where each t_i is computed by the strategy ζ according to the small-step semantics of [22] and [23, Section 4.2] for $0 \leq i \leq n$.

Given the close relation between $\mathcal{T}^{\zeta}_{\mathcal{R}}(t_0)$ and $\mathcal{T}_{\mathcal{R}}(t_0)$, Definition 3 ensures that t_n is reachable from state t_0 using the strategy ζ whenever $t_n \in [\![\zeta]\!](t_0)$. Intuitively, $\mathcal{T}^{\zeta}_{\mathcal{R}}(t_0)$ constraints the computation tree $\mathcal{T}_{\mathcal{R}}(t_0)$ by pruning the computations that do not satisfy the strategy ζ, hence reducing the search space of $\mathcal{R}$.

In Maude, strategy expressions may be encoded as callable named definitions that work akin to functions, allowing organisation and reuse. These definitions support several features such as optionally being parametric over several input parameters. Named strategy definitions must be typed like a normal operator, and must be defined inside a new kind of module called *strategy module*. Strategy modules act as an overlay over existing Maude programs and may import system and functional modules, but system and functional modules may not import strategy modules.

In Maude, named strategy definitions are actually divided into two different constructs, one specifying the strategy name alongside its signature using the `strat` keyword, similarly to the definition of a new operator using `op`, and a second one using the `sd` keyword to actually map the strategy call to a strategy expression, analogously to an equation definition. Furthermore, named strategies encode complex features and may take arguments, just like non-constant operators [15, Section 10.2]. The simplified syntax for both syntactic constructs is as follows:

$$\texttt{strat}\ \mathit{id}\ \texttt{:}\ \tau_1\ \tau_2\ \ldots\ \tau_n\ \texttt{@}\ \tau_{subject}\ \texttt{.}$$
$$\texttt{sd}\ \zeta\ \texttt{:=}\ \zeta'\ \texttt{.}$$

where *id* is an identifier, $\{\tau_1, \ldots, \tau_n\} \subseteq (S \cup \{[s]\}_{s \in S})$ is a whitespace-separated sequence of types corresponding to the input arguments, $\tau_{subject} \in (S \cup \{[s]\}_{s \in S})$ is the *subject* type, and ζ, ζ' are two strategy expressions. What we call the subject type $\tau_{subject}$ is the type of all terms on which this strategy can be applied, i.e., it is the type such that $[\![\zeta]\!](t)$ is defined for all terms satisfying $t :: \tau_{subject}$.

For our purposes, we do not make use of the capability of receiving arguments and we keep a one-to-one relation defining one `sd` "equation" per each 0-arity `strat` "operator", so we can simplify the syntax even further:

$$\texttt{strat}\ i\ \texttt{:}\ \texttt{@}\ \tau_{subject}\ \texttt{.}$$
$$\texttt{sd}\ i\ \texttt{:=}\ \zeta'\ \texttt{.}$$

Let us illustrate the strategy language by means of the following program, which models the controller of an unmanned space probe orbiting Earth in space, and resembling an artificial satellite. This program is a Maude implementation extending a simple model described in [11, 12], which has been extracted from a series of papers related to several efforts conducted by the European Space Agency to improve mission planning and scheduling of several operations, including the well-known Mars Express mission. We use this program as a case study through the rest of this work. The complete source code is available in the companion file `satellite.maude`[4].

EXAMPLE 5

Let us model in Maude a satellite that floats through space 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, eventually, telematically delivered to mission control when the probe is facing Earth. This device orbits our planet, so 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 values for a collection $\mathcal{X}$ of *state variables*. Each state variable is an instance of a sum type, i.e., each variable represents one piece of information by storing exactly one value at a time from a predetermined and finite selection of values. Specifically, the state variables and their associated values are defined in Listing 2.6 and are summarised in Table 2.1. The family of all possible valid variable values $\mathcal{C} = \{\mathcal{C}_x\}_{x \in \mathcal{X}}$ is split into sets corresponding to the different state variables these values are associated with. The set of allowed values a state variable $x \in \mathcal{X}$ may take is denoted by $\mathcal{C}_x$.

```
sorts PointingMode PointingModeState .
ops earth slewing science comm maintenance : -> PointingMode .
op pm:__ : Time PointingMode -> PointingModeState .

sort GroundVisibility GroundVisibilityState .
ops visible notVisible : -> GroundVisibility .
op gv:__ : Time GroundVisibility -> GroundVisibilityState .

sort InstrumentStatus InstrumentStatusState .
ops idle warmup process turnoff : -> InstrumentStatus .
op is:__ : Time InstrumentStatus -> InstrumentStatusState .

sort State .
op {_,_,_} : PointingModeState GroundVisibilityState InstrumentStatusState -> State .
```

Listing 2.6: Definition of state variables of a space probe and their possible values

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:

$$t = \texttt{\{ pm: 12 earth, gv: 28 notVisible, is: 7 idle \}}$$

[4]This source code is also available for download at https://raw.githubusercontent.com/skneko/strass/master/core/examples/satellite/satellite.maude

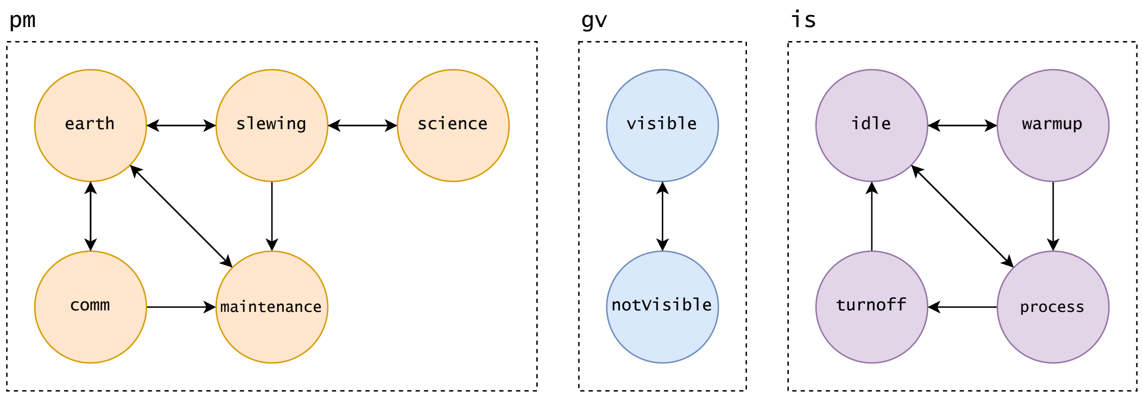
Table 2.1: Summary of state variables of the space probe and their possible values

Var. $x \in \mathcal{X}$	Description of variable	Possible values $\mathcal{C}_x \subset \mathcal{C}$	Description of values
`pm`	**Pointing mode**: specifies the main function being performed by the probe based on its relative rotation w.r.t. Earth.	`earth`	The probe is pointing towards Earth. This position is a requirement for successful communication.
		`slewing`	The probe is now actively rotating. The speed of the movement prevents scienfitic data collection and communication.
		`science`	The probe is aligned with a point of interest and collecting scientific data.
		`comm`	The probe is communicating with mission control on Earth.
		`maintenance`	The probe is executing a self-maintenance procedure. Data collection and communication are not possible.
`gv`	**Ground visibility**: specifies whether the probe can be seen from mission control w.r.t. the orbit of Earth.	`visible`	The probe can be seen from mission control.
		`notVisible`	The probe is occluded by Earth.
`is`	**Instrument status**: indicates if the on-board scientific equipment is currently operative.	`idle`	The equipment is not operating.
		`warmup`	The equipment is preparing to start operating shortly.
		`process`	The equipment is operating.
		`turnoff`	The equipment is ceasing its operation and will turn off.

In this state t, the pointing mode (pm) indicates 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, the on-board instruments (is) have been idling for 7 time units. The value of a state variable $x \in \mathcal{X}$ in a state t is denoted $value_t(x) : \mathcal{X} \mapsto \mathcal{C}_{\mathcal{X}}$ and the accompanying clock annotation is called *duration* and denoted $duration_t(x) : \mathcal{X} \mapsto \mathbb{N}$, e.g., $value_t(\texttt{pm})$ is `earth` and $duration_t(\texttt{pm})$ is 12.

There are two ways in which this system may evolve at every step: on the one hand, the `advance-time` rule may act as a time "tick" by advancing all the clocks simultaneously, and thus reaching a new state t' where the durations would be 13, 29 and 8 while keeping the same values for each state variable unmodified. On the other hand, one of the many other transition rules could mutate the value of a state variable if such a transition is possible. For example, the satellite could become visible from mission control just as it moves through the trajectory of Earth's orbit, and in such a case a new state would be reached where all values and clocks remain untouched except for $value(\texttt{gv}) =$ `visible` and $duration(\texttt{gv}) = 0$. Note that advancing time and mutating state are modeled using different rules and are hence two independent possibilities which may occur independently and intertwined in any combination.

Figure 2.1: Representation of the enabled transitions for state variables of the space probe



However, not all state transitions are admissible: mutating state is subject to conditions, so it is not allowed for certain states to reach others according to the values of their state variables. For instance, variable `is` may not transition from `idle` directly to `turnoff`, but a transition from `idle` to `warmup` does exist. This notion of enabled transition can be defined using a judgement function $\lambda : (\mathcal{X}, \mathcal{C}_{\mathcal{X}}, \mathcal{C}_{\mathcal{X}}) \mapsto \mathbb{B}$, taking a state variable, its current value and the value we are attempting to assign, to finally return a Boolean value[5]. Nevertheless, in our model implementation, this function does not actually exist but enabled transitions are encoded as a series of rules. Enabled transitions are summarized in Fig. 2.1. Furthermore, transitions are also restricted according to their durations, defining an allowed minimum and maximum amount of time the different state variables must hold each specific value, e.g., the probe must be visible between 60 and 100 time units, so it can not change that value before 60 time units have passed and is required to change after 100 time units have passed. The *allowed duration* of a state value is a mapping $\delta : \mathcal{C} \mapsto (\mathbb{N}, \mathbb{N}^{\infty})$ from any value (for any variable) to a pair of time values (m, M) encoding the closed time interval $[m, M]$ allowed to hold that value. Thus, a state t can only transition into a new state t' if and only if the Boolean formula $(\forall x \in \mathcal{X})\ \lambda(x, value_t(x), value_{t'}(x)) \wedge m \leq duration_t(x)$, where $(m, M) = \delta(value_t(x))$,

[5] Keeping an already assigned value is always allowed (modulo duration restrictions), so $(\forall x \in \mathcal{X})\ \lambda(x, c, c) = true$.

is satisfied. In a state t, time may advance only if $duration_t(x) \leq M$ where $(m, M) = \delta(value_t(x))$. The allowed duration function δ is encoded in Maude as shown in Listing 2.7 and is summarized in Table 2.2.

```
op duration : StateVariable -> TimeInterval .
eq duration(earth) = [ 1, +inf ] .
eq duration(slewing) = [ 30, 30 ] .
eq duration(science) = [ 36, 58 ] .
eq duration(comm) = [ 30, 50 ] .
eq duration(maintenance) = [ 90, 90 ] .
eq duration(visible) = [ 60, 100 ] .
eq duration(notVisible) = [ 1, 100 ] .
eq duration(idle) = [ 1, +inf ] .
eq duration(warmup) = [ 5, 5 ] .
eq duration(process) = [ 5, 5 ] .
eq duration(turnoff) = [ 5, 5 ] .

--- ... omitted code ...

vars T Tmin Tmax : Time .

op canContinue : Time TimeInterval -> Bool .
eq canContinue(T, [ Tmin, Tmax ]) = T <= Tmax .
eq canContinue(T, [ Tmin, +inf ]) = true .

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

Listing 2.7: Maude encoding of duration bound restrictions for state values of the space probe

Table 2.2: Duration bound restrictions for state values of the space probe

Var. $x \in \mathcal{X}$	**Value** $c_{\mathcal{X}} \in \mathcal{C}$	**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

Let us now consider the strategy `advance-time +`. Here, `advance-time` is the label of the rule that describes the advancement of time, moving forward all of the clocks in tandem without changing the values of the state variables. Recall that, in the strategy language, a rule label describes its application on the subject term, and + denotes the transitive (but not reflexive) closure. Thus, this strategy conveys the meaning "time must advance one or more times". Let us use the `srew` command to apply this strategy on the considered term t:

```
srew { pm: 12 earth, gv: 28 notVisible, is: 7 idle } using advance-time + .
```

```
Solution 1
result State: { pm: 13 earth, gv: 29 notVisible, is: 8 idle }

--- ... output truncated ...

Solution 71
result State: { pm: 83 earth, gv: 99 notVisible, is: 78 idle }

Solution 72
result State: { pm: 84 earth, gv: 100 notVisible, is: 79 idle }

No more solutions.
```

Notice that the first term is the result of advancing time once, the second term results from advancing time twice, and so on, forming a chain where each solution is a temporal continuation of the previous one. It is perhaps surprising that this command yields a finite set stopping after 72 solutions: the + closure indicates one step *or more* and imposes no upper limit on the number of applications. However, recall the rule `advance-time` is conditional and is subject to the previously stated condition that the maximum duration bound of a value is not to be exceeded. The figures in Table 2.2 reveal that the maximum duration bound for the value `notVisible` (associated to the variable `gv`) is 100, which is exactly the value that appears in the last solution. This is not a coincidence: the rule `advance-time` cannot be applied anymore because the hypothetical solution 73 would violate the constraint $duration_{\text{Sol.73}}(\texttt{gv}) = 101 \leq M$ for $(m, M) = \delta(\texttt{notVisible}) = (1, 100)$. In spite of the fact that there are rules in the program that could apply now and continue this derivation, the considered strategy only allows the `advance-time` rule in exclusivity that now is unable to trigger, and thus solution 72 cannot transition any further.

CHAPTER 3

Safety Enforcement via Programmable Strategies in Maude

In this chapter, we introduce the developed safety enforcement technique by first presenting a new language enabling the specification of the notion of safety for a given rewrite theory and then detailing a novel method which leverages the Maude strategy language to carry out an automatic program correction procedure that enforces the safety constraints expressed by the user.

3.1 The Maude Safety Policy Specification (MSPS) Language

As discussed before, Maude can be used to encode a rewrite theory that models some complex, concurrent, reactive system by describing the transitions between system states as rewrite rules. Programmers and domain experts using Maude may incur in mistakes when defining constructs such as rewrite rules and equations, causing a misalignment between the intended semantics and the actual semantics of the specified artifact, and thus threatening the functional safety of the model and the real software system.

The first step towards ensuring the system safety is being able to express *what* is safety in a certain domain or context, this is, the user needs to be able to unambiguously encode what states of a program are desirable or safe, or dually, undesirable or unsafe, allowing unsafe behaviors to be unmistakably identified and tagged as such (to a certain extent). We call this description the *safety policy* of a system. In order to be able to automate this process, this means the encoding must also be representable in some formal language that can be processed by a machine.

We propose a new formal language, called the *Maude Safety Policy Specification* (MSPS) language, which can convey the safety policy associated to a Maude program. Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, this language allows the user to provide a series of statements using the signature Σ that, together, form a safety policy $\mathcal{A}$ w.r.t. $\mathcal{R}$. A safety policy expressed in MSPS is encoded as text containing one statement per line, according to the grammar in Definition 4.

Definition 4. A policy safety $\mathcal{A}$ for a rewrite theory $\mathcal{R}$ can be expressed in the *Maude Safety Policy Specification* (MSPS) language using the following grammar, given in Extended Backus–Naur (EBNF) notation as described in [18].

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

where *term* is a valid Maude term and *identifier* is a valid Maude identifier.

As denoted by the grammar, each statement in a safety policy $\mathcal{A}$ can be either a *state assertion* or a *path strategy*, which roughly speaking relate to the safety of terms and transitions, respectively.

3.1.1. State assertions

State assertions are used to denote the (un)safety of states. The set of all state assertions, $state(\mathcal{A})$, enables automatic rulings of safety for any ground term $t \in \tau(\Sigma)$ representing a system state of $\mathcal{R}$.

Definition 5. Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, with $\Sigma = \mathcal{D} \uplus \Omega$, a *state assertion* has the form $\Pi \ \# \ \varphi$, where $\Pi \in \tau(\Omega, \mathcal{V})$ is a (possibly non-ground) constructor term and φ is a quantifier-free first-order logic formula.

As seen in Definition 5, state assertions have two parts: first, a *pattern* term Π that selects states according to their shape; and second, a logic formula φ that decides whether the selected state is safe. Intuitively, a state assertion requires that all terms matching the pattern Π must obey the requirement φ. The pattern matching algorithm creates a variable context when matching a given state with the pattern, allowing the formula φ to reason about the contents of such a term. The pattern Π is a possibly non-ground term. The formula φ can be constructed using the usual Boolean operators as well as user-defined operators. For the sake of the user's convenience, we allow the addition of new operators and equations in an independent step, so that they can be used as custom predicates in the property φ even if they are not originally part of Σ, as we show in Section 4.6.

Operationally, state assertions check the logic invariants $\Pi \ \# \ \varphi$ in any subterm and not only at the top of the considered system state, empowering the user to express complex properties with notable expressive power. For instance, if N and M are two variables of sort `Nat`, the state assertion `N . M # N > M` holds for the program defined in Example 3 and indicates that, for any pair of natural numbers inside the same list, the first number must be greater than the second one. For example, this assertion works for the list `3 . 2 . 1`, in spite of the fact it contains more than two numbers: it causes the condition to be checked repeatedly, pairwise. This, in turn, states that all lists of natural numbers must be arranged in strictly decreasing order.

Definition 6. Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and let t be a system state in $\mathcal{R}$. We say a state assertion *holds* in t, denoted $\Pi \ \# \ \varphi \models t$, if for every position $w \in Pos(t)$ and for every substitution σ, $t_{|w} =_{Ax} \Pi\sigma$ implies that $\varphi\sigma$ is satisfied. If no such position w exists, the assertion vacuously holds. We also say t is *safe* w.r.t. $\Pi \ \# \ \varphi$.

We know how to determine if an individual system state is safe w.r.t. a specific state assertion as posed in Definition 6, and this notion may also be extended for collections of state assertions. Consider the state assertions $\Pi \ \# \ \varphi \in state(\mathcal{A})$ contained in some safety policy $\mathcal{A}$ of a rewrite theory $\mathcal{R}$, and a term t representing a system state of $\mathcal{R}$. This term t may match several state assertions at once, i.e., there may be a subset of state assertions $\{\Pi_0 \# \varphi_0, \ldots, \Pi_n \# \varphi_n\} \subseteq \mathit{states}(\mathcal{A})$ such that their patterns simultaneously match the term with a series of corresponding substitutions σ_i, $t =_{Ax} \Pi_i\sigma_i$, for $0 \leq i \leq n$. In this case, we say the state assertions in such a subset *overlap* on t. Intuitively, the expected semantics in such a situation is that all the overlapping state assertions must hold on t simultaneously, as the user has expressed a list of conditions for safety and leaving even one unsatisfied could pose a great threat for the system safety. There is, therefore, an implicit conjunction joining all state assertions in a safety policy, i.e., they can be chained using a logical AND.

For instance, remember the state assertion `N:Nat . M:Nat # N:Nat > M:Nat` for the list of natural numbers defined in Example 3. Let us say that the considered lists contain quantities that will be used as the denominator of a division in some algorithm, and thus we consider `0` to be invalid as an element. In such case, it could be argued that the list `4 . 3 . 1 . 0` is undesired, and that running the algorithm on it could eventually produce a division by zero error. We can add a new state assertion in order to forbid this value from appearing inside lists:

```
L:NatList . 0 . L':NatList # false
```

However, if we only check the first assertion, we could reach the false and potentially dangerous conclusion that this list is admissible in the considered domain, as all pairs satisfy the monotonic order imposed by the guard $N > M$ despite it violates the second assertion.

Definition 7. Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory. We say a safety policy $\mathcal{A}$ *holds* in a term t representing a system state in $\mathcal{R}$, denoted $\mathcal{A} \models t$, if and only if all system assertions contained in $\mathcal{A}$ hold in t, i.e. $(\forall (\Pi \# \varphi) \in \mathit{state}(\mathcal{A}))\ \Pi \# \varphi \models t$. We also say t is *safe* w.r.t. $\mathcal{A}$.

3.1.2. Path strategies

While safety may be expressed from the point of view of the suitability of states w.r.t. the expected semantics of a given domain, there exists another notion involving safety, which relates to the way states evolve into other states. Sometimes, even if two states t and t' are safe, what may be unsafe is the way the overall transition has taken place, as the domain constraints may require some intermediate states to be traversed in a certain admissible way. For example, imagine t represents a container of water that is full with a considerable but still safe quantity of water. The state t' represents the empty container, which is also a safe state. However, the one-step transition relating both denotes the container has lost all its volume in one time unit, which is either impossible or dangerous for the pipes draining such container. An approach purely based on state assertions is unable to reason about the admissible transitions between safe states, and is hence unable to express that such an scenario may be undesirable. This is the reason why a second kind of statement is considered in MSPS: path strategies.

As introduced before, Maude has a mechanism for specifying how terms evolve by controlling and constraining possible rule applications by means of the strategy language. Such control may be used to describe the subset of transitions between states that are safe, and thus adding the capability of attaching path strategies to a MSPS specification is helpful for encoding the safety constraints of a system in one self-contained, complete safety policy $\mathcal{A}$ for the rewrite theory $\mathcal{R}$. Path strategy statements enable the user to attach a Maude strategy to the safety policy while relating it to a sort in $\mathcal{R}$, and 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.

Definition 8. Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and let S be the set of sorts in $\mathcal{R}$. A *path strategy* P_s, with $s \in S$, imposes a strategy ζ to all terms of the sort s. If said sort is not relevant, we simply use P. We denote the set of all the path strategies that can be built in $\mathcal{R}$ by $PStr(\mathcal{R})$.

Path strategies are notably useful for systems that model actions that repeat predictably over time, in an operation cycle that is inherent to its domain. This is common in *real-time systems*. For instance, consider again a computer-controlled container of water. A collection of rules models the possible actions that may exist between the system and its environment, e.g., a human agent pushing a button to request the system to empty the container. However, a time advancement rule must be executed repeatedly and regularly to track the inner volume of the container as it changes per unit of time, regardless of the inbound orders. A scenario that ignores the advancement of time, where orders keep being processed without recalculating and updating the volume, should be unfeasible. This controlled rule interleaving may be enforced using a path strategy.

The syntax of path strategies is as follows:

$$\texttt{path for } s \texttt{ : } \zeta$$

where s is the sort this path strategy P_s is being associated to, and ζ is a strategy expression.

For the sake of the user's convenience, strategy expressions ζ in path strategies have been extended with a custom operator that is especially useful when expressing safety policies: the *all except* L operator, denoted `all-(`L`)` in MSPS, where L is 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. See Example 6.

Definition 9. Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory. Let $t \in \tau(\Sigma)$. Let $L \subseteq labels(R)$. The new operator `all-(`L`)` is defined as a macro that expands to Maude's strategy language as follows:

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

Example 6

Consider again the satellite controller of Example 5. We can use the MSPS language to specify a safety policy $\mathcal{A}_{\texttt{SATELLITE}}$ encoding representative domain constraints as shown in Listing 3.1. Each line contains a different declaration, where the first five lines contain state assertions Π # φ and the last line contains a path strategy `path for` s `:` ζ.

Line by line, the safety policy at hand imposes the following constraints on the considered domain:

Line 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

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

Listing 3.1: Safety policy for the space probe controller

than what is actually required for a certain scenarios, especially when importing already existing modules for the purpose of code reuse. We simulate such situation in this example: the auxiliary module TIME contains some generic, program independent definitions for the concept of time, using (potentially negative) integers as orderable logic time values or *time stamps*. In this case, however, we only wish to use the domain of naturals, being zero the first valid time instant possible. We thus use this assertion to match any time value and explicitly indicate it must be equal to or greater than 0.

Line 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, to the enforce that the visibility must be adequate as well in such moment.

Line 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 where 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.

Line 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.

Line 5. **During scientific data collection operations, the on-board instruments will not idle for longer than double of the maximum processing time allowed.**
The time dedicated to maintenance operations is valuable and hence it is important the on-boards instruments remain operative and are 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 as encoded by $(m, M) = \delta(\texttt{process})$. 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. This auxiliary operator is not part of the original program and is hence provided as part of the auxiliary predicates of the safety policy, shown in Listing 3.2.

Line 6. **State variable change transitions and time advancement transitions must be intertwined.**
As explained in Example 5, 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 form: (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 interlaced time advancement is not allowed.

In the contained strategy expression

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

we use `all-(advance-time)` to refer to all rules in the program bar 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 `(all-(advance-time) | idle)` indicates that *at most* one of such state change rules should be applied, where `idle` represents the absence of a 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.

```
var Tmin Tmax : Time .

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

Listing 3.2: Auxiliary safety policy predicates for the space probe controller

3.2 Computing Safe Maude Programs

We have previously explained that the MSPS language allows the user to express a safety policy for a given program, describing the safety of states using state assertions and controlling how computations evolve through the use of path strategies. In this section we introduce an automatic correction technique that allows the automatic enforcement of a given safety policy for a given program. Given a program $\mathcal{R}$ and a safety policy $\mathcal{A}$, the technique transforms $\mathcal{R}$ into a new *safe* program $\mathcal{R}'$ whose computations satisfy $\mathcal{A}$. This new program uses the Maude strategy language to identify unsafe situations and drive the control away from them, effectively restricting computations to the ones deemed safe by $\mathcal{A}$.

Note that the procedure for straightening a program is an operation that must only be run once per each pair of program $\mathcal{R}$ and safety policy $\mathcal{A}$, in order to obtain the safe version $\mathcal{R}'$. Once this result has been obtained, it may be stored and then executed a potentially infinite number of times. In this program lifecycle, we say that operations and properties that apply to this one-time reparation procedure occur at *transformation time*. In contrast, we say that everything related to the later executions of $\mathcal{R}'$ occur at *run time*.

In order to enforce the allowed semantics of safety policies as shown in Section 3.1, a strategy module is generated and added to the original program encoding $\mathcal{R}$, acting as an overlay that does not actually modify the semantics (and thus properties) of $\mathcal{R}$. In other words, the presented technique attains automatic run time safety by extending the original program with new, additional source code, and without modifying said original code. Recall the concept of computation tree $\mathcal{T}_{\mathcal{R}}(t)$, containing the program search space of states reachable from an initial state t, alongside the corresponding sequences of applied rewrite rules. Roughly, the new attached strategy module drives the program constraining its computations to a new computation tree $\mathcal{T}'_{\mathcal{R}}(t) \subseteq \mathcal{T}_{\mathcal{R}}(t)$ that only comprises safe states and transitions.

This strategy module is generated in three steps. First, state assertions in the given safety policy are encoded as strategy expressions in Maude's strategy language. Secondly, path strategies are transformed and integrated with state assertions. Finally, the generated strategy expressions are wrapped in named strategy definitions and then arranged to construct the final strategy module that will be overimposed to the program.

3.2.1. Encoding State Assertions as Strategies

The main objective of state assertions is removing unsafe states from a given computation tree, which intuitively can be seen as limiting, constraining or *filtering* such tree. State assertions in a safety policy are encoded as Maude strategy expressions that can determine whether a term is safe, i.e., if the encoded state assertion holds in a term. Because these strategies are used to prune unsafe states from the tree, we call them *state filters*.

Definition 10. Let $\Pi \,\#\, \varphi$ be a state assertion in $\mathcal{A}$. A *state filter* for $\Pi \,\#\, \varphi$, denoted $F_{\Pi \,\#\, \varphi}$, is a strategy defined as `not(amatch` Π `s.t.` $not(\varphi)$`)`, where *not* represents the Boolean negation[1].

Proposition 1. Let $\mathcal{R} = (\Sigma, E, \mathcal{R})$ be a rewrite theory. Given a state assertion $a \in \mathcal{A}$, and a term $t \in \tau(\Sigma)$:

$$[\![F_a]\!](t) = \begin{cases} \varnothing & \text{if } a \not\models t \\ \{t\} & \text{otherwise} \end{cases}$$

Proof. Immediate given the semantics of the strategy operators in Definition 2. □

EXAMPLE 7

Consider the safety policy defined in Example 6, which defines safety for the controller of a space probe. The following example shows the codification of the state assertion shown in line 3 of Listing 3.1 as a filter strategy:

```
{pm: Tpm:Time comm, gv: Tgv:Time GV:GroundVisibility, is: Tis:Time IS:InstrumentStatus} #
                    GV:GroundVisibility == visible
```

[1]This Boolean operator is not to be confused with the strategy constructor `not(`α`)`, where α is a strategy. Here, *not* is applied to a Boolean formula and has the traditional Boolean algebra semantics.

$$\Downarrow$$

```
not(amatch {pm: Tpm:Time comm, gv: Tgv:Time GV:GroundVisibility, is: Tis:Time IS:InstrumentStatus}
                                s.t.  GV:GroundVisibility == visible = false)
```

Given a state assertion $a \in state(\mathcal{A})$ of a safety policy $\mathcal{A}$ for $\mathcal{R}$, and a term t representing a system state in $\mathcal{R}$, its corresponding state filter strategy F_a succeeds if said term t is safe w.r.t. $\mathcal{A}$, by returning a singleton set with the same term unchanged. However, if the term is instead deemed unsafe, the strategy fails by returning the empty set. Intuitively, a state filter strategy is equivalent to the `idle` strategy constructor when $a \models t$, and equivalent to `fail` when $a \not\models t$.

This specific behavior makes it especially convenient for pruning: imagine we have applied a rule and discovered new leaves to be added to the computation tree currently under exploration. By applying a filter in all leaves and then flattening each returned set as new sibling nodes, we will keep states that are safe (as they are returned in a singleton set) and remove states that are unsafe (as the empty set effectively removes a subtree). While this pruning operation could be ran on the entire computation tree after it has been explored to completion, it is generally more efficient to prune iteratively in small steps in order to avoid spending time exploring potentially big derivations stemming from unsafe states, only to remove these subtrees with all their children nodes later.

Definition 11. Let $\{a_1, \ldots, a_n\} \in state(\mathcal{A})$ be the state assertions of a safety policy. The *lifted* state filter $F_{\{a_1,\ldots,a_n\}}$ for $\{a_1, \ldots, a_n\}$ is defined as F_{a_1} ; ... ; F_{a_n}. For simplicity, $F_{state(\mathcal{A})}$ is denoted by $F_{\mathcal{A}}$.

As stated before, when the safety of a state is to be judged w.r.t. to *several* state assertions, an implicit conjunction exists: a safety policy holds for a state if all state assertions hold for it. The lifted state filter formally encodes this notion, by naturally extending state filters to sets of state assertions. This is implemented using the sequential strategy constructor α ; β because its pipeline semantics cause it to return early as soon as any of the filters detects unsafety. Each filter in the sequence is applied to the results of the previous one, and as soon as one of the filters fail with an empty set, then the remaining filters trivially fail too as they are applied to zero terms. In contrast, if all filters succeed they will pass a singleton set to each other until the input term is yielded unchanged at the end of the filter pipeline. Thus, this composition of state filters acts as a filter itself and preserves the original semantics of conditionally acting as `idle` or `fail`.

Proposition 2. Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory. Given a safety policy $\mathcal{A}$, and a term $t \in \tau(\Sigma)$:

$$[\![F_{\mathcal{A}}]\!](t) = \begin{cases} \emptyset & \text{if } \exists a \in \mathcal{A} \text{ s.t. } a \not\models t \\ \{t\} & \text{otherwise} \end{cases}$$

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

$n = 0$. $state(\mathcal{A})$ is the empty set. Hence, the proposition is vacuously true.

$n > 0$. We have $state(\mathcal{A}) = \{a_1, \ldots, a_n\}$. By Definition 11, $F_{\mathcal{A}} = F_{a_1}, \ldots, F_{a_n}$. Let $t \in \tau(\Sigma)$ be a term. Two cases can be distinguished:

Case *i*. All the state assertions $a_1, \ldots, a_n$ hold in t. In that case, $a_1, \ldots, a_{n-1}$ also holds in t. Hence, by inductive hypothesis $[\![F_{\{a_1,\ldots,a_{n-1}\}}]\!](t) = \{t\}$. Also, by

Proposition 1, $[\![F_{\{a_n\}}]\!](t) = \{t\}$ because a_n holds in t. Then

$$\begin{aligned}
[\![F_{\mathcal{A}}]\!](t) &= \bigcup_{t' \in [\![F_{\{a_1,\ldots,a_{n-1}\}}]\!](t)} [\![F_{\{a_n\}}]\!](t') && \text{(by Definition 2)}\\
&= [\![F_{\{a_n\}}]\!](t) && \text{(by inductive hypothesis)}\\
&= \{t\} && \text{(by Proposition 1)}
\end{aligned}$$

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 $[\![F_{\{a_1,\ldots,a_n\}}]\!](t) = \emptyset$. Hence,

$$\begin{aligned}
[\![F_{\mathcal{A}}]\!](t) &= \bigcup_{t' \in [\![F_{\{a_1,\ldots,a_{n-1}\}}]\!](t)} [\![F_{\{a_n\}}]\!](t') && \text{(by Definition 2)}\\
&= \emptyset && \text{(by inductive hypothesis)}
\end{aligned}$$

If $a = a_n$, then we have

$$\begin{aligned}
[\![F_{\mathcal{A}}]\!](t) &= \bigcup_{t' \in [\![F_{\{a_1,\ldots,a_{n-1}\}}]\!](t)} [\![F_{\{a_n\}}]\!](t') && \text{(by Definition 2)}\\
&= [\![F_{\{a_n\}}]\!](t) && \text{(by inductive hypothesis)}\\
&= \emptyset && \text{(by Proposition 1)}
\end{aligned}$$

□

Corollary 1. Let $\mathcal{R} = (\Sigma, E, \mathcal{R})$ be a rewrite theory. Let $\mathcal{A}$ be a safety policy for $\mathcal{R}$. Let $t \in \tau(\Sigma)$ be a term representing a system state of $\mathcal{R}$. Then, $[\![F_{\mathcal{A}}]\!](t) = \{t\}$ if an only if $\mathcal{A} \models t$.

Proof. Direct consequence of Proposition 2 and the definition of safe state w.r.t. a set of state assertions. □

Using a lifted state filter strategy $F_{\mathcal{A}}$ for the whole safety policy $\mathcal{A}$, the notion of state safety has been completely covered from the point of view of one node in the computation tree. However, it is still necessary to define how a computation tree is unfolded and pruned using $F_{\mathcal{A}}$.

In Maude, the user may explore the computation tree parting for an initial term t, $\mathcal{T}_{\mathcal{R}}(t)$, by using the `search` command. As introduced in Section 2.3, this command applies as many rules as possible and then normalises the reached terms in order to recursively generate the nodes of the tree. The notion of applying all possible rules to a term is encoded by the trivial strategy `all`. And it is not only needed to apply all rules, but to apply rules zero or more times (where the application of zero rules yields the root of the tree), so it can be deduced that the strategy $\boxed{\texttt{all *}}$ is conceptually equivalent to using the `search` command described in Section 2.3, when the command is configured to show all leaves of the tree, using `=>*`. In fact, it is possible to empirically prove that both yield exactly[2] the same computation tree comprising the entire search space of $\mathcal{R}$.

Consider the strategy $\boxed{\texttt{(all ; } F_{\mathcal{A}}\texttt{) *}}$. This strategy unfolds the computation tree as described before but with an important difference: it prunes unsafe leaves iteratively. After the application of all possible rules has caused the discovery of all the possible child terms of a node, the children which are not deemed safe w.r.t. the safety policy are simply

[2]Ignoring the order between sibling nodes, which is an irrelevant implementation detail that does not affect the notion of structural equality between computation trees.

removed. If the definition is examined closely, it can be observed that, at each iteration, `all` comes before $F_{\mathcal{A}}$, so rule expansions come first, and then the pruning process comes afterwards. The resulting tree is just as expected and described before, a (potentially) smaller version with unsafe states removed, where subtrees stemming from unsafe states are never explored.

There is still one detail that needs to be fixed: what does the tree look like if the initial term is unsafe? In the first iteration, the initial term will be first expanded, and then children will be removed. This is not an adequate behavior: in the degenerate case where the initial term given by the user is already unsafe, the considered strategy will yield safe states that are reachable parting from that unsafe situation. Intuitively, we should reject any possible computation and return an empty tree, explicitly denoting the safety policy does not hold in such inadmissible initial conditions. This can be achieved by running an additional filtering phase on the initial term, only once at the beginning of the process: $\boxed{F_{\mathcal{A}}\ \texttt{;}\ \texttt{(all ;}\ F_{\mathcal{A}}\texttt{)*}}$. Note that the first call to $F_{\mathcal{A}}$ is not affected by the closure $*$. We say this strategy is the *safe* version of $\boxed{\texttt{all *}}$ w.r.t. $\mathcal{A}$.

3.2.2. Integrating Path Strategies and State Assertions

The definitions presented beforehand, alone, completely cover safety w.r.t. states: given a term t, $\mathcal{T}_{\mathcal{R}}^{(F_{\mathcal{A}}\ \texttt{;}\ \texttt{(all ;}\ F_{\mathcal{A}}\texttt{) *})}(t)$ is a state-wise safe computation tree for t w.r.t. $\mathcal{A}$. There is, however, another important aspect of safety covered by the semantics of MSPS: safety w.r.t. transitions. Path strategies can be used to restrict the way in which computations evolve for terms of certain sorts. If a path strategy P_s is defined for some sort $s \in S$, and the initial given term t is of that sort, $t :: s$, then the unfolding strategy that must build the tree of safe computations is not $\boxed{\texttt{all *}}$, but the path strategy P_s defined by the user.

Definition 12. Let $\mathcal{R} = (\Sigma, E, \mathcal{R})$ be a rewrite theory and let $\mathcal{A}$ be a safety policy for $\mathcal{R}$. Let $P, P', P'' \in PStr(\mathcal{R})$ be path strategies in $\mathcal{A}$. The *auxiliary safe transformation* of the path strategy P, $safe'(P)$, is defined as follows:

$$safe'(P) = \begin{cases} \ell\ ;F_{\mathcal{A}} & \text{if } P := \ell\ \wedge\ \ell \in labels(\mathcal{R}) \\ \texttt{all}\ ;F_{\mathcal{A}} & \text{if } P := \texttt{all} \\ safe^{*}(P')^{\bullet} & \text{if } P := (P')^{\bullet},\quad \text{with } \bullet \in \{*, +, !\} \\ safe^{*}(P') \circ safe^{*}(P'') & \text{if } P := P' \circ P'',\quad \text{with } \circ \in \{;\ ,\ |\} \end{cases}$$

Similarly to the natural transformation described for the trivial case $\boxed{\texttt{all *}}$, a safe transformation is defined as a helper function (Definition 12) that intertwines filtering phases $F_{\mathcal{A}}$ in the path strategy in order to enforce state-wise safety. Then, a one-time initial filtering phase is inserted at the head of this intermediate result (Definition 13) to handle the degenerate case of the initial state being already unsafe, finally yielding the completed safe version of the path strategy.

Note that the function defined in Definition 12 is called auxiliary because its result is not the safe version of the provided path strategy P, but is used as intermediate data for Definition 13. This transformation intuitively adds filtering phases $F_{\mathcal{A}}$ in the points that follow rule applications, as rules cause states to transition and change, and hence may introduce unsafety w.r.t. states.

Definition 13. Let $\mathcal{A}$ be a safety policy and let P be a path strategy. A *safe path strategy* w.r.t. $\mathcal{A}$, denoted $safe(P)$, is defined as $F_{\mathcal{A}}\ \ ;\ \ safe'(P)$.

It can be observed that $\boxed{\texttt{all *}}$ is a special case of path strategy, and that the transformation formulated before for this strategy is fully generalised by Definition 13.

Proposition 3. Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and let $\mathcal{A}$ be a safety policy. Let $safe(P_s)$, where $s \in S$ is part of Σ, be a safe path strategy for $\mathcal{A}$. For every computation $t_0 \rightarrow_{R,E} \ldots \rightarrow_{R,E} t_n$ in $\mathcal{T}_{\mathcal{R}}^{safe(P_s)}(t_0)$ it is true that:

i. It is also a computation in $\mathcal{T}_{\mathcal{R}}(t)$;

ii. $t_n \in [\![P]\!](t_0)$;

iii. $\mathcal{A} \models t_i$ for all $0 \leq i \leq n$; and

iv. $t_i :: s$ for all $0 \leq i \leq n$.

Proof. Claim (i) trivially holds since $\mathcal{T}_{\mathcal{R}}^{safe(P)}(t)$ includes a subset of the computations of $\mathcal{T}_{\mathcal{R}}(t)$ as per Definition 3. Claim (iv) is a direct consequence of the semantics of Maude regarding rewriting [15, Chapter 5], as rule applications parting from a term t s.t. $ls(t) = s \in S$ can only result in terms $\{t'\}$ where all $t' :: s$. Let us prove the rest of claims:

iii. Let $\mathcal{C} = (t_0 \rightarrow_{R,E} \ldots \rightarrow_{R,E} t_n)$ be a computation in $\mathcal{T}_{\mathcal{R}}^{safe(P)}(t_0)$, where $safe(P) = F_{\mathcal{A}}\ ;\ safe'(P)$. Then, by Definition 3, we have:

$$t_n \in [\![safe(P)]\!](t_0) = [\![F_{\mathcal{A}}\ ;\ safe'(P)]\!](t_0)$$

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 2, $[\![F_{\mathcal{A}}]\!](t_i) = \emptyset$. We distinguish two cases: t_i is the initial state t_0, and t_i is any state in $\{t_1, \ldots, t_n\}$.

$t_i = t_0$. In this case, $[\![F_{\mathcal{A}}]\!](t_0) = \emptyset$; hence, by the definition of the sequential operator ; in Definition 2:

$$[\![safe(P)]\!](t_0) = [\![F_{\mathcal{A}}\ ;\ safe'(P)]\!](t_0) = \bigcup_{t' \in \emptyset} [\![safe'(P)]\!](t') = \emptyset$$

which leads to a contradiction since we assumed $t_n \in [\![safe(P)]\!](t_0)$.

$t_i \in \{t_1, \ldots, t_n\}$. 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 these operators in $safe(P)$ is followed by an application of the state filter $F_{\mathcal{A}}$, that is, $\boxed{\ell\ ;\ F_{\mathcal{A}}}$ and $\boxed{\texttt{all}\ ;\ F_{\mathcal{A}}}$. Since t is not safe w.r.t. $\mathcal{A}$, we have

$$[\![\ell\ ;\ F_{\mathcal{A}}]\!](t_{i-1}) = \emptyset$$
$$[\![\texttt{all}\ ;\ F_{\mathcal{A}}]\!](t_{i-1}) = \emptyset$$

Hence, t_i is not reachable from t_{i-1}, which leads to a contradiction as we assumed $t_{i-1} \rightarrow_{R,E} t_i$.

ii. Let $\mathcal{C} = (t_0 \rightarrow_{R,E} \ldots \rightarrow_{R,E} t_n)$ be a computation in $\mathcal{T}_{\mathcal{R}}^{safe(P)}(t_0)$. Thus, $t_n \in [\![safe(P)]\!](t_0)$. By claim (iii), we know that each t_i in $\mathcal{C}$ are safe w.r.t. $\mathcal{A}$. Hence, by Definition 11, $[\![F_{\mathcal{A}}]\!](t_i) = \{t_i\}$, for $i = 1, \ldots, n$ which directly implies that

$$[\![\ell\ ;\ F_{\mathcal{A}}]\!](t_i) = [\![\ell]\!](t_i) \tag{3.1}$$
$$[\![\texttt{all}\ ;\ F_{\mathcal{A}}]\!](t_i) = [\![\texttt{all}]\!](t_i) \tag{3.2}$$

By Eqs. (3.1) and (3.2) and Definition 12, it is straightforward to show that $safe'(P) = P$ by trivial structural induction on P. Finally,

$$\begin{aligned} t_n \in [\![safe(P)]\!](t_0) &= [\![F_{\mathcal{A}}\ ;\ safe'(P)]\!](t_0) && \text{(by Definition 13)} \\ &= [\![safe'(P)]\!](t_0) && \text{(by } [\![F_{\mathcal{A}}]\!](t_i) = \{t_i\}) \\ &= [\![P]\!](t_0) && \text{(by } safe'(P) = P) \end{aligned}$$

□

Given a rewrite theory $\mathcal{R}$ and a safety policy $\mathcal{A}$, safe path strategies combine the notions of safety w.r.t. states and safety w.r.t. transitions and are able to build a computation tree $\mathcal{T}_{\mathcal{R}}^{safe(P)}(t)$ that is simultaneously safe from both points of view w.r.t. the policy encoded by the user in $\mathcal{A}$. All computations that were safe before are still included in this new tree, whereas computations including at least one unsafe state in any step (regardless of whether it is initial, final or intermediate) are entirely discarded. No new computations are introduced w.r.t. the original program. Roughly speaking, the apparent result is that the program has been automatically "fixed" as it now behaves as if it was programmed to never exhibit unsafety from the beginning. In this sense, $[\![safe(P)]\!](t)$ can be roughly seen as a safe equivalent to Maude's `search` command when the target pattern is a free variable as seen in Example 8.

3.2.3. Arranging the Strategy Definitions

The previous sections provide a formal framework for enforcing safety using Maude's strategy language. This section aims to provide a bridge between this formal description of the technique and its actual implementation, which generates a strategy module that overlays the considered Maude program to be repaired. The generated strategy module contains a hierarchy of families of named strategy definitions where strategies higher in the hierarchy use the strategy definitions beneath them.

Let $\mathcal{R}$ be a rewrite theory and let $\mathcal{A}$ be a safety policy for $\mathcal{R}$. Let $\mathcal{T}'_{\mathcal{R}}(t) \subseteq \mathcal{T}_{\mathcal{R}}(t)$ be a computation tree constrained to only safe states (e.g., the tree $\mathcal{T}_{\mathcal{R}}^{safe(P)}(t)$ for a path strategy P in $\mathcal{A}$). From top to bottom, this conceptual hierarchy is as follows:

1. **Global strategy**: encodes how to unfold the safe computation tree $\mathcal{T}'_{\mathcal{R}}(t)$, driving the control to perform a search of new states and overall making computations evolve. There is one global strategy that is used in a single computation stemming from the initial state.

2. **Local strategy**: encodes how to make one step in the computation tree $\mathcal{T}'_{\mathcal{R}}(t)$ by pruning unsafe states. There is also one local strategy in a given computation stemming from t.

3. **State filter strategies**: the state assertions in $state(\mathcal{A})$ encoded as Maude strategies. These strategies allow judgements regarding the safety of terms to be performed. There is one state filter strategy per each state assertion in $state(\mathcal{A})$.

Note that the description above corresponds to *one computation tree* $\mathcal{T}'_{\mathcal{R}}(t)$ and do not correlate to the output produced by the implementation presented later, which must contain enough definitions to support all possible computation trees, and hence contains *several* global and local strategies. This set of definitions is obviously finite, and every time a safe computation tree $\mathcal{T}'_{\mathcal{R}}(t)$ is to be explored, one global and one local strategy must be selected.

Linking this logical view with the definitions presented before, when unfolding one safe computation tree $\mathcal{T}'_{\mathcal{R}}(t_0)$ for an initial term t_0, the global strategy is the strategy that controls how computations evolve overall, i.e., a safe path strategy $safe(P_s)$. Thus, the safe computation tree is the computation tree generated by the corresponding path strategy, $\mathcal{T}'_{\mathcal{R}}(t_0) \equiv \mathcal{T}_{\mathcal{R}}^{safe(P_s)}(t_0)$, by definition. When given the initial term t_0, the selected global strategy is $safe(P_s)$ for the corresponding P_s whose sort s satisfies $t :: s$. Because one safety policy can only contain one path strategy P_s per each sort $s \in S$, and a term has only one type, this selection is unambiguous. As an added precaution, the correction of this selection is enforced by Maude, whose type checker will emit an error if using a safe strategy $safe(P_u)$, $u \in S$, for t_0 when $t_0 \not: u$.

In in Example 8 we have compared the commands `search` and `srew` as the unsafe and safe counterparts of the same conceptual operation, respectively. As seen in the `search` command, the effective exploration of a program's search state may require the user to specify and limit the number of desired steps between the initial term and the resulting reachable terms, and this command satisfies such a need by offering the symbols `1`, `*`, `+` and `!` with the semantics described in [15, Section 5.4.3]. The `srew` command, however, has no such notion, and the problem space exploration must be controlled using closures present *inside* the strategy provided to said command. For instance, to explore states reachable using the path strategy P with zero or more steps, such strategy body should be wrapped with the `*` closure, $P \equiv \zeta*$, so that the safe strategy $safe(P)$ then equals $\boxed{F_{\mathcal{A}}\ ;\ (safe'(P))\ *}$. However, this causes the path strategy definition to be directly coupled to this tree unfolding decision. If we later desire to explore only normal forms w.r.t. P, we must re-define P as P' to use the `!` closure, $P' \equiv \zeta!$, so that after the safe transformation we obtain $\boxed{F_{\mathcal{A}}\ ;\ (safe'(P'))\ !}$. This is not convenient, as changing the safety policy requires running the transformation procedure again, for what is basically a run time decision. We thus provide a mechanism solving this issue, enhancing the usability of the generated strategies and taking advantage of the existing acquaintance with the `search` command.

Definition 14. The *search shorthands* of a safe path strategy $safe(P)$ are a family of strategy transformations provided for convenience and defined as follows:

$$safe^{\bullet}(P) = F_{\mathcal{A}}\ ;\ (\ safe'(P)\ )\ \bullet$$

where $\bullet \in \{*, +, !\}$.

Comparing with the flexibility of the `search` command, we encourage the user to define path strategies in a way that is completely decoupled from the global tree unfolding control flow, so that the path strategy only expands *one step*, in contrast to wrapping the strategy body with a closure (i.e., preferring $P \equiv \zeta$ over $P \equiv \zeta*$, $P \equiv \zeta+$, etc.). Several named strategy definitions are provided so that the user can quickly choose between $safe(P)$, $safe^{*}(P)$, $safe^{+}(P)$ and $safe^{!}(P)$ as the global strategy for each computation tree to explore, at run time, having encoded the path strategy P only once and without repeating the transformation time.

The generated strategy module contains a set of strategy definitions named s`-path` and $s\bullet$, where s is the name of a sort $s \in S$ and $\bullet \in \{*, +, !\}$, which respectively correspond to $safe(P_s)$ and $safe^{\bullet}(P_s)$. For example, for an initial term of sort `Configuration`:

- `Configuration-path` expands to $safe(P_{\texttt{Configuration}})$,
- `Configuration*` expands to $safe^{*}(P_{\texttt{Configuration}})$,
- `Configuration+` expands to $safe^{+}(P_{\texttt{Configuration}})$ and
- `Configuration!` expands to $safe^{!}(P_{\texttt{Configuration}})$.

The strategy module contains enough definitions to cover all possible combinations the user may need, even though the actual mappings between those publicly exposed names and the generated strategy expressions is decided by optimizations and is considered an implementation detail.

If the initial given term t_0 has sort $t :: u$, $u \in S$, but no corresponding path strategy P_u has been defined, the default path strategy $\boxed{\texttt{all } \bullet}$ will be used implicitly when calling the named definition $s\bullet$.

EXAMPLE 8

Consider again the safety policy defined in Example 6, which defines safety for the controller of a space probe. Let us show how we can encode the path strategy shown in line 6 of Listing 3.1, after undergoing the safe path strategy transformation described in Definition 13:

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

$$\Downarrow$$

```
                                  State-state ;
( idle | (gv-notVisible-visible ; State-state) | ... | (pm-slewing-science ; State-state) ) ;
                                 advance-time ;
                                  State-state
```

Note that `State-state` here is a call to the lifted state filter $F_{\mathcal{A}}$. The first occurrence of `State-state` corresponds to the first appearance of $F_{\mathcal{A}}$ preceding $safe'(P)$ in Definition 13, while the rest of the strategy after the first ; is the result of the auxiliary transformation described in Definition 12. The concatenation of rules next to `idle` using the choice operator | corresponds to the expansion of the macro `all-(advance-time)`. Finally, the last call to $F_{\mathcal{A}}$ is due to the appearance of the rule label `advance-time`, which is explicitly called in the original path strategy.

Recall that, as explained in Section 2.3, the `search` command allows the Maude user to explore the computation tree $\mathcal{T}_{\mathcal{R}}$ of a program $\mathcal{R}$. Using the generated strategies, the `srew` command is driven to explore a constrained, safe version of $\mathcal{T}_{\mathcal{R}}$ where all states and transitions between states satisfy the safety policy $\mathcal{A}$, thus performing a similar function where the only difference resides in the fact the later enforces $\mathcal{A}$. Indeed, the strategy definitions exposed by our technique leverage the acquaintance with the `search` command to carefully expose a conceptually *safe equivalent* of the `search` command by means of the search shorthands introduced in Definition 14.

Given an initial state, the following `search` command explores the next four states stemming from it by applying one 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 which fulfill the basic domain restrictions explained in Example 5, such as the duration bounds, but that are not necessarily safety w.r.t. the safety policy in Example 6. The first solution corresponds to one application of the time advancement rule. The second solution arises from the transition of the pointing mode from `comm` to `earth`. The third solution results from the transition of `pm` from `comm` to `maintenance`. There are two ways in which this violates the considered safety policy: first, in the both transitions of the value of the variable `pm` (solutions 2 and 3) time had not advanced, violating the rule that a clock cycle requires the intertwining of state variable changes and time advancement; and second, the fourth solution corresponds to an unsafe state because a state assertion in our safety policy explicitly forbids the communication pointing mode ($\mathit{value}(\texttt{pm}) = \texttt{comm}$) while the probe is not visible from mission control ($\mathit{value}(\texttt{gv}) = \texttt{notVisible}$).

Let us now try to issue the safe equivalent of the previous command, which explores the next four *safe* states reachable using one or more transitions, as indicated by the search shorthand `State+`, which here corresponds to the `=>+` option from the `search` 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 }
```

As before, four states are returned, but this time all of them satisfy the safety policy. Note that in solutions 2 and 3 the state variable change is now accompanied by the corresponding time advancement, as observed in the duration of `gv`, which increments from `70` to `71`. Furthermore, the solution 4, previously explicitly unsafe, has now been suppressed in favor of a new, different solution that is naturally also safe. This solution was the next one in the line of solutions to be explored, and hence would have been solution number 5 in the previous `search` command if it were not limited to only showing its first four solutions.

As defined in previous sections, global strategies call or use the lifted state filters $F_{\mathcal{A}}$. These lifted filters act in the scope of one step of the tree, pruning the children nodes of an application (or set of concurrent applications) of a rewrite rule, and effectively fulfilling the role of local strategies. In case the user wishes to apply $F_{\mathcal{A}}$ on demand in order to check safety w.r.t. states without building a computation tree, a strategy named s`-state` must be invoked, where $s \in S$ is the sort of the provided term.

Let $\mathcal{A}$ be a safety policy. Finally, and completing the hierarchy, state filter strategies correspond to the state filters F_a for a state assertion $a \in \mathit{state}(\mathcal{A})$. They are used by the local strategy $F_{\mathcal{A}}$. These strategies are not intended to be called directly, and the names of the corresponding strategy definitions are an implementation detail.

Example 9

Recall the artificial satellite controller, alongside its accompanying safety policy, from Example 6.

The resulting transformed program containing the generated strategy module is available in the companion file `satellite_fixed.maude`[3]. We show a formatted excerpt of the generated strategy module in Listing 3.3.

[3]This file may be downloaded from https://raw.githubusercontent.com/skneko/strass/master/core/examples/satellite/fixed/satellite_base.maude

```
smod SATELLITE-SAFE is
  protecting SATELLITE .
  protecting EXT-BOOL .
  op max : TimeInterval -> Time .
  eq max([Tmin:Time, Tmax:Time]) = Tmax:Time .
--- ...
  strat State! : @ State .
  strat State* : @ State .
  strat State+ : @ State .
  strat State-path : @ State .
  strat State-state : @ State .
--- ...
  strat TimeOrInf-state : @ TimeOrInf .
  strat s1 : @ Time .
  strat s2 : @ State .
  strat s3 : @ State .
  strat s4 : @ State .
  strat s5 : @ State .
  strat StateVariable! : @ StateVariable .
  strat StateVariable* : @ StateVariable .
  strat StateVariable+ : @ StateVariable .
  strat StateVariable-state : @ StateVariable .
--- ...
  sd State! := State-state ; (State-path !) .
  sd State* := State-state ; (State-path *) .
  sd State+ := State-state ; (State-path +) .
  sd State-path := (
      idle
      | (gv-notVisible-visible ; State-state)
      | (gv-visible-notVisible ; State-state)
      --- ...
      | (pm-slewing-science ; State-state)
      ; advance-time ; State-state .
  sd State-state := s2 ; s3 ; s4 ; s5 ; s1 .
  sd StateVariable! := (all) ! .
  sd StateVariable* := (all) * .
  sd StateVariable+ := (all) + .
  sd StateVariable-state := idle .
--- ...
  sd TimeOrInf-state := s1 .
  sd s1 := not(amatch T:Time s.t. T:Time >= 0 = false) .
  sd s2 := not(amatch
        {pm: Tpm:Time comm,
         gv: Tgv:Time GV:GroundVisibility,
         is: Tis:Time IS:InstrumentStatus}
      s.t. GV:GroundVisibility == visible = false) .
  sd s3 := not(amatch
        {pm: Tpm:Time maintenance,
         gv: Tgv:Time GV:GroundVisibility,
         is: Tis:Time IS:InstrumentStatus}
      s.t. IS:InstrumentStatus == idle = false) .
  sd s4 := not(amatch
        {pm: Tpm:Time maintenance,
         gv: Tgv:Time notVisible,
         is: Tis:Time IS:InstrumentStatus}
      s.t. true) .
  sd s5 := not(amatch
        {pm: Tpm:Time science,
         gv: Tgv:Time GV:GroundVisibility,
         is: Tis:Time idle}
      s.t. Tis:Time <= 2 * max(duration(process)) = false) .
endsm
```

Listing 3.3: Excerpt of the generated strategy module for the space probe controller

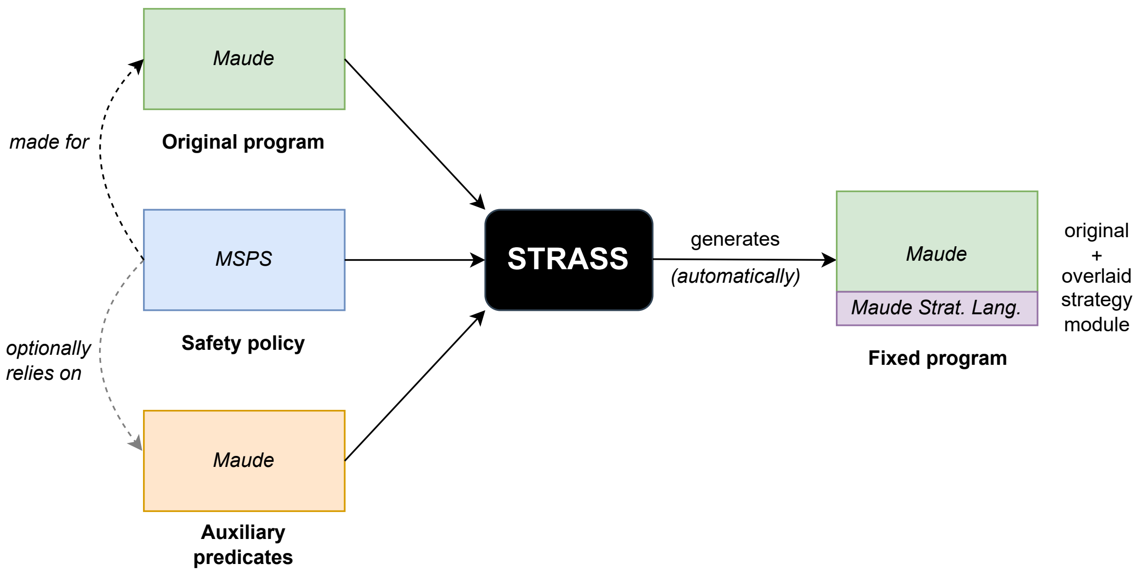
CHAPTER 4

STRASS: Strategy-based Automatic Safety Assurance Tool

The introduced formal framework and correction technique have been efficiently implemented in an interactive tool called STRASS (**Str**ategy-based **A**utomatic **S**afety A**s**surance Tool), which is described in detail in this chapter. The tool features a friendly web user interface and is publicly available at `http://safe-tools.dsic.upv.es/strass`.

Following the presented methodology, the tool takes as input a Maude program and a MSPS safety policy (see Section 3.1), and returns a new Maude program which that is safe w.r.t. the provided safety policy (see Section 3.2). 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. This is required for ensuring the correctness of the transformation.

Figure 4.1: High level view of the inputs and outputs of STRASS



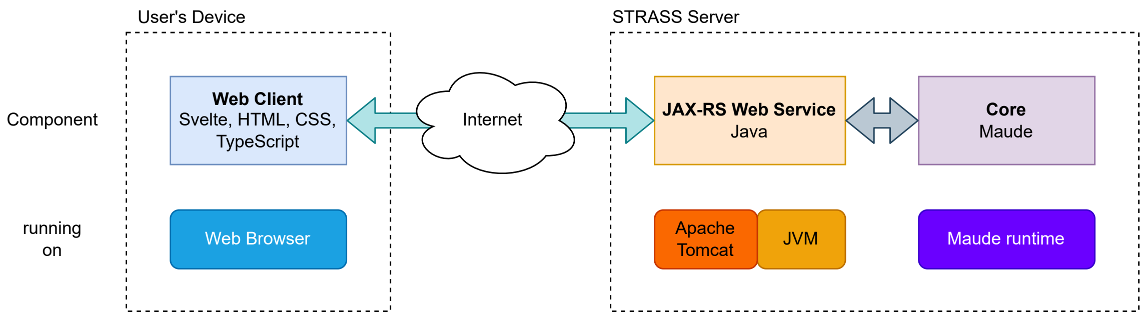
4.1 Architecture Overview

STRASS is divided into three components that are implemented using appropriate technologies that are better suited for their purposes. The implementation consists of about

800 lines of Java source code; 1200 lines of Maude[1] code and 2000 lines of combined HTML, CSS, TypeScript [7] and Svelte [5].

The first component is a *web client* which is automatically downloaded and rendered by the user's web browser when accessing the URL of STRASS website. This web client takes the responsibility of offering a friendly graphical user interface and conducting the interaction with the user. As this interaction dynamically occurs, this web client sends asynchronous AJAX requests through the network [1] to an HTTP RESTful API exposed by our server. This API is implemented using a Java *web service* running on an Apache Tomcat [6] server instance.

Figure 4.2: Overview of the architecture of STRASS



The described technique is implemented entirely in Maude using its meta-level capabilities, which allow for efficient and ergonomic program analysis and repair. The exposed Java service uses operating system utilities for process-to-process communication with a Maude interpreter instance running STRASS. We call this component holding our domain logic the *core* of the tool.

The tool follows RESTful principles and is entirely stateless, so it does not feature any type of persistence. However, the web client component holds some transient state, scoped to the duration of one session, to provide for basic functionality of the graphical user interface.

4.2 Web Client

The web client component has been implemented using a technology made for building complex web applications called *Svelte*. Svelte is a reactive, high-performance programming language and system embedding a custom derivative of JavaScript into HTML and CSS. Svelte source code can be compiled into a bundle of standard HTML, CSS and JavaScript that can be distributed using a static server; the application runs entirely in the user's browser. In order to benefit from static typing, Svelte optionally accepts the partial or total use of TypeScript instead of JavaScript for driving the application's logic. STRASS is using this option and features exclusively TypeScript for the user interface logic.

Regarding styling and layout, the web client uses the popular library *Bootstrap* [2], which includes a wide variety of CSS classes for quickly annotating the HTML markdown with styling rules, alongside custom CSS definitions.

Usability has been an important focus during the development of the tool. The user experience design is based upon the well-known concept of assistant or "wizard", presenting a sequence of dialog boxes that lead the user through a series of three predefined

[1]We use a developer version of Maude implemented in C++ called *Mau-Dev* [3].

steps. Two buttons, *Next* and *Previous*, consistently appear in the same section during the whole session, allowing the user to proceed or go back to a previous step, respectively. In order to enhance usability and prevent loss of information, the web client explicitly preserves and restores all the introduced information and view state when necessary, so that going back to a previous step or finding a validation error will redraw the dialog box exactly as it was left by the user.

The interface features several text editors where the user may input and edit the different inputs provided for the tool, such as the safety policy. Those editors are powered by Monaco[4], a free and open-source framework for making web text editors developed by Microsoft that is used as a base for Visual Studio Code. This provides a rich set of advanced editing techniques: the user may insert several cursors for concurrent editing by holding down the Alt key and clicking, or may jump automatically to the next error by pressing ⇧Shift+F8. Pressing the F1 key while using an editor shows a command palette listing all available actions with their corresponding key associations. Furthermore, the provided inputs are validated for errors using a complete Maude interpreter instance, to then add hints and markings providing rich diagnostic information and advice in the text editors, resulting in an experience comparable to the one of IDE programs.

4.3 Web Service

In order to provide the service for the web client, a server exposes a RESTful API based on the industry standard practice of using the HTTP protocol for carrying JSON data payloads. The component we call the *web service* refers to the definitions that describe the API and its methods, stating the arguments and the responses associated to the different endpoints. A *web server* acts an an underlying and necessary component, receiving the HTTP requests and transferring the already processed and decoded data to the web service, to finally build and send appropriate HTTP responses according to the decisions taken by the service.

This implementation of the web service uses Jakarta RESTful Web Services (JAX-RS), a technology originally developed by Sun Microsystems and now maintained by the Eclipse Foundation. It is an API specification for the Java language that empowers the programmer to quickly and effectively define web services that follow the Representational State Transfer (REST) architectural pattern. An API can be defined with JAX-RS by using functions with specific annotations. This annotated Java code can then be compiled and bundled alongside some assets as a WAR file, to then be deployed in any web server compatible with this specification. The specific JAX-RS compatible web server used to power STRASS is Apache Tomcat.

In order to communicate with the underlying Maude interpreter instance, the web service uses an operating system independent API provided by Java's standard library in order to launch a *shell* spawning this interpreter as a new process. The service also uses the standard input and output streams, which are also exposed by this Java API and are transparently provided by the underlying operating system. The data going to (resp., coming from) Maude through the standard streams undergoes a *marshalling* (resp., unmarshalling) process, suffering minimal transformations that make it suitable for direct processing by the corresponding tools.

Table 4.1: Summary of the endpoints in the REST API of STRASS

Verb	Path	Description of path	Arguments	Type	Description of arguments
`POST`	`/program/check`	Validates whether a program is well-formed.	`program`	string	The Maude source code to be checked.
`POST`	`/constraints/check`	Validates whether a safety policy is well-formed.	`programWithAddendum`	string	The full Maude source code containing the program plus all auxiliary predicates and definitions.
			`rootModuleName`	string	The name of the module to load and use as root of the input program, which will in turn import other modules.
			`addendumModuleName`	string	The name of the module to load as root of the auxiliary predicates exposed in the safety policy.
			`constraints`	string	The MSPS source code of the safety policy to check.
`POST`	`/program/fix`	Generates the strategy module for the input program.	`programWithAddendum`	string	The full Maude source code containing the program plus all auxiliary predicates and definitions.
			`rootModuleName`	string	The name of the module to load and use as root of the input program, which will in turn import other modules.
			`addendumModuleName`	string	The name of the module to load as root of the auxiliary predicates exposed in the safety policy.
			`constraints`	string	The MSPS source code of the safety policy to enforce.

4.4 Domain Logic Core

The core component of the tool implements the correction technique and formal framework. The whole domain logic is entirely programmed in this component, with the other components only performing the minimal amount of transformations on data that are needed to progress through the different tools and environments involved.

This component is entirely implemented in Maude. The Maude language has a powerful and wide set of reflection capabilities that make it especially suitable for inspecting and modifying other Maude programs: using Maude code, a user may take some module and then *ascend* it into the *meta-level* obtaining a *meta-module*, i.e., converting it to a rich data structure containing all required definitions and metadata, which can be read using a set of functions defined in Maude's prelude. Also, a user may construct a meta-module either from scratch or from modifying an existing one, to then *descend* from the meta-level again into runnable source code. These capabilities are used as the basic foundation of the implementation of STRASS: first, meta-level constructs (meta-modules, meta-terms, meta-operations...) are obtained from the given input code, then methodically read in order to extract data, and finally new meta-level elements are generated so they can eventually descend into the new program source code.

We summarize the actions taken by the core logic of STRASS as four phases that are executed in order:

Phase 1. **Meta-Level Ascent**: the input program is transformed into rich meta-level data structures using functions defined in the prelude (standard library).

Phase 2. **Static Analysis**: the input program is analysed to extract data and derive new information.

Phase 3. **Parsing**: the safety policy is processed using a custom parser and then validated.

Phase 4. **Generation**: the new strategy module and its contained definitions are generated at the meta-level.

Phase 5. **Formatting** (meta-level descent): the generated constructs are converted into a string of pretty-printed, distributable source code.

The subsequent sections describe all of the phases in detail.

4.4.1. Initialization Steps

The first phase consists on a small sequence of straighforward calls to function `upModule` available in Maude's prelude, providing the input source code of both the provided program and the auxiliary predicates (which, contrary to the MSPS safety policy, are just regular Maude code). Recall that Maude modules may "import" other modules, and so the `upModule` function allows choosing whether to load a self-contained, flattened meta-module with all the transitive dependencies included, or a module that is accurate w.r.t. the the original one, with the import directives left unresolved. This tool loads the provided source code using the former mode of operation, ascending into a flattened meta-module in order to support multi-module programs accurately.

Right after Maude meta-level capabilities have reflected the provided source code into tractable rich data structures, the second phase performs a static analysis procedure which is limited in scope but not trivial. The objective of this analysis is extracting useful data by processing the obtained meta-module, which can be later consumed in the following phases of the procedure.

4.4.2. Parsing the Safety Policy

The next phase consists on the parsing of the provided safety policy, which is written in our custom language MSPS as defined in 3.1. STRASS includes a custom parser that leverages Maude meta-level commands in order to support all the features the user would expect from Maude's parser, such as mixfix operators or custom precedence attributes [15, Section 3.9]. This custom parser takes (i) the entire signature Σ of the input program, (ii) the auxiliary predicates provided by the user and (iii) the automatically generated custom definitions. Together, they form are used to generate a transient meta-module we call the *parsing context*, including all operators that should be taken into account for parsing. These custom definitions define the different syntactic constructs of MSPS, instantiated for the specific sorts contained in the provided program. Recall that state assertions have the form $\boxed{\Pi \texttt{ \# } \varphi}$ where $\Pi \in \tau(\Sigma, \mathcal{V})$ is a *pattern* term and φ is a logic formula, and that path strategies have the form $\boxed{\texttt{path for } s \texttt{ : } \zeta}$ where $s \in S$ is a sort and ζ is a strategy expression. The following is an example illustrating two of the custom definitions added to this parsing context:

```
op _#_ : k [Bool] -> __STRASS_MonomorphizedConstraint [ctor prec(0) gather(& &)] .
op path`for_:_ : __STRASS_SortName __STRASS_Action -> __STRASS_MonomorphizedConstraint [ctor prec(0) gather(& &)] .
```

where $k \in \{[s] \mid s \in S\}$ is a kind in S. Several copies of the first operator are included for the different kinds the variable k may take. Specifically, after collecting all the sorts of the patterns of the provided state assertions, there will be as many copies as different kinds appear from lifting such sorts, that is, there are as many copies as kinds in $\{[ls(\Pi)] \mid \Pi\#\varphi \in states(\mathcal{A})\}$, where $\mathcal{A}$ is the provided safety policy and $ls(t)$ returns the least sort of a term t. In order to construct and support these definitions and others that are not shown here, some retrieved information from the static analysis phase is used, such as the set of sorts S and the set of existing rule labels $labels(R)$, where R is the set of rewrite rules encoded by the provided input program.

The task of parsing becomes more complex when we factor in scopes, i.e., which definitions should be visible from different points of the program. In the pattern Π of a state assertion we wish to use the operators and sorts defined in the considered theory $\mathcal{R}$, allowing for the unbounded definition of any kind of term that can be constructed in the domain of $\mathcal{R}$. Regarding the formula φ, we not only wish to include all operators of $\mathcal{R}$ (so operators acting as functions can be used as part of the invariant condition), but the auxiliary predicates as well. It is important to keep this distinction because the safety policy $\mathcal{A}$ models the safety of $\mathcal{R}$ *only*, and allowing definitions from the auxiliary predicates to permeate into the patterns Π would break this semantic mapping between $\mathcal{A}$ and $\mathcal{R}$, enabling the definition of state assertions that parse without error yet have no translation nor effect in the final transformed program. These diverging needs are approached by using two different parsing contexts, one including the auxiliary predicates and the other one excluding them.

On top of that, and enhancing the parsing function further, the parser has been endowed with the capability of tracking line numbers and error codes for the different syntactic constructs in the safety policy, allowing for a rich API response that accurately identifies which specific state assertions or path strategies are erroneous instead of outright rejecting the entire safety policy as a whole.

4.4.3. Generating the New Strategy Module

Once parsing is complete, we have both the input program and the safety policy available as programatically tractable data structures, and hence the strategy module generation phase can begin. Strategy modules may contain all varieties of definitions acceptable in a system module together with the generated strategy definitions. A series of functions take control of this phase by filling a new strategy module with a series of contained definitions, without polluting the name space of the original program, which will remain isolated and unmodified in its original module. Let us recall that this new strategy module will then be appended to the original program, simply extending it.

Remember that named strategy definitions in Maude are defined using two different syntactic constructs that use the keywords `strat` and `sd`, and are analogous to `op` and `eq`, respectively. We generate such constructs in a pairwise manner, maintaining a strict one-to-one relation. First, we encode the provided state assertions as state filter strategies $F_{\Pi \,\#\, \varphi}$, $\Pi \,\#\, \varphi \in \mathit{states}(\mathcal{A})$, and wrap the newly created expressions in named strategy definitions which use the name `s`i, $i \geq 1$. This is, for each state assertion $\Pi \,\#\, \varphi \in \mathit{state}(\mathcal{A})$ the following definitions are added to the strategy module:

```
strat s$i$ : @ $ls(\Pi)$ .
  sd s$i$ := $F_{\Pi \# \varphi}$ .
```

Note that the subject type of the strategy at hand is mainly derived from the pattern Π of the corresponding state assertion, as this pattern determines which terms are subject to the invariant check of φ. Thus, the type signature of the strategy is defined by using the least sort of Π.

Secondly, the lifted state filter $F_{\mathcal{A}}$ is generated, which enforces all state assertions for *any given term* of the provided input program. As stated in Definition 11, this strategy will call all of the state filter strategies `s`i: `s1 ; s2 ; ... ; s`n. Let us temporarily use the identifier `all-state` to refer to the corresponding named strategy definition:

```
strat all-state : @ $\tau$ .
  sd all-state := $F_{\mathcal{A}}$ .
```

where τ represents a type. What should this type τ be? It is important to remember that this strategy may be applied to *any* term of the rewrite theory $\mathcal{R}$, and that the different state filter strategies `s`i may have any subject type with no specific or predictable relation between them. With enough background knowledge of the Maude language, it could be natural to use the polymorphic sort `Universal` for τ, hence expressing that we wish this strategy to apply to any term regardless of its type. However, this is not allowed in Maude version 3.2.1. Thus, this `all-state` named strategy definition is not generated, and it is necessary to engineer a semantically equivalent method that can satisfy the type safety guarantees being enforced by Maude leveraging the type checking instead of trying to unsafely override it.

Our approach is based on *monomorphization* [24, Section 2]: 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 concrete enough for some implementation purpose, e.g., for generating concrete machine code in a compiler. In this case, a series of clones of the given strategy definitions are created to satisfy all the possible type signatures that are necessary for the input program and safety policy tandem:

```
strat $s_i$-state : @ $s_1$ .
```

```
sd $s_i$-state := $F_{\mathcal{A}}^{s_1}$ .
        ...
strat $s_n$-state : @ $s_n$ .
  sd $s_n$-state := $F_{\mathcal{A}}^{s_n}$ .
```

where $0 < i \leq n$, $s_i \in S$ are sorts in the rewrite theory $\mathcal{R}$ encoded in the provided input program.

The purpose of this technique is twofold: it allows us to generate a valid strategy module whose contained type signatures are guaranteed to satisfy Maude's type checker; and it also allows optimizations that benefit from knowing the specific type that will be affected by a strategy, while having room for modifying the strategy as several independent copies exist. In particular, monomorphization enables the optimization technique we have called *sort-dependence filter erasure*: in each individual named strategy definition s_i`-state`, the wrapped lifted state filter strategy $F_{\mathcal{A}}^{s_i}$ is simplified by conditionally removing calls to filter state strategies $F_{\Pi \,\#\, \varphi}$ when it is possible to statically ensure that $\Pi \,\#\, \varphi$ vacuously holds because Π does not match subterms of any term of sort $s \in S$. For instance, if no term of sort `System` contains a natural number then $F_{\mathcal{A}}^{\texttt{System}}$ may safely omit the state filter strategies that are only relevant to natural numbers. In order to formalize this optimization, we define a new notion of "sort dependency".

Definition 15. Let S be the set of sorts of the rewrite theory $\mathcal{R}$ encoded in the input program. We say a sort $s \in S$ is *dependent* upon another sort $u \in S$, denoted $s \Lleftarrow u$, if there exists a term t and a position w such that $t :: s$ and $t_{|w} :: u$.

Roughly, this concept of sort dependency encodes whether a sort may be "constructed from" terms of other sorts. For example, consider a sort `NatList` that is the type of terms describing valid lists of natural numbers, e.g., if the infix operator `_::_ : Nat NatList -> NatList` is a list constructor, then the term $\boxed{\texttt{1 :: 2 :: 3}}$ is a `NatList`. We say this sort depends on the sort `Nat`, `NatList` $\Lleftarrow$ `Nat`, because it may (and in this case, *is*) "constructed from" natural numbers such as `1`, `2` and `3`. If we made the sort `Nat` disappear by removing all of its constructors, the number of terms of sort `NatList` that could exist would be reduced, e.g., $\boxed{\texttt{1 :: 2 :: 3}}$ would not be a valid term anymore. This notion is useful to determine which state assertions are relevant for terms of a given sort, as explained later.

One of the specific steps performed by our static analysis is finding and storing sort dependencies. If S is the set of sorts of the rewrite theory $\mathcal{R}$, this step generates a data structure called the *sort dependency map* (SDM), associates each sort to the set of sorts it transitively depends upon:

$$(\forall u \in S)\ SDM(s) = \{u \mid s \Lleftarrow^{+} u\} \subseteq S$$

where $\Lleftarrow^{+}$ denotes the transitive closure of the relation $\Lleftarrow$.

Example 10

Consider again the space probe controller from Example 5. A possible example excerpt of the sort dependency map (SDM) of the program is as follows:

$$\begin{aligned}
SDM(\texttt{GroundVisibilityState}) &= \{\texttt{GroundVisibility}, \texttt{Int}, \texttt{Nat}, \texttt{NzInt}, \texttt{NzNat}, \texttt{Time}, \texttt{Zero}\}\\
SDM(\texttt{InstrumentStatusState}) &= \{\texttt{InstrumentStatus}, \texttt{Int}, \texttt{Nat}, \texttt{NzInt}, \texttt{NzNat}, \texttt{Time}, \texttt{Zero}\}\\
SDM(\texttt{State}) &= \{\texttt{GroundVisibility}, \texttt{GroundVisibilityState},\\
&\quad \texttt{InstrumentStatus}, \texttt{InstrumentStatusState}, \texttt{Int}, \texttt{Nat},\\
&\quad \texttt{NzInt}, \texttt{NzNat}, \texttt{PointingMode}, \texttt{PointingModeState}, \texttt{Time}, \texttt{Zero}\}\\
SDM(\texttt{StateVariable}) &= \{\texttt{GroundVisibility}, \texttt{InstrumentStatus}, \texttt{PointingMode}\}
\end{aligned}$$

For instance, we can see that `GroundVisibilityState` transitively depends on `Time`, that is, `GroundVisibilityState` $\Leftarrow^{+}$ `Time`. For this specific case, if we examine Listing 2.6 we can observe in line 7 an operator explaining this dependence:

```
7 op gv:__ : Time GroundVisibility -> GroundVisibilityState .
```

It is possible to obtain the entire SDM using STRASS from a command line, by loading it alongside the file containing the satellite controller program like this:

```
maude strass/core/src/main.maude satellite.maude
```

and then executing this command:

```
Maude> reduce in STRASS : arrangedSortDependencies(upModule('SATELLITE, true)) .
result SortDependencyMap: 'Bool |-> ('GroundVisibility ; 'Infinity ; 'InstrumentStatus ;
'Int ; 'Nat ; 'NzInt ; 'NzNat ; 'PointingMode ; 'StateVariable ; 'Time ; 'TimeInterval ;
'TimeOrInf ; 'Universal ; 'Zero), 'GroundVisibilityState |-> ('GroundVisibility ; 'Int ;
'Nat ; 'NzInt ; 'NzNat ; 'Time ; 'Zero), --- ... result truncated ...
```

Having laid down Definition 15, the sort-dependence filter erasure optimization can be described as follows. Let $\mathcal{R}$ be the rewrite theory encoded by the provided input program, let S be the set of sorts in $\mathcal{R}$ and let $\mathcal{A}$ be the provided safety policy. Having that the lifted filter state strategy $F_{\mathcal{A}}$ is equal to the sequential concatenation F_{a_1} ; ... ; F_{a_n} of the set $\{F_{\Pi_i \,\#\, \varphi_i} \mid (\Pi_i \,\#\, \varphi_i) \in \mathit{states}(\mathcal{A})\}$, then each monomorphized filter state strategy expression $F^{s}_{\mathcal{A}}$, $s \in S$, is equal to the sequential concatenation of the (sub)set

$$\{F_{\Pi_i \,\#\, \varphi_i} \;\mid\; (\Pi_i \,\#\, \varphi_i) \in \mathit{states}(\mathcal{A}) \;\wedge\; \mathit{ls}(\Pi_i) \in \mathit{SDM}(s)\}$$

If this set is empty, the generated strategy $F^{s}_{\mathcal{A}}$ defaults to `idle`, which is the neutral element of the sequential operator `;`.

EXAMPLE 11

Consider again our driving example of the artificial satellite controller originally described in Example 5, and the generated strategy module in Listing 3.3. We use this listing to show the typical outcomes caused by the implemented optimizations in a realistic example. The effects of the sort-dependence erasure optimization can be observed in line 40, where the lifted state filter monomorphized for terms of sort `TimeOrInf` only uses the filter `s1` and ignores the others because they are not relevant (i.e., they always vacuously hold) for said terms. Furthermore, from line 34 to 37, in all of the search shorthands for the sort `StateVariable`, `StateVariable`•, the body has been reduced to the simplified form `all` • because of the inlining optimization.

This optimization is useful because it generates simpler strategies that may be easier to understand and modify by the final user, may he ever want to undergo such optional task. Also, removing safety checks in a semantically safe manner may potentially improve performance, as the overall run time burden required to ensure safety is reduced to strictly fewer strategy operations.

The next step consists in generating the definitions of safe path strategies for the provided path strategies in the safety policy. Let $s \in S$ be a sort in $\mathcal{R}$ and let ζ be a strategy expression. For each path strategy P_s, which is denoted by indicating the sort s explicitly using the syntax $\boxed{\texttt{path for } s \texttt{ : } \zeta}$, the following definitions are generated:

$$\begin{array}{c}
\texttt{strat } s\texttt{-path : @ } s \texttt{ .} \\
\texttt{sd } s\texttt{-path := } safe(P_s) \texttt{ .}
\end{array}$$

In the expression created by the transformation $safe(P_s)$, recall the state filter $F_{\mathcal{A}}$ is s`-state`.

Finishing the strategy generation step, a series of search shorthands are generated according to Definition 14:

$$\begin{array}{c}
\texttt{strat } s\texttt{* : @ } s \texttt{ .} \\
\texttt{strat } s\texttt{+ : @ } s \texttt{ .} \\
\texttt{strat } s\texttt{! : @ } s \texttt{ .} \\
\texttt{sd } s\texttt{* := } safe^{*}(P_s) \texttt{ .} \\
\texttt{sd } s\texttt{+ := } safe^{+}(P_s) \texttt{ .} \\
\texttt{sd } s\texttt{! := } safe^{!}(P_s) \texttt{ .}
\end{array}$$

where $safe^{\bullet}(P_s)$ defaults to $\boxed{F_{\mathcal{A}} \texttt{ ; ((all ; } F_{\mathcal{A}}\texttt{) } \bullet\texttt{)}}$ if P_s has not been defined for sort s (i.e., if no path strategy has been defined for s).

An additional, cross-cutting optimization called *inlining* is applied as follows. This optimization leads to simpler strategies in certain cases. For example, consider that search shorthands have been generated for a sort s but the corresponding monomorphized state filter $F^{s}_{\mathcal{A}}$ uses its default value because no state assertion is relevant for such a sort s. Then, the inlining transformation consists of the sequence:

$$\begin{array}{c}
\texttt{sd } s\texttt{-state := idle .} \\
\texttt{sd } s\bullet \texttt{ := } s\texttt{-state ; ((all ; } s\texttt{-state) } \bullet\texttt{) .} \\
\Downarrow \\
\texttt{sd } s\texttt{-state := idle .} \\
\texttt{sd } s\bullet \texttt{ := } \underline{\texttt{idle}} \texttt{ ; ((all ; } \underline{\texttt{idle}}\texttt{) } \bullet\texttt{) .} \\
\Downarrow \\
\texttt{sd } s\texttt{-state := idle .} \\
\texttt{sd } s\bullet \texttt{ := } \underline{\texttt{all } \bullet} \texttt{ .}
\end{array}$$

where $\bullet \in \{*, +, !\}$ is used as a compact notation for denoting the three corresponding strategies as one. We say the strategy s`-state` has been *inlined into* the strategies $s\bullet$.

As an additional detail, the generated strategies may need to use operators encoded in the auxiliary predicates in φ at run time, so the auxiliary predicates provided by the user are directly inserted into the new strategy module, effectively exposing them to all of the the generated strategies but not to the original program.

4.4.4. Finishing Up

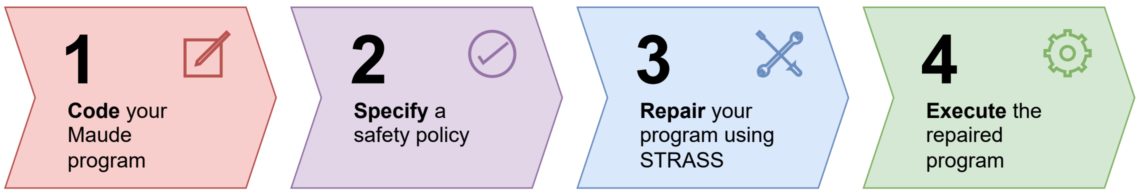
Finally, by executing the process described so far we obtain a strategy module containing all the necessary monomorphized and optimized strategy definitions, alongside their auxiliary predicates. However, this strategy module is actually a *meta*-strategy module, a data structure in the domain of Maude's interpreter. Just like we ascended the source code provided by the user into the meta-level, we now need to descend the generated module from the meta-level into plain, distributable Maude source code.

While Maude's prelude contains operators for descending meta-level structures into source code tokens (e.g., `downModule`), the provided functionality is too limited in scope for the specific needs of STRASS. Hence, a custom descent library has been created, consisting of a functional module whose definitions leverage the existing operators in the prelude as far as possible in order to generate *pretty-printed* source code preserving the semantics of the meta-level structure at hand. The resulting source code is indented and formatted according to the usual style conventions of the Maude ecosystem, enabling the user to easily read and audit the automatically generated declarations. The generated source code is simply appended to the unmodified original source code.

4.5 Features

From a high level user's point of view, the features of the STRASS tool are as follows:

Figure 4.3: Graphical outline of the STRASS usage process


1
Code your Maude program
2
Specify a safety policy
3
Repair your program using STRASS
4
Execute the repaired program

- 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.
- Includes several representative examples that can be rapidly selected from a drop-down list.
- Bleeding edge built-in editor based on Monaco [4], a technology developed by Microsoft and featuring advanced text editing capabilities such as multiple cursor editing and code unfolding.
- Complete syntax highlighting for both Maude and MSPS.
- Detailed error messages with in-editor hints and markings, resembling an IDE.
- Performs automatic, transparent optimizations to simplify the generated strategy module.
- Ability to upload Maude source code files from the local storage.
- Allows the user to avoid modifications on the original program by means of the auxiliary predicate system, making it easier to reason about the properties of the transformed program w.r.t. the original program.
- Includes all empirical evaluation results (benchmark) and artifacts for download.

Overall, this set of features make STRASS a compelling and useful tool which can positively contribute to the Maude community.

4.6 A Typical Repair Session

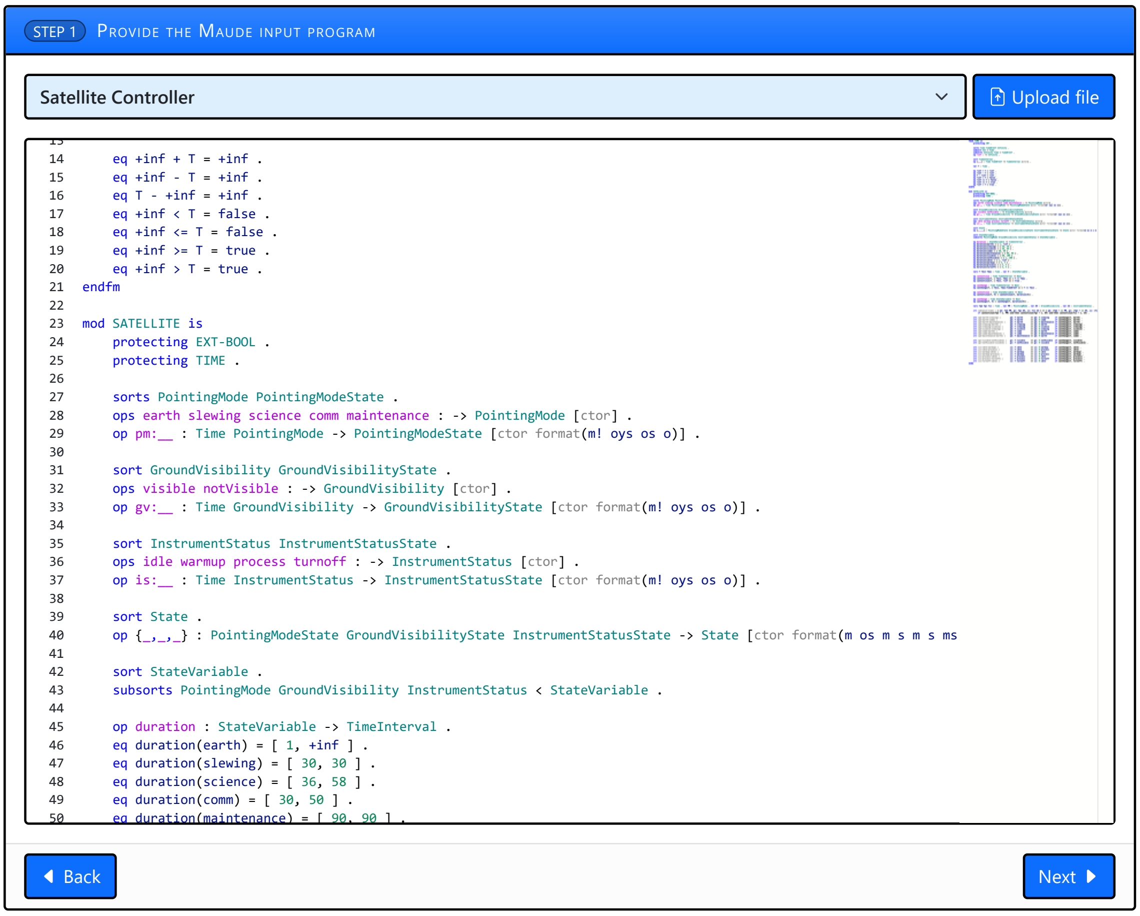
Let us illustrate how STRASS works in practice by showing a typical safety enforcement session for our space probe specification of Example 5.

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. 4.4), whose `SATELLITE` module encodes the space probe controller program $\mathcal{R}_{\texttt{SATELLITE}}$ described in Example 5.

After pressing the Next button, the following phase allows the user to specify the safety policy to enforce by writing into a dedicated editor area, as well as the optional auxiliary predicates it may rely on (see Fig. 4.5). In this case, because the *Satellite Controller* example has been selected, the corresponding areas are already prefilled with the safety policy $\mathcal{A}_{\texttt{SATELLITE}}$ and predicates of Example 6.

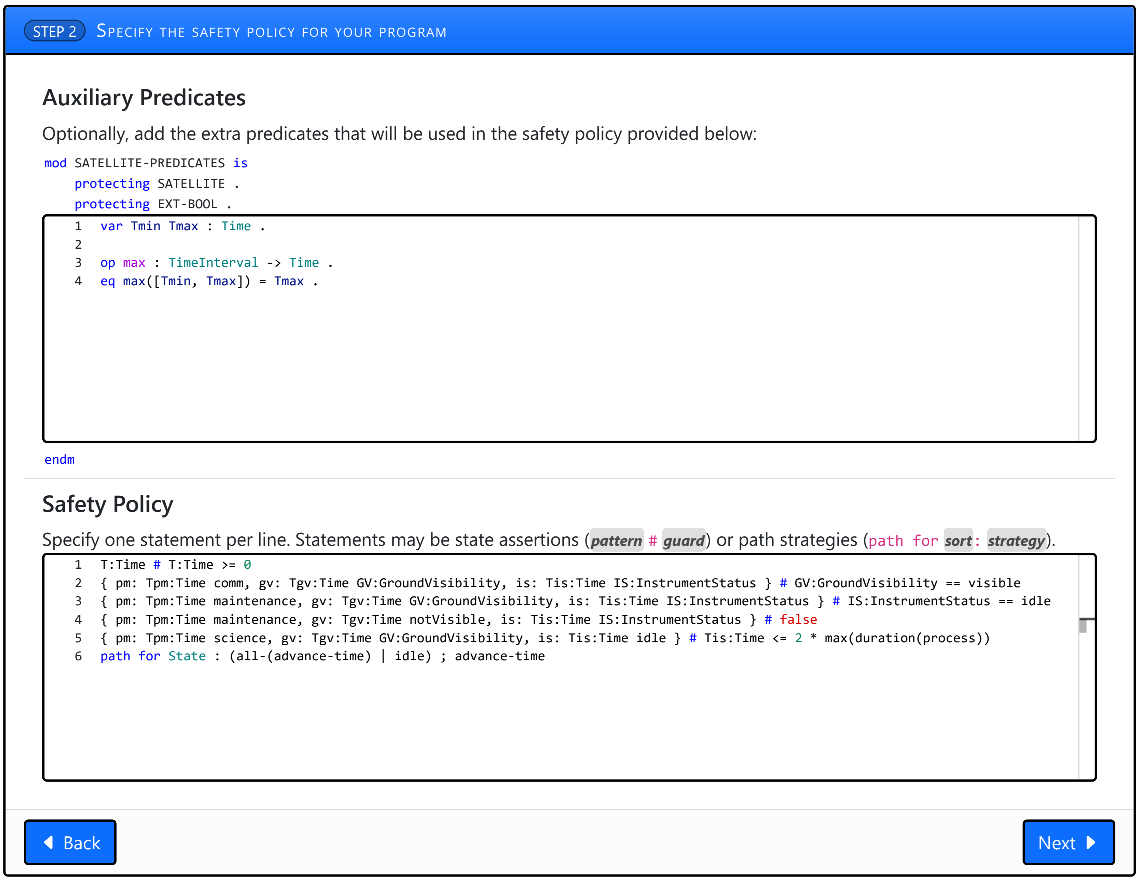
After pressing Next a second time, STRASS will automatically compute and generate the strategy module `SATELLITE-SAFE` encoding $\mathcal{A}_{\texttt{SATELLITE}}$ as shown in Listing 3.3, which is overlaid on top of the original program $\mathcal{R}_{\texttt{SATELLITE}}$, resulting in a new corrected program $\mathcal{R}'_{\texttt{SATELLITE}}$ that is entirely self-contained and may stored and provided to a Maude interpreter instance running in the user's device. This fixed program is given to the user inside the read-only dedicated text area (see Fig. 4.6). Using the resulting program, the user can replicate the behavior shown in Example 8 by issuing the appropriate `srew` commands using the included strategy definitions.

Figure 4.4: Loading the `SATELLITE` module in STRASS



```
    eq +inf + T = +inf .
    eq +inf - T = +inf .
    eq T - +inf = +inf .
    eq +inf < T = false .
    eq +inf <= T = false .
    eq +inf >= T = true .
    eq +inf > T = true .
endfm

mod SATELLITE is
    protecting EXT-BOOL .
    protecting TIME .

    sorts PointingMode PointingModeState .
    ops earth slewing science comm maintenance : -> PointingMode [ctor] .
    op pm:__ : Time PointingMode -> PointingModeState [ctor format(m! oys os o)] .

    sort GroundVisibility GroundVisibilityState .
    ops visible notVisible : -> GroundVisibility [ctor] .
    op gv:__ : Time GroundVisibility -> GroundVisibilityState [ctor format(m! oys os o)] .

    sort InstrumentStatus InstrumentStatusState .
    ops idle warmup process turnoff : -> InstrumentStatus [ctor] .
    op is:__ : Time InstrumentStatus -> InstrumentStatusState [ctor format(m! oys os o)] .

    sort State .
    op {_,_,_} : PointingModeState GroundVisibilityState InstrumentStatusState -> State [ctor format(m os m s m s ms

    sort StateVariable .
    subsorts PointingMode GroundVisibility InstrumentStatus < StateVariable .

    op duration : StateVariable -> TimeInterval .
    eq duration(earth) = [ 1, +inf ] .
    eq duration(slewing) = [ 30, 30 ] .
    eq duration(science) = [ 36, 58 ] .
    eq duration(comm) = [ 30, 50 ] .
    eq duration(maintenance) = [ 90, 90 ] .
```

Figure 4.5: Loading the safety policy $\mathcal{A}_{\texttt{SATELLITE}}$ in STRASS



Auxiliary Predicates

Optionally, add the extra predicates that will be used in the safety policy provided below:

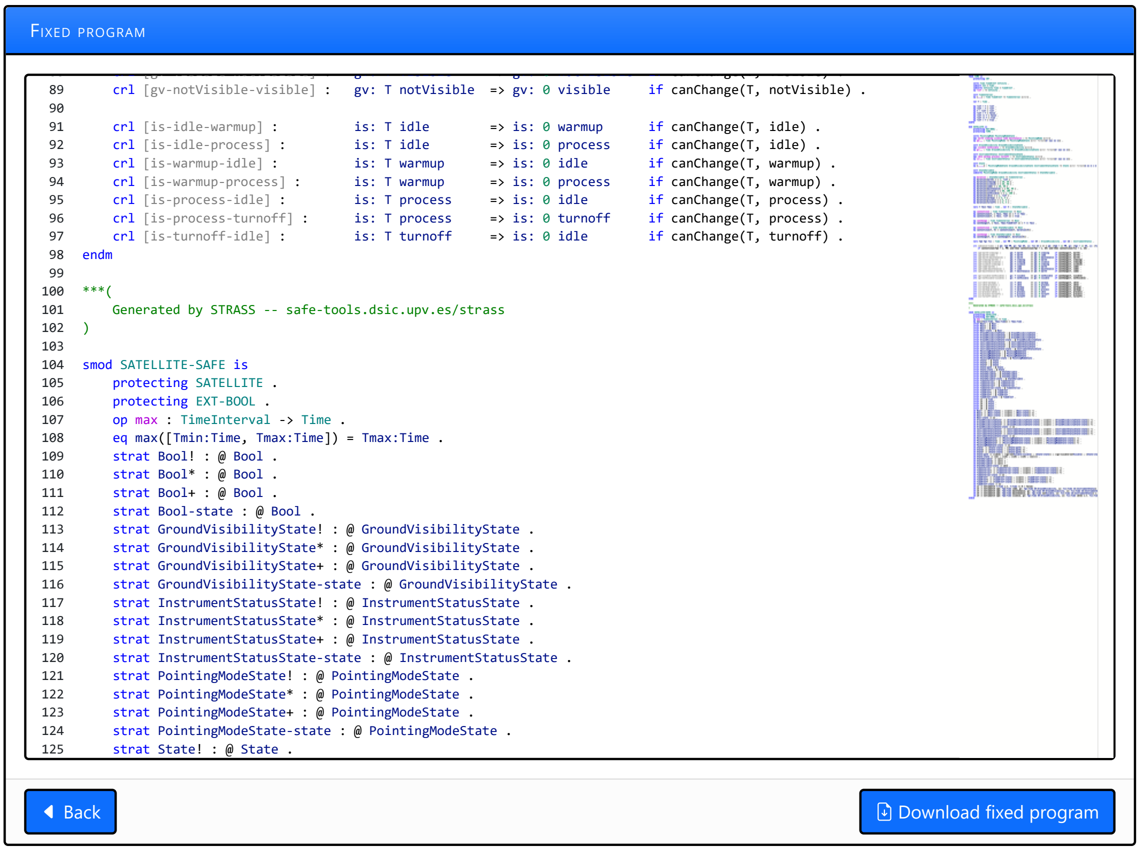
```
mod SATELLITE-PREDICATES is
    protecting SATELLITE .
    protecting EXT-BOOL .
    var Tmin Tmax : Time .

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

Safety Policy

Specify one statement per line. Statements may be state assertions (*pattern* # *guard*) or path strategies (path for *sort* : *strategy*).

```
T:Time # T:Time >= 0
{ pm: Tpm:Time comm, gv: Tgv:Time GV:GroundVisibility, is: Tis:Time IS:InstrumentStatus } # GV:GroundVisibility == visible
{ pm: Tpm:Time maintenance, gv: Tgv:Time GV:GroundVisibility, is: Tis:Time IS:InstrumentStatus } # IS:InstrumentStatus == idle
{ pm: Tpm:Time maintenance, gv: Tgv:Time notVisible, is: Tis:Time IS:InstrumentStatus } # false
{ pm: Tpm:Time science, gv: Tgv:Time GV:GroundVisibility, is: Tis:Time idle } # Tis:Time <= 2 * max(duration(process))
path for State : (all-(advance-time) | idle) ; advance-time
```

Figure 4.6: The resulting Maude module computed by STRASS



```
    crl [gv-notVisible-visible] :  gv: T notVisible  => gv: 0 visible     if canChange(T, notVisible) .

    crl [is-idle-warmup] :         is: T idle        => is: 0 warmup      if canChange(T, idle) .
    crl [is-idle-process] :        is: T idle        => is: 0 process     if canChange(T, idle) .
    crl [is-warmup-idle] :         is: T warmup      => is: 0 idle        if canChange(T, warmup) .
    crl [is-warmup-process] :      is: T warmup      => is: 0 process     if canChange(T, warmup) .
    crl [is-process-idle] :        is: T process     => is: 0 idle        if canChange(T, process) .
    crl [is-process-turnoff] :     is: T process     => is: 0 turnoff     if canChange(T, process) .
    crl [is-turnoff-idle] :        is: T turnoff     => is: 0 idle        if canChange(T, turnoff) .
endm

***(
    Generated by STRASS -- safe-tools.dsic.upv.es/strass
)

smod SATELLITE-SAFE is
    protecting SATELLITE .
    protecting EXT-BOOL .
    op max : TimeInterval -> Time .
    eq max([Tmin:Time, Tmax:Time]) = Tmax:Time .
    strat Bool! : @ Bool .
    strat Bool* : @ Bool .
    strat Bool+ : @ Bool .
    strat Bool-state : @ Bool .
    strat GroundVisibilityState! : @ GroundVisibilityState .
    strat GroundVisibilityState* : @ GroundVisibilityState .
    strat GroundVisibilityState+ : @ GroundVisibilityState .
    strat GroundVisibilityState-state : @ GroundVisibilityState .
    strat InstrumentStatusState! : @ InstrumentStatusState .
    strat InstrumentStatusState* : @ InstrumentStatusState .
    strat InstrumentStatusState+ : @ InstrumentStatusState .
    strat InstrumentStatusState-state : @ InstrumentStatusState .
    strat PointingModeState! : @ PointingModeState .
    strat PointingModeState* : @ PointingModeState .
    strat PointingModeState+ : @ PointingModeState .
    strat PointingModeState-state : @ PointingModeState .
    strat State! : @ State .
```

CHAPTER 5

Empirical Evaluation

We experimentally evaluate the performance and the outcomes of the STRASS system with a set of *benchmarks*. We have coupled several Maude programs with safety policies, and we used STRASS to generate the corresponding safe versions of the programs. 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 controller system of a dam that opens and closes gates to regulate water flow and volume;

(iv) *Maze*, a non-deterministic maze exploration algorithm;

(v) *Space Invaders*, a Maude specification of a classic computer game of the 70's; and

(vi) *Satellite Controller*, the controlling system for an unmanned space probe used as our leading example.

For each benchmark, we considered the original program $\mathcal{R}$ and its safe version $\mathcal{R}_S$ computed by STRASS. Each program has been designed or modified so that, given a term t_0 containing some arbitrary bound b, its search space $\mathcal{T}_{\mathcal{R}}(t_0)$ is limited in depth by a monotonically increasing function of this bound, i.e., increasing b causes the computation tree $\mathcal{T}_{\mathcal{R}}(t_0)$ to have an increased depth. A representative initial term t_0 has been chosen for the particular domain of each benchmarked program. Then, three depth bounds $\{b_1, b_2, b_3\}$ have been chosen for each program, and consequently three executions have been made per program replacing the bound b in t_0 by its corresponding value. Note that, because the bound b is expressed as a finite quantity (in our case, a natural number), the resulting bounded search spaces are also finite, and thus the program execution is also necessarily terminating.

As explained before, having enforced a safety policy $\mathcal{A}$ with STRASS, using the command `srew` it is possible to explore the safe search space for an initial term t_0 of sort $s \in S$, i.e., the constrained computation tree $\mathcal{T}_{\mathcal{R}}^{safe(P_s)}(t_0)$ where $t_0 :: s$. We used the search shorthand `s*` to explore this tree. However, this command is not adequate for the execution of benchmarks because its execution time is heavily influenced by I/O tasks, as the command prints each solution found to the standard output and provides no option to suppress such functionality. We have hence followed another method that leverages the information provided by `srew` to then launch a custom `metaSrewrite` execution at the meta-level guiding the Maude interpreter to compute the desired results with no interference or burden from secondary functionality such as I/O.

In order to explain our method, it is important to note that the `srew` command yields the found solutions, which are just terms, alongside a sequentially increasing natural number which uniquely identifies each solution: the *solution index*. The index of the first solution or term is 1, the next solution is labelled 2, and so on. Knowing that the finite bound b implies termination, we have left the interpreter finish the execution of the issued `srew` commands in order to obtain the entire bounded search space of safe terms. Then, we have noted what is the biggest solution index, which corresponds to the latest solution found by the interpreter.

Finally, we have ascended the initial term t_0 into the meta-level in order to issue a `metaSrewrite` call. This operator is the meta-level equivalent of the `srew` command, with the difference it accepts more options which enable a finer control of the specific workload performed by Maude. In particular, we can use this operator to request Maude to directly compute one specific solution identified by its unique index. The operator is declared as follows:

```
op metaSrewrite :  Module Term Strategy SrewriteOption Nat ~> ResultPair?
```

Given a generic call `metaSrewrite(`m`, `t_0`, `s`, `o`, `i`)`:

1. The first argument of sort `Module`, m, corresponds to a meta-module encoding the rewrite theory under benchmark;
2. the term t_0 corresponds to the initial term;
3. the strategy s is the same as the one used in the `srew` command, $s*$;
4. the strategy rewrite option o is set to `breadthFirst` in order to emulate the behavior of the `srew` command; and
5. the last argument, a natural number (`Nat`), corresponds to the desired solution index.

The execution of a given call to `metaSrewrite` using the biggest possible solution index[1] for argument i yields the last solution found by `srew`, but the running time now essentially corresponds to the computation of the solution. We denote this time as $T_{\mathcal{R}_S}$.

In order to obtain the time of the original program in a fair way, we perform the same procedure using the same term and bounds, but this time using the uncorrected program $\mathcal{R}$ and the trivial strategy `all *`, which is equivalent to a complete search of the entire bounded search space of potentially unsafe terms. Again, we use the index of the last solution obtained to launch a meta-level `metaSrewrite` computation that yields the time $T_{\mathcal{R}}$.

All of the experiments were conducted on an Intel Xeon Silver 4215R 3.3GHz CPU with 378GB of RAM. Table 5.1 summarizes our results. Column *Depth* corresponds to the bound b set 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

[1]Because the `metaSrewrite` operator uses zero-based indexes and the `srew` command provides indexes starting at one, we offset the index by subtracting one in order to refer to the same solution.

$\mathcal{R}$	*Depth*	*Size*$_{\mathcal{R}}$	*Size*$_{\mathcal{R}_S}$	$T_{\mathcal{R}}$	$T_{\mathcal{R}_S}$	*Speedup*
	30	511,921	1,205	9,668	40	241.70
Blocks World	40	2,004,332	2,568	41,384	87	475.68
	50	5,841,540	4,689	139,198	171	814.02
	15	350,391	53,624	40,967	7,635	5.37
Containers	20	1,465,829	88,097	166,614	13,289	12.54
	25	4,172,116	122,538	549,430	19,518	28.15
	15	139,948	220	4,198	14	299.86
Dam Controller	30	2,271,930	505	82,845	30	2,761.50
	45	9,581,406	790	392,157	48	8,169.94
	4	196	23	8	2	4
Maze	6	133,225	78	59,553	9	6,617
	8	>3,154,238	303	>1,236,722	53	>23,334.38
	15	518,379	88,680	21,679	4,985	4.35
Space Invaders	20	1,797,799	268,115	120,930	19,609	6.17
	25	5,024,516	720,649	515,246	65,345	7.89
	40	648,965	8,585	35,776	1,508	23.72
Satellite	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 5.1: Experimental results of the safety enforcement technique.

$T_{\mathcal{R}}$ and $T_{\mathcal{R}_S}$ and are measured in milliseconds. We recorded the total speedup $T_{\mathcal{R}}/T_{\mathcal{R}_S}$ in Column *Speedup*.

We set a timeout of 60 minutes for the generation of the computation trees that is only overrun by the original program of the case of *Maze*. We use $>$ to denote the corresponding time measured at the moment the computation was halted due to timeout. The transformation time is negligible (less than 0.1 milliseconds) in all cases and is hence not included.

All speedup relations are positive, in some cases by several orders of magnitude, denoting the fact the safe versions of the program are notably faster at run time than their uncorrected counterparts. This may be, at a first glance, surprising, as our correction technique introduces a run time overhead as additional actions need to be taken in order to ensure the safety of the results w.r.t. the safety policy. However, the safe versions are faster because their search space is comparatively smaller, and this difference is big enough to offset the introduced run time overhead entirely. Given the fact that unsafe states are pruned iteratively and early, entire subtrees of the search space are preemptively removed as computations advance and need not to be explored, as safe states reached through an unsafe state are considered equally undesirable. The final result is that the search spaces are reduced notably, as evidenced by the columns *Size*$_{\mathcal{R}}$ and *Size*$_{\mathcal{R}_S}$.

The obtained results justify the claim that STRASS efficiently implements a mechanism for the run time enforcement of a safety policy. The posed programs exhibit typical features of Maude, showing the potentiality of our framework for synthesizing safe versions of complex, industrial size programs encoded in the Maude language, such as controllers of safety-critical systems.

CHAPTER 6
Conclusions

We have introduced an automatic program transformation technique that enables the efficient enforcement of invariants at run time in order to ensure the safety of a Maude program $\mathcal{R}$ w.r.t. a safety policy $\mathcal{A}$. We have defined a new language called MSPS that allows the specification of a safety policy $\mathcal{A}$ by means of state assertions and path strategies, covering the notions of safety both from the points of view of the admissibility of states and the admissibility of state transitions. We have also defined the correction technique as a composition of Maude strategy constructions which are then simply overlaid on the original program as an additional control layer which respects the separation of concerns principle. Finally, we have detailed the implementation of the STRASS tool and we have empirically assessed its efficiency and scalability.

6.1 Discussion of Results

The proposed framework leverages an existing Maude mechanism, the strategy language, to implement a safety enforcement mechanism elegantly: the frontier between the original source code and the automatically generated code is clear, as the transformed program simply extends the original one by appending an extra, separate module. The usage of a superimposed strategy module ensures a high degree of compatibility with many Maude features that have native, first-class support in the C++ implementation of the Maude interpreter. The use of a custom language dedicated to express the safety policy, MSPS, enables the user to quickly, expressively and easily convey what safety constraints are to be imposed on a system.

The original objective in the conception of STRASS and its underlying formal framework was to obtain a method that could enforce safety at run time with an acceptable overhead, and so performance has been a continuous, transversal focus though the entire research and development process. The experimental results show that STRASS not only avoids causing unacceptable slowdowns, but can in many cases result in performance gains as unsafe states are systematically pruned and left unexplored. This fact surpasses all our initial expectations, proving the potential of our technique in complex scenarios and opening new possibilities where safety enforcement could be used not only as a means to ensure correction, but as a method for performance optimization or improvement.

By choosing a dynamic technique, it is possible to specify and enforce the invariants of interest in an straightforward manner, this being especially useful and applicable in the singularly complex domain of concurrent and reactive systems that are often employed in safety-critical contexts. The scalability and ease of use of an equivalent static technique could be jeopardized by the underlying difficulty of proving certain complex properties for many states, which in some realistic scenarios may be still unfeasible in practice [19].

A series of extensive efforts on user experience have resulted in the development of a friendly, accessible web-based tool that enables non-expert Maude users to enjoy the benefits of run time safety enforcement for safety-critical contexts. Features such as detailed, high quality IDE-style error messages have required cross-cutting support in all of the involved components, and witness the fact that Maude is a language and system where not only complex tooling can be efficiently implemented, but it can also serve as the underlying core platform for user-facing software.

6.2 Related Work

We use the technique described in [8, 9] as our main frame of reference. The technique in question also enforce invariants for Maude programs at run time, but does so by essentially transforming the original rules into new conditional rules whose conditions ensure the satisfability of all of the provided assertions by means of new equationally-defined functions. Both this and our technique avoid the need to rely on an external system monitor, which could have a significant negative performance impact. This technique, however, only applies to *topmost* rewrite theories (i.e., programs where terms can only be rewritten at the root position), while our technique covers a wider range of admissible programs that are not necessarily topmost, and that optionally make use of advanced Maude language features previously unsupported, such as the existence of strategies in the input module. Our proposal is also conceptually simpler, leveraging the programmable Maude strategy language to avoid depending on ad-hoc, hard-coded safety checks that are intertwined with the existing code, leaving the underlying equational theory unchanged. The generated strategies are integrated alongside the original program but kept separate in its own module, which results ina benefit regarding code tracing and maintenance [21]. Also, this clear separation allows for incremental refinement to be built by simply appending new logical constraints, and ultimately to code that is easier to understand and maintain.

We also consider the technique presented in [21], which is the closest to our work since it defines a generic "strategy" to impose state invariants that can be expressed in different logics. As this technique predates the Maude strategy language, which is efficiently supported by the native C++ implementation of the Maude interpreter, this "strategy" is actually implemented as a parametric system module that follows the concept of an iterator of terms, thus effectively implementing the run time safety checks fully in Maude. Furthermore, safety enforcement in [21] is achieved by imposing an ad-hoc strategy (programmed in Maude) 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.

6.3 Future Work

The MSPS language enables the user to specify safety w.r.t. transitions by means of path strategies, which in turn contain a strategy expression. This, however, requires the user to be familiar with the basic constructs of the Maude strategy language to a certain degree. An additional, novel construct could significantly enhance the expressivity of the MSPS language further by denoting the safety of a transition from a state t to another state t' by means of a Boolean formula capable of reasoning with variables from both t and t' jointly. For instance, if a tank of water containing a volume V cannot lose more than 20 liters per transition, this can be naturally expressed as $V_t - V_{t'} \leq 20$. This approach builds upon the acquaintance with state assertions, which also use a Boolean formula, and removes the need for previous exposure to the Maude strategy language. Such a construct, however, would be unable to reason about transitions from the point of view of rule applications, and is hence considered complementary to path strategies.

The use of our technique in combination with static invariant enforcement techniques, and more generally, static software verification, is a challenging idea yet to be explored. Consider a safety policy detailing a series of invariants, some of which may be checked statically, e.g., by means of a model checker. In order to reduce the run time overhead derived from safety enforcement, the user could gradually attempt to statically check as many invariants from the given safety policy as possible, so that when a certain invariant is statically proven to be satisfied for all states, the safety policy is accordingly relaxed in the sense that the assertion is removed from the run time checks. This allows for a form of hybrid safety enforcement process where the user can incrementally improve performance as invariants are gradually proven statically. The STRASS tool could be extended to support such a process. This could be achieved by achieving the seamless integration with static verification tools together with a superset of MSPS containing annotations marking whether a state invariant is to be enforced statically or dynamically.

In conjunction with the raised lines of future work, it would be ultimately desirable to extend the MSPS language with temporal formulas that could be expressed in CTL or LTL. This would provide a more powerful framework for safety enforcement covering for a wide range of different scenarios and needs.

Bibliography

[1] AJAX in Mozilla Developer Network web documentation. URL https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX. Retrieved 2022-07-06.

[2] Bootstrap. URL https://getbootstrap.com/. Retrieved 2022-07-06.

[3] Mau-Dev: a developer extension of Maude. URL http://safe-tools.dsic.upv.es/maudev. Retrieved 2022-06-27.

[4] Monaco Editor. URL https://microsoft.github.io/monaco-editor/. Retrieved 2022-06-27.

[5] Svelte. URL https://svelte.dev/. Retrieved 2022-07-06.

[6] Apache Tomcat. URL https://tomcat.apache.org/. Retrieved 2022-07-06.

[7] TypeSscript: JavaScript with syntax for types. URL https://www.typescriptlang.org/. Retrieved 2022-07-06.

[8] M. Alpuente, D. Ballis, and J. Sapiña. Static Correction of Maude Programs with Assertions. *J. Syst. Softw.*, 153:64–85, 2019. doi: 10.1016/j.jss.2019.03.061. URL https://doi.org/10.1016/j.jss.2019.03.061.

[9] M. Alpuente, D. Ballis, and J. Sapiña. Imposing assertions in Maude via program transformation. *MethodsX*, 6:2577–2583, 2019. ISSN 2215-0161. doi: https://doi.org/10.1016/j.mex.2019.10.035.

[10] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A Declarative Debugger for Maude Functional Modules. *Electronic Notes in Theoretical Computer Science*, 238(3): 63–81. doi: 10.1016/j.entcs.2009.05.013.

[11] A. Cesta, G. Cortellessa, S. Fratini, and A. Oddi. MrSPOCK - Steps in Developing an End-to-End Space Application. *Computational Intelligence*, 27:83–102, 02 2011. doi: 10.1111/j.1467-8640.2010.00373.x.

[12] M. Cialdea Mayer, A. Orlandini, and A. Umbrico. Planning and Execution with Flexible Timelines: A Formal Account. *Acta Informatica*, 53, oct 2016. doi: 10.1007/s00236-015-0252-z.

[13] E. M. Clarke. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.

[14] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: A Tutorial. *J. UCS*, 12:1618–1650, Jan. 2006.

[15] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott. *Maude Manual (Version 3.2.1)*, Feb 2022. URL http://maude.lcc.uma.es/maude32-manual-html/maude-manual.html.

[16] F. Durán and J. Meseguer. On the Church-Rosser and Coherence Properties of Conditional Order-Sorted Rewrite Theories. *The Journal of Logic and Algebraic Programming*, 81(7):816–850, 2012. doi: 10.1016/j.jlap.2011.12.004. Rewriting Logic and its Applications.

[17] S. Escobar, C. Meadows, and J. Meseguer. Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, pages 1–50, Jan. 2007. doi: 10.1007/978-3-642-03829-7_1.

[18] ISO/IEC 14977:1996. Information technology — Syntactic metalanguage — Extended BNF, 1996.

[19] E. Ledinot, J.-P. Blanquart, J.-M. Astruc, P. Baufreton, J.-L. Boulanger, C. Comar, H. Delseny, J. Gassino, M. Leeman, P. Quéré, et al. Joint use of static and dynamic software verification techniques: a cross-domain view in safety critical system industries. In *Embedded Real Time Software and Systems (ERTS2014)*, 2014.

[20] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, Apr 1992. doi: 10.1016/0304-3975(92)90182-f.

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

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

[23] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Model Checking Strategy-Controlled Systems in Rewriting Logic. *Automated Software Engineering*, 29(1), Dec 2021. doi: 10.1007/s10515-021-00307-9.

[24] A. Tanaka, R. Affeldt, and J. Garrigue. Safe Low-level Code Generation in Coq Using Monomorphization and Monadification. *Journal of Information Processing*, 26:54–72, 2018. doi: 10.2197/ipsjjip.26.54.