DM-CHECK

DM-Check User Manual

A Deductive Model Checker for Maude Rewrite Theories

K. Bae[a], S. Escobar[b], R. López-Rueda[b], and J. Meseguer[c]

[a]Pohang University of Science and Technology, Pohang, South Korea
[b]VRAIN, Universitat Politècnica de València
[c]University of Illinois Urbana-Champaign

Public Alpha 12
Manual revision: October 12, 2026

About This Manual

This manual documents the session-oriented interface of DM-Check, a deductive model checker for invariant verification of Maude rewrite theories. It is written for two complementary uses. New users can follow the installation and first-session chapters in sequence; experienced users can use the command reference, troubleshooting guide, and case studies independently.

The manual separates four kinds of material that should remain distinct in a reproducible verification development: the Maude specification, the symbolic sets used as initial states or invariants, the DM-Check interactive command transcript, and the NuITP proofs of generated obligations. The examples are organized around that separation.

Acknowledgements

The methodology and case studies documented here build on the joint DM-Check research publications, including work with Julia Sapiña on the interaction between DM-Check and NuITP.

Generative AI and AI-assisted tools

During the preparation of this manual, the authors used generative AI tools for language editing and structural revision. All generated suggestions were reviewed and edited by the authors, who take full responsibility for the final content.

License and Copyright

Copyright © 2023–2026 Universitat Politècnica de València, Spain.

DM-Check is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or, at your option, any later version.

DM-Check is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

See the GNU General Public License at: https://www.gnu.org/licenses/

The copyright statement on this page refers to the manual. The startup banner printed by the tool refers to the implementation and may use a different initial year.

Contents

About This Manual i

List of Tables vii

List of Listings viii

I Getting Started 1

1 Overview 2
1.1 Purpose and Scope . . . 2
1.2 Main Objective . . . 2
1.3 Limitations . . . 2
1.4 Intended Audience . . . 3
1.5 Official Web Resources . . . 3
1.6 How to Cite DM-Check . . . 3
1.7 A Typical Verification Session . . . 4
1.8 How to Read the Manual . . . 4

2 Installation and First Session 4
2.1 Supported Release and Requirements . . . 4
2.2 Recommended Project Layout . . . 5
2.3 Recording Interactive Sessions . . . 5
2.4 Loading the Tool . . . 6
2.5 Typing Commands . . . 7
2.6 Pre-flight Checklist . . . 7
2.7 A First Verification Session . . . 7

3 The Input Language 9
3.1 Rewrite Theories . . . 9
3.2 Admissibility Assumptions and Supported Specification Class . . . 10
3.3 Variables and Sorts . . . 11
3.4 Constructors and Defined Symbols . . . 11
3.5 Structural Axioms . . . 12
3.6 Constrained Patterns . . . 12
3.7 Constraints . . . 12
3.8 Initial States, Invariants, and Error Patterns . . . 13
3.9 Candidate Invariants . . . 13
3.10 Error Patterns . . . 14
3.11 Auxiliary Functions in Constraints . . . 14

II Verification Guide 15

4 Methodological Background 16
4.1 Rewrite Theories as Transition Systems . . . 16
4.2 Constrained Narrowing . . . 16
4.3 One-Step Closure . . . 16
4.4 Subsumption . . . 17
4.5 Intersection . . . 17

4.6 Emptiness . . . 17
4.7 Where Proof Obligations Come From . . . 18
4.8 Semantic Transformations . . . 18
4.9 Invariant Generation as a Fixpoint . . . 20

5 Verification Workflows **20**
5.1 Checking a Given Invariant . . . 21
5.2 Proving Safety Properties . . . 21
5.3 Generating an Invariant . . . 22
5.4 When to Generalize . . . 22
5.5 Classifying the Status of a Result . . . 22
5.6 Maintaining a Reproducible Proof . . . 22

6 Working with NuITP **23**
6.1 Why NuITP Is Needed . . . 23
6.2 How Obligations Should Be Read . . . 23
6.3 NuITP Proof Material . . . 24
6.4 Recommended Discipline . . . 24
6.5 Designing Useful Lemmas . . . 24
6.6 Keeping DM-Check and NuITP Material Consistent . . . 25

III Reference **26**

7 Understanding DM-Check Output and Proof Trees **27**
7.1 What a Proof Tree Represents . . . 27
7.2 Message Classes . . . 28
7.3 Node, Parent, and Match Identifiers . . . 28
7.4 Conventions in Tool-Generated Output . . . 29
7.5 Paths from the Initial Pattern . . . 29
7.6 Reading the Main Outcomes . . . 31
7.6.1 Successful closure . . . 31
7.6.2 Open implication . . . 32
7.6.3 No matcher . . . 32
7.6.4 Intersection result . . . 32
7.7 A Diagnostic Decision Sequence . . . 34

8 Command Reference **34**
8.1 Command Summary . . . 34
8.2 Session and Information Commands . . . 36
8.2.1 `set module` . . . 36
8.2.2 `help` and `h` . . . 37
8.2.3 `load` and `save proof` . . . 37
8.2.4 `show module`, `show last`, `show lemmas`, and `show obligations` . . . 38
8.2.5 `show path` and `show path states` . . . 39
8.2.6 `reset lemmas` . . . 39
8.2.7 `quit` and `q` . . . 40
8.3 Verification Commands . . . 40
8.3.1 `check ind-invariant` . . . 40
8.3.2 `check ... subsumed by ...` . . . 41
8.3.3 `intersect ... with ...` . . . 41
8.4 Invariant-Generation Commands . . . 42

8.4.1 `generate ind-invariant` and `gen ind-invariant` . . . 42
8.4.2 `generate next level` and `gen next level` . . . 42
8.5 Lemmas and Proof-Tree Closure Commands . . . 43
8.5.1 `add lemma` . . . 43
8.5.2 `empty` . . . 44
8.5.3 `lemma` . . . 44
8.5.4 `contained` . . . 44
8.6 Pattern-Transformation Commands . . . 44
8.6.1 Constructor case splitting . . . 45
8.6.2 Variable abstraction . . . 45
8.6.3 Explicit substitution . . . 46
8.6.4 Variant unification . . . 46
8.6.5 Narrowing an auxiliary term . . . 47
8.6.6 Generalization . . . 47
8.6.7 Boolean splitting . . . 47
8.7 Deprecated Command: `recheck` . . . 48

9 Examples and Case Studies 48
9.1 Readers–Writers . . . 48
9.2 Alternating Bit Protocol . . . 49
9.3 QLOCK . . . 51
9.4 Lamport's Bakery Protocol . . . 54
9.4.1 Starting from the initial states . . . 55
9.4.2 Closing the generated frontier . . . 56
9.4.3 Discharging the obligations in NuITP . . . 58
9.4.4 Derived ticket invariant and mutual exclusion . . . 59
9.5 Other Examples . . . 59

IV Troubleshooting 60

10 Troubleshooting and Limitations 61
10.1 No Active Module . . . 61
10.2 The Module Is Not Activated . . . 61
10.3 A Command Using == Is Rejected . . . 61
10.4 NuITP Could Not Process a Formula . . . 61
10.5 The Command Produced an Ill-formed Result . . . 61
10.6 Unification May Be Incomplete . . . 61
10.7 A Script Was Aborted . . . 62
10.8 A Case Split Adds Coverage Obligations . . . 62
10.9 The Invariant Is Not Proved . . . 62
10.10 Subsumption Fails . . . 62
10.11 The Parser Rejects a Pattern . . . 62
10.12 A Rule Is Not Used . . . 62
10.13 Invariant Generation Does Not Terminate . . . 63
10.14 The Generated Invariant Is Too Weak . . . 63
10.15 The Deprecated `recheck` Command Is Used . . . 63
10.16 The Generated Invariant Is Too Strong . . . 63

V Appendices 64

A Quick Reference **65**
A.1 Minimal Checking Workflow . . . 65
A.2 Minimal Generation Workflow . . . 65
A.3 Questions to Ask When a Proof Stops . . . 65
A.4 Reproducibility Checklist . . . 65

B Legacy Syntax and Migration **66**
B.1 Module-Qualified Legacy Commands . . . 66
B.2 Accepted Aliases and Deprecated Spellings . . . 67
B.3 Migration Rules . . . 68

References **70**

List of Tables

1 Official web resources related to DM-Check. 3
2 Main layers of a DM-Check verification project. 4
3 Main symbolic sets used in a DM-Check verification. 13
4 Logical proof conditions and their principal DM-Check commands. 16
5 Main semantic transformation commands. 19
6 Information that should be retained from a proof-tree node. 29

Listings

1 Recommended organization of a DM-Check project. 5
2 Loading a specification and the DM-Check. 6
3 Syntax for loading a specification and the DM-Check. 6
4 DM-Check startup banner. 6
5 Selecting the Readers–Writers module. 7
6 Checking the Readers–Writers inductive invariant. 8
7 Checking symbolic containment by subsumption. 8
8 Readers–Writers module. 48
9 Readers–Writers invariant check. 49
10 Readers–Writers containment check. 49
11 ABP invariant check. 50
12 Representative ABP open obligation. 50
13 ABP lemma added to the DM-Check session. 50
14 QLOCK invariant check. 51
15 QLOCK initial containment. 51
16 QLOCK deadlock-freedom query before transformations. 52
17 QLOCK final transformation sequence. 53

Part I:
Getting Started

1 Overview

DM-Check is a deductive model checker for invariant verification of systems specified as rewrite theories in Maude [1, 2]. It reasons symbolically over sets of states described by constrained patterns, computes one-step successors by constrained narrowing, and uses NuITP to discharge the proof obligations that arise from constraints, emptiness checks, subsumption, and auxiliary semantic arguments.

The tool is intended for systems where explicit-state exploration is either impossible or insufficient: parametric protocols, concurrent algorithms with an unbounded number of components, systems with counters or queues, and specifications whose reachable state space is infinite but has a finite symbolic description.

1.1 Purpose and Scope

The purpose of this manual is to explain how to prepare a rewrite theory for symbolic analysis, how to check or generate an inductive invariant, how to interpret open proof nodes, and how to discharge the resulting semantic obligations in NuITP. It does not attempt to reproduce the complete literature of constrained narrowing or inductive theorem proving; the cited papers and manuals remain the authoritative theoretical references.

1.2 Main Objective

The main property handled by DM-Check is invariant analysis. Given a set A of initial states and a state predicate Q, the goal is to prove that every state reachable from A satisfies Q. In the tool this is usually done by finding or checking an inductive invariant B:

$$A \subseteq B \qquad \text{and} \qquad R[B] \subseteq B,$$

where $R[B]$ denotes the set of one-step successors of states in B. Once B has been established, other safety properties can be proved by showing that B is contained in the desired property or has empty intersection with the error states.

An invariant-generation workflow for DM-Check is presented in [3]. Starting from the initial symbolic states, DM-Check can compute successive symbolic frontiers, fold states already covered by the accumulated approximation, and help the user transform or generalize frontier patterns until a fixpoint is reached.

1.3 Limitations

DM-Check is not an explicit-state model checker, and it should not be used as if it were one. It does not enumerate all concrete states reachable from an initial state, and it does not return a finite transition graph unless such a graph is already implicit in the symbolic representation. Its input language is also not a temporal-logic property language. The workflow described in this manual focuses on invariant and safety properties expressed as symbolic sets of states.

It is also useful to distinguish DM-Check from a fully automatic invariant synthesizer. The tool can generate symbolic levels and can help the user reach a finite fixpoint, but the decisive abstraction step may still require human insight. This is especially common when counters grow without bound, when several frontier patterns must be represented by one more general pattern, or when the proof depends on a semantic property of an auxiliary function.

The intended use of the tool is therefore interactive and deductive. DM-Check automates the routine symbolic exploration and bookkeeping. NuITP handles inductive proof obligations generated by DM-Check. The user supplies the specification, chooses the relevant symbolic descriptions, and provides lemmas or generalizations when the automatic part reaches its limit.

In addition to the unconditional systems considered in the original works [1, 2, 3], the current implementation supports conditional rewrite rules whose conditions are conjunctions of equations. Rule conditions and constrained-pattern constraints use this equational form. The semantic obligations generated by DM-Check are implications whose antecedents and conclusions are built from such equational constraints; these are the facts needed to justify folding, emptiness, or containment steps. The distributed `qlock-cond` example exercises this conditional-rule support.

Two additional boundaries are important. First, if Maude reports that a set of narrowing steps or unifiers may be incomplete—for example, for some problems modulo an associative operator without commutativity—DM-Check rejects the command instead of drawing a conclusion from a partial result. Second, like NuITP, DM-Check treats the operators declared with the `ctor` attribute as exactly the constructors of the data algebra when it checks constructor-case coverage. An incorrect `ctor` declaration can therefore invalidate the coverage argument, so the specification must identify its constructors faithfully.

1.4 Intended Audience

This manual assumes that the reader is familiar with basic Maude syntax: modules, sorts and subsorts, operator declarations, equations, and rewrite rules. Readers who need a systematic introduction to Maude should consult the Maude manual [4].

For non-trivial case studies, the user should also be prepared to read and prove NuITP obligations. DM-Check can call NuITP automatically, but difficult proofs often require auxiliary lemmas or an interactive inductive proof. Readers who are not familiar with NuITP should consult the NuITP manual [5].

1.5 Official Web Resources

The papers and manuals cited in this document are the primary references for the theoretical and methodological aspects of DM-Check. For installation files, public examples, and tool-specific updates, users should consult the official web resources listed in Table 1.

Table 1: *Official web resources related to* DM-Check.

Resource	Use
DM-Check [6]	Tool distribution, installation instructions, user manual, command summary, and public examples.
NuITP [7]	NuITP releases, manual, examples, and download information.
Maude [8]	Maude downloads, documentation, examples, publications, and related tools.

1.6 How to Cite DM-Check

Publications that use DM-Check should cite the main journal article for the invariant-verification methodology [2]. Publications using the invariant-generation workflow should also cite the WRLA 2026 paper [3]. When reporting experiments, include the DM-Check release, the Maude banner, the NuITP release, and the command transcript used to reproduce the result.

1.7 A Typical Verification Session

A complete DM-Check development usually has four layers. The first layer is the Maude specification: the user defines the state sort, constructors, equations, auxiliary functions, and transition rules. The second layer is the symbolic description of the relevant state sets: initial states, candidate invariants, generated frontiers, target properties, or error patterns. The third layer is the DM-Check command transcript, which records the interactive commands used to check or generate the invariant. The fourth layer is the NuITP proof material needed for obligations that cannot be discharged automatically.

Table 2: *Main layers of a* DM-Check *verification project.*

Stage	Layer	Contents
1	Maude specification	Constructors, equations, rewrite rules, and auxiliary functions.
2	Symbolic sets	Initial patterns, candidate invariants, generated frontiers, and error patterns.
3	DM-Check transcript	Interactive commands for containment, closure, intersection, invariant generation, and symbolic transformations.
4	NuITP proofs	Lemmas and obligations about constraints and auxiliary functions.

Keeping these layers separate is good practice. It makes it easier to replay a verification session, to identify which proof obligations belong to the invariant itself, and to distinguish tool interaction from the mathematical argument being carried out.

1.8 How to Read the Manual

Part I is the shortest route from installation to a first verification session. Part II explains the mathematical and deductive workflow, including the interaction with NuITP. Part III contains the output and command reference as well as representative case studies. Part IV provides troubleshooting guidance, and Part V contains the appendices.

Readers performing their first verification should read Chapters 2 and 3 before moving to the workflow in Chapter 5. Readers who already have a recorded command transcript can go directly to Chapters 7 and 8.

2 Installation and First Session

This section introduces the first basic concepts. In Section 2.1, we show the requirements for executing DM-Check. In Section 2.2, we present the contents of the downloaded file. In Section 2.3, the interactive session is discussed. In Section 2.4, we present the first steps for loading a file and, in Section 2.5, we present how other commands should be entered. In Section 2.6, we discuss the different assumptions and checks that a user must take into account. And Section 2.7 shows an example of an interactive verification session.

2.1 Supported Release and Requirements

This manual documents the public release indicated on the cover. DM-Check runs on top of Maude and uses NuITP as its inductive theorem-proving component.

The DM-Check download contains two separate files: `dm-check.maude`, which contains the implementation and session-oriented interface of the model checker, and `NuITP.maude`, which contains the compatible NuITP version required by that DM-Check release. No separate NuITP installation is needed; keep both files in the same directory when loading

DM-Check.

Use the Maude version identified in the official DM-Check installation instructions. Since Maude builds and tool releases are identified by their own banners and commit information, the exact Maude and DM-Check banners should be kept in the session transcript. For details about NuITP proof-engine usage, consult the latest NuITP manual available for the version distributed with DM-Check.

The officially distributed Maude binaries support the following platforms and architectures [8]:

- Linux on `x86_64`;
- macOS on Intel `x86_64`;
- macOS on Apple Silicon `ARM64`.

To use it on Windows, the recommended configuration is to run the Linux distribution of Maude through the Windows Subsystem for Linux (WSL).

No separate `dm-check-ui.maude` file is required in this release. Historical distributions and legacy loading procedures are discussed only in Appendix B.

2.2 Recommended Project Layout

For a reproducible development, keep the system specification, the DM-Check interaction transcript, and the NuITP proof material in separate files:

```
tools/
  dm-check.maude
  NuITP.maude
theories/
  example/
    example.maude
    example.dmc
    dmcheck-transcript.txt
    example.nuitp
    notes/
      commands.txt
```

Listing 1: *Recommended organization of a DM-Check project.*

The DM-Check transcript is part of the verification record. It identifies the release used, the proof-tree node numbers, the substitutions printed by the tool, and the obligations that were still open at the end of the session.

2.3 Recording Interactive Sessions

DM-Check is used through an interactive session, but command sequences can be replayed with `load <file>` and the accepted state-changing commands of a session can be written to a script with `save proof <file> .`. The `load` command is entered without a final period, whereas `save proof` is terminated by a period. Both require Maude to be started with `-allow-files` (or `-trust`).

A `.dmc` script may contain blank lines, line comments beginning with `#`, `--`, or `***`, and nested block comments of the form `***( ... )`. A backslash at the end of a physical line continues the same command on the next line. A script cannot load another script. Execution stops at the first Error; a Warning records that the command was applied with a condition that should be reviewed and does not stop the script.

The script produced by `save proof` is a flat replay script. It contains the accepted commands that changed the proof state, including commands originally read by `load`, but omits inspection, help, exit, load, and save commands. It does not contain the computed results, certify the proof, or prove any lemma introduced by `add lemma`. Consequently, the unedited console transcript remains the authoritative verification record, while the `.dmc` file is the preferred replay artifact.

For auditability, the transcript should include the tool banners, the active module, all commands used in the verification, the generated node and match identifiers, and the final list of proof obligations. When the proof depends on node identifiers, match identifiers, or generated obligations, the transcript should be kept with the corresponding Maude specification and external NuITP proof material.

2.4 Loading the Tool

A typical session first loads the user specification and then the DM-Check interface. The file `NuITP.maude` distributed with DM-Check should be in the same directory as `dm-check.maude`. When the corresponding files are located in the current working directory, start Maude with file operations enabled when scripts will be loaded or saved:

```
maude -allow-files
```

The loading commands then have the following form:

```
Maude> load example
Maude> load dm-check
```

Listing 2: *Loading a specification and the DM-Check.*

Alternatively, explicit paths and file extensions may be provided:

```
Maude> load <path-to-specification>/example.maude
Maude> load <path-to-tools>/dm-check.maude
```

Listing 3: *Syntax for loading a specification and the DM-Check.*

After loading `dm-check.maude`, the following banner is displayed:

```
Maude> load dm-check

    ============================================
                     DM-Check
              Deductive Model Checker
            for Maude Rewrite Theories

       (public alpha 12 built Oct 9th 2026)
    ============================================
                Copyright 2023-2026
        Universitat Politècnica de València
    ============================================
```

Listing 4: *DM-Check startup banner.*

DM-Check does not use a separate user-interface file. The implementation and session-oriented interface are both contained in one single file `dm-check.maude`.

After the interface is loaded, select the module to be analyzed:

```
set module <module-name> .
```

For example:

```
set module R&W .
```

Changing the active module resets the current verification session. Stored lemmas, generated proof obligations, and the last proof tree are cleared, so the module should normally be selected before issuing any verification or invariant-generation command.

The command `q` returns from DM-Check to the Maude prompt. Another specification may then be loaded and `dm-check.maude` loaded again in the same Maude process. The compatible `NuITP.maude` file restores the default setting that excludes the SMT `BOOLEAN` module from subsequently loaded specifications. Consequently, a standalone `.nuitp` proof is no longer sensitive to whether the specification or `NuITP.maude` is loaded first; the examples in this manual load the specification first for consistency.

2.5 Typing Commands

Most DM-Check commands must be terminated by a period. The exceptions are `load <file>` and `quit`/`q`. Long commands may be split across several physical input lines by placing a backslash (\) at the end of each incomplete line:

```
check <initial-patterns> \
subsumed by \
<candidate-invariant> .
```

The final line contains the terminating period and does not require a trailing backslash. Line continuation affects only how the command is entered: the complete multiline input is parsed as a single DM-Check command.

2.6 Pre-flight Checklist

Before drawing conclusions from a failed session, verify the following conditions:

- the specification module has been loaded in Maude before using DM-Check;
- the intended module has been selected with `set module`;
- all unconditional transition rules intended for symbolic exploration are marked with `[narrowing]`;
- operator declarations used by NuITP satisfy the `rpo` requirements of the version distributed with DM-Check;
- variables in commands have explicit sort annotations when parsing is ambiguous;
- the `NuITP.maude` and `dm-check.maude` files distributed with DM-Check are kept in the same directory.

2.7 A First Verification Session

The Readers–Writers example, revisited in Section 9.1, provides a compact introductory verification session. It illustrates how to select a verification module, check that a symbolic set is an inductive invariant, and establish a subsequent containment property by subsumption.

After loading the Readers–Writers specification, activate its module with `set module`:

```
DM-Check> set module R&W .

  Module R&W is now active.
```

Listing 5: *Selecting the Readers–Writers module.*

Selecting the module initializes the verification session and makes its equations, rules, sorts, and operators available to the subsequent DM-Check commands.

The following symbolic set describes states with either no active writer or exactly one active writer and no active reader:

$$B = (\langle N, 0\rangle \mid true) \ \vee \ (\langle 0, s(0)\rangle \mid true).$$

Its transition closure is checked with `check ind-invariant`:

```
DM-Check> check ind-invariant \
>      < N:Natural , 0 > | true \
>   \/ < 0 , s(0) > | true .

  Invariant satisfied.
```

Listing 6: *Checking the Readers–Writers inductive invariant.*

The output message "`Invariant satisfied.`" means that every symbolic one-step successor of a ground state denoted by the supplied disjunction is again denoted by the same disjunction. Therefore,

$$R[B] \subseteq B.$$

A second command checks whether the established invariant is contained in another symbolic set Q:

```
DM-Check> check \
>      < N:Natural , 0 > | true \
>   \/ < 0 , s(0) > | true \
>  subsumed by \
>      < 0 , 0 > | true \
>   \/ < R:Natural , s(W:Natural) > | true \
>   \/ < R:Natural , 0 > | true \
>   \/ < s(R:Natural) , W:Natural > | true .

  Subsumption satisfied.
```

Listing 7: *Checking symbolic containment by subsumption.*

The output message "`Subsumption satisfied.`" states that every ground state denoted by B is also denoted by Q. Hence,

$$B \subseteq Q.$$

This is a symbolic containment result; it does not perform an additional transition-closure check.

Thus, this first session establishes the following two facts:

$$R[B] \subseteq B \qquad \text{and} \qquad B \subseteq Q.$$

However, to obtain a complete invariant proof from a designated set of initial states A, the verification must additionally establish:

$$A \subseteq B.$$

Alternatively, when the safety property is described negatively by an error set *Error*, it may be derived by proving:

$$B \cap Error = \varnothing.$$

3 The Input Language

This section illustrates the different inputs given to the tool. In Section 3.1, we explain the system specification given as a Maude rewrite theory but, in Section 3.2, we describe the different assumptions and conditions that such rewrite theory must satisfy. In Section 3.3, we warn the user about variables and their sorts and subsorts whereas, in Section 3.4, we explain the relevance of using constructor symbols for data and defined function symbols for auxiliary functions or properties, both described using equations. In Section 3.5, we focus on structural axioms such as associativity, commutativity and identity. In Section 3.6, we explain the concept of *constrained pattern* as a state pattern using only constructor symbols modulo axioms but with an attached constraint that restrict the possible instances of the state pattern but, in Section 3.7, we give extra comments about constraints. In Section 3.8, we present the more dynamic view of providing invariants to be analyzed whereas, in Section 3.9, we discuss about how relevant an invariant may be and, in Section 3.10, we discuss symbolic patterns as a way to described the negative of an invariant. Finally, in Section 3.11, we talk about auxiliary functions necessary to express properties.

3.1 Rewrite Theories

DM-Check analyzes Maude system modules that describe rewrite theories. A suitable module should make explicit:

- the sort of system states;
- the constructor operators used to build canonical states;
- the structural axioms, such as associativity, commutativity, or identity;
- the unconditional transition rules, marked with the `[narrowing]` attribute, and any supported conditional rules;
- auxiliary functions and predicates used in constraints.

Unconditional rules used by DM-Check are written as ordinary Maude rewrite rules and must be made available to the symbolic engine with the `[narrowing]` attribute. For example, Readers—Writers from Section 9.1 contains the following rule:

```
rl [w-in]: [N]< 0,0 >[0 | N]
        => [N]< 0,1 >[0 | N] [narrowing] .
```

Conditional rewrite rules are also supported, provided that their conditions are conjunctions of equations written using syntax accepted by NuITP. Their general form is:

```
crl [label] : <left-hand side> => <right-hand side>
  if <term-1> = <term-1'> /\ <term-2> = <term-2'> .
```

Supported conditional rules contribute their equational conditions to the constraints of the generated symbolic successors, including when the rule is applied below the root of the selected term. Conditional rules with matching, rewrite, or membership conditions (`:=`, `=>`, or `:`) are not silently ignored: `set module` rejects the module. The obligations delegated to NuITP are implications whose antecedents and conclusions are built from equational constraints.

Unconditional rules without the `[narrowing]` attribute are ignored by DM-Check during symbolic exploration. When the active module contains such rules, the tool prints a warning, since they do not contribute transitions to the symbolic proof.

Equations intended to be used by Maude's variant machinery must belong to an equational subtheory with the finite variant property and must be declared with the `[variant]` attribute

when they are meant to participate in variant unification. Equations that should not be considered by variant unification should deliberately be left outside that variant subtheory.

Since DM-Check relies on NuITP to discharge deductive obligations, modules used in those proofs should follow the `rpo` requirements of the installed NuITP release. All examples distributed through the official DM-Check web site include the corresponding `rpo` declarations. Older specifications may therefore need to update their operator declarations before being used with the current tool distribution. The exact policy should follow the latest NuITP manual: precedence values are module-dependent and should not be assigned mechanically.

3.2 Admissibility Assumptions and Supported Specification Class

The theoretical framework behind DM-Check is not the class of all executable Maude modules. It assumes a rewrite theory whose equational part admits canonical ground representatives, a constructor subtheory for states, and a state sort at the top of a connected component of sorts [2, 3]. For manual use, the following operational checklist is usually the most useful form of these assumptions:

- system states should be represented by constructor terms of the chosen state sort;
- rewrite rules should be topmost, rewriting complete states rather than arbitrary state subterms;
- the equations of the data and auxiliary functions should provide unique canonical forms for ground terms, modulo the declared structural axioms;
- the terms in constrained patterns and the right-hand sides of rewrite rules must use only constructors and symbols defined by `[variant]` equations; all other defined functions and predicates must be placed in the constraint;
- all unconditional transition rules intended for symbolic exploration must be marked with `[narrowing]`; supported conditional rules are transformed internally;
- functions used in constraints and proof obligations must satisfy the requirements of the NuITP proof backend, including the `rpo` requirements documented in the latest NuITP manual.

The structural axioms declared in the Maude module, such as associativity, commutativity, and identity, are taken into account by matching and unification where supported. They are part of the symbolic proof problem, not merely a pretty-printing convention.

Before activating a module, DM-Check examines its flattened definition, including imported statements. It rejects a module that uses the predefined `_==_` or `_=/=_` in an equation, membership, rule, or condition; contains an `owise` equation; or contains a conditional rewrite rule with a non-equational condition. On symbolic terms, the predefined equality operators and `owise` equations may select cases syntactically and thereby lose states. A user-defined equality predicate specified by `[variant]` equations is admissible; the `.=.` predicate of Token Ring is an example. Because the prelude modules `SET`, `MAP`, and `LIST` contain such unsupported statements, modules importing them cannot be activated directly. The same flattened-module check rejects membership axioms declared with `mb` or `cmb`, including imported axioms. Maude's unification ignores membership axioms, so narrowing in their presence can lose states. If the axiom only declares the result sort of an operator, declare that sort in the operator signature instead: `mb f(X:S) : S'` is represented by `op f : S -> S'`. Otherwise, express the membership test by a predicate in the constraint. If activation is rejected, the previously active module remains selected and the user may issue another `set module` command without restarting the session.

For compatibility with the NuITP backend, user specifications intended for DM-Check should begin with `set include BOOL off .`, before their module declarations, as all distributed examples do. If the prelude module `BOOL` is included and the active module contains an operator with an `id:` attribute, NuITP may be unable to process even the implication `true -> true`; DM-Check then reports that NuITP could not process the formula and leaves the command unapplied.

Variant unification and constrained narrowing also depend on Maude being able to compute a complete result set. When Maude reports an incomplete unification or narrowing result, DM-Check rejects the affected command and restores the preceding proof state. Reformulating the specification—for example, by using a suitable `assoc id` or `assoc comm` representation— is then necessary.

3.3 Variables and Sorts

Variables must be written with explicit sort annotations:

```
N:Nat
NZ:NzNat
PS:ProcSet
WS:ProcWaitSet
```

Explicit annotations are needed for the tool to work, and also help the user read proof obligations. This is especially important in examples with several related subsorts, such as idle processes, waiting processes, non-empty process sets, or non-zero natural numbers.

3.4 Constructors and Defined Symbols

The distinction between constructors and defined symbols is important for symbolic verification. Constructors describe the canonical shape of states and data values. Defined symbols compute auxiliary information used in constraints. It is a requirement that the terms in constrained patterns and the right-hand sides of rewrite rules contain only constructors and symbols whose defining equations have the `[variant]` attribute. Any other defined symbol must be moved to the constraint. For example, `< f(M) > | true` must be written as `< Z > | Z = f(M)` when `f` is not part of the variant subtheory. If this requirement is violated, successor or intersection computations may lose cases and can incorrectly report `Invariant satisfied` or `No intersection`.

Like NuITP, DM-Check treats the operators declared with the `ctor` attribute as exactly the constructors when it checks whether a `case` generator set covers a sort. An incorrect declaration can therefore make the coverage check unsound. Constructor attributes must describe the canonical data algebra accurately. After `set module`, the tool warns about user sorts for which it finds no constructor result; the module is nevertheless activated, but a case split on such a sort will leave coverage obligations.

For example, in the Bakery case study, fully described in Section 9.4, process configurations are represented by constructors for idle, waiting, and critical processes. Functions such as `tickets` and `i` summarize ticket information. This separation keeps the symbolic state readable:

```
(N:iNat | M:iNat | WS:ProcWaitSet ; CS:ProcCritSet) |
    tickets(WS:ProcWaitSet ; CS:ProcCritSet) = i(M:iNat, N:iNat)
```

The precise names and sorts depend on the module, but the requirement is general: put the constructor and variant-defined structural part of the state in the pattern, and put other defined functions and semantic side conditions in the constraint.

3.5 Structural Axioms

DM-Check works with the structural axioms declared in the Maude specification, such as associativity, commutativity, and identity. This is essential in examples involving sets, multisets, queues, or arithmetic-like structures. A pattern may match another pattern modulo these axioms even when the textual order of components differs.

This feature is powerful but requires care. If an operator is declared associative-commutative, the user should not rely on the printed order of its arguments. If an identity element is present, patterns may match in ways that are not obvious from the concrete syntax. When a proof behaves unexpectedly, inspect the generated matchers and substitutions rather than only the printed pattern.

3.6 Constrained Patterns

A constrained pattern has the form:

$$u \mid \varphi$$

where u is a symbolic state pattern and φ is a constraint. It denotes all ground constructor instances of u that satisfy φ. In DM-Check syntax, a constrained pattern is written as:

```
(<state-pattern>) | <constraint>
```

The constraint `true` is used when no side condition is needed:

```
([N:Nat + 1]< 0, 1 >[0 | N:Nat + 1]) | true
```

Finite unions of symbolic state sets are represented as disjunctions of constrained patterns. The disjunction symbol is written as `\/`:

```
(<pattern-1>) | <constraint-1> \/ (<pattern-2>) | <constraint-2>
```

A constrained pattern should be read as the set of all its ground instances. For example:

```
([N:Nat + 1]< 0, 1 >[0 | N:Nat + 1]) | true
```

does not describe one state. It describes all states of that shape, for all ground values of `N:Nat`. If `Nat` is unbounded, this is an infinite set of states. The constraint may restrict the possible instances:

```
(M:iNat | N:iNat | IS:ProcIdleSet) | N:iNat ~ M:iNat = True
```

Here the state pattern has two variables, but the constraint restricts them to instances satisfying the relation `~`. This style is often preferable to using repeated variables [1] directly in the pattern, because it makes the equality or relation explicit as a proof obligation.

3.7 Constraints

Constraints are interpreted in the initial algebra of the Maude specification. In the current interface, the constraint fragment used by DM-Check consists of conjunctions of equations. In the examples these equations typically involve Boolean predicates, membership-like predicates, interval relationships, or auxiliary functions.

The constants `true` and `false` may be written directly on either side of an equation in a constraint. When one side is otherwise ambiguous between the Boolean sorts used by Maude and NuITP, DM-Check determines its sort from the other side. Thus both `N:Nat < 300 =`

[1] If two occurrences of the same variable appear in a pattern, they must be instantiated by the same ground term. If two different variables appear in a pattern and are related by a constraint, the equality or relation becomes part of the semantic condition that DM-Check may have to prove.

`false` and `false = N:Nat < 300` are accepted; a qualification such as `(false).Bool` is not required. Examples from this manual include:

```
true
N:iNat ~ M:iNat = True
allN(NS:iSet, N:iNat) = True
tickets(WS:ProcWaitSet) = i(M:iNat, N:iNat)
tt = set(NeS:NeMSet) /\ M:MSet = l2ms(L:List) /\ NeS:NeMSet = N:Nat M:MSet
```

A constraint should be executable enough for Maude and NuITP to reason about it, but not so low-level that the invariant becomes unreadable. If a constraint becomes very long, consider introducing an auxiliary predicate in the Maude module and proving lemmas about that predicate.

3.8 Initial States, Invariants, and Error Patterns

The user typically writes three kinds of symbolic sets described in Table 3.

Table 3: *Main symbolic sets used in a* DM-Check *verification.*

Symbolic set	Purpose
Initial states	The states from which reachability starts. A single constrained pattern may denote infinitely many initial states.
Inductive invariant	A symbolic over-approximation of the reachable states that is closed under one-step transitions.
Error pattern	A symbolic description of forbidden states, used in negative safety proofs by intersection.

The same notation is used for all three. What changes is the role played by the set in the proof. This is why naming and comments in recorded command sequences matter. A disjunction that is clear when written in a paper can become difficult to manage when it appears several times in a command-line session.

For invariant generation, the specification should expose the structure that is expected to recur in the reachable frontier. Counters, process sets, tickets, queues, and protocol phases should be represented in a way that can naturally appear in constrained patterns.

If an important relation is hidden inside a complicated defined function, the generated frontier may be difficult to fold. If the same relation is summarized by a small auxiliary function with useful lemmas, the invariant-generation process is usually easier to guide.

3.9 Candidate Invariants

A candidate invariant should normally say more than the final safety property. For example, the property "there are not two critical processes" is usually not inductive by itself. To prove it, one often needs an invariant that also records how tickets, counters, queues, or tokens evolve.

In practice, a useful invariant has three qualities:

- it contains the initial states;
- it is stable under every transition rule;
- it exposes enough auxiliary structure to imply the desired safety property.

If the invariant fails to be closed under transitions, the first question should not be whether the property is false. The more common situation is that the candidate invariant is missing some bookkeeping relation that the system preserves.

3.10 Error Patterns

Error patterns are often much smaller than invariants. For mutual exclusion, a single pattern can describe all states with two critical processes. For a one-writer property, a pattern can describe all states with at least two active writers. For communication protocols, an error pattern may describe an output stream that violates the expected order.

The error pattern does not need to be reachable. It is intentionally an over-description of bad states. The verification goal is to prove that the verified invariant has empty intersection with it.

3.11 Auxiliary Functions in Constraints

Constraints may use functions defined in the Maude specification. In the Alternating Bit Protocol (see Section 9.2), auxiliary functions describe properties of queues and delivered lists. In Lamport's Bakery protocol (see Section 9.4), functions such as `tickets` and `i` summarize the ticket structure of process sets.

These functions make invariants readable, but they also create proof obligations. The user should expect to prove lemmas about membership, intervals, queues, multisets, prefixes, or cancellation properties.

Part II:
Verification Guide

4 Methodological Background

This section summarizes the formal ideas needed to interpret DM-Check output. It is not a substitute for the papers, but it explains the notions that appear in ordinary tool use.

Table 4: *Logical proof conditions and their principal DM-Check commands.*

Condition	Principal command	Meaning
$A \subseteq B$	`check A subsumed by B .`	Initial states are covered by the candidate invariant.
$R[B] \subseteq B$	`check ind-invariant B .`	Every symbolic one-step successor is folded back into the invariant.
$B \subseteq Q$	`check B subsumed by Q .`	The established invariant implies a positively described property.
$B \cap Error = \varnothing$	`intersect B with Error .`	No state described by the invariant is an error state.

4.1 Rewrite Theories as Transition Systems

A Maude system module can be viewed as a rewrite theory

$$\mathcal{R} = (\Sigma, E \cup B, R),$$

where Σ is the order-sorted signature, E is the set of equations, B is the set of structural axioms, and R is the set of rewrite rules. Equations define data and canonical forms. Rewrite rules define transitions between states.

DM-Check uses this transition-system view directly. A ground constructor term of sort `State` represents a concrete state. A rewrite step represents one system transition. A constrained pattern represents a symbolic set of such concrete states.

Examples are topmost: rules rewrite complete states rather than arbitrary subterms of states. This is the usual style for concurrent systems specified as configurations, counters, queues, or protocol states.

4.2 Constrained Narrowing

Constrained narrowing is the symbolic transition mechanism used by DM-Check. Given a constrained pattern and a rewrite rule, the tool computes symbolic successors that represent all concrete one-step successors obtained by applying that rule to instances of the pattern.

The key point is that a single narrowing step may describe infinitely many concrete transitions. This is what allows DM-Check to reason about parametric or infinite-state systems.

4.3 One-Step Closure

For a symbolic set B, transition closure means:

$$R[B] \subseteq B$$

This is exactly what `check ind-invariant` tries to prove. The command applies each relevant transition rule symbolically to each pattern in B, obtains child patterns, and then tries to show that each child is already covered by B.

There are several possible outcomes for a child:

- it is subsumed automatically by an existing disjunct;
- it is empty because its constraint is unsatisfiable;
- it requires a lemma about an auxiliary function;
- it must be split into cases;
- it reveals that the invariant is missing a disjunct or is too weak.

This is why failed closure is informative. It shows where the proposed invariant does not yet explain the transition behavior.

4.4 Subsumption

Subsumption is the main automatic folding criterion. A constrained pattern $u \mid \varphi$ is subsumed by another constrained pattern $v \mid \psi$ when the first denotes a subset of the second. Operationally, DM-Check looks for a matching substitution and then proves the corresponding implication between constraints.

Subsumption is a sufficient criterion for containment, but it is not complete. A pattern may fail to be subsumed by any single disjunct while still being contained in the union of several disjuncts.

4.5 Intersection

Intersection is the symbolic operation behind negative safety proofs. Given two disjunctions of constrained patterns, DM-Check tries to determine whether they have a common ground instance. If no such instance exists, the intersection is empty. The command:

```
intersect <invariant> with <error-patterns> .
```

is therefore a way of proving that no state described by the invariant is an error state. Since the invariant contains all reachable states, this proves that no reachable state is an error state.

The reply "`No intersection.`" should be read as a proof result, not as a failed search. It means that the symbolic description of the invariant and the symbolic description of the bad states are disjoint without session assumptions. At the start of an intersection, however, DM-Check takes the current session lemmas introduced by `add lemma`. NuITP uses them when checking whether the constraints of intersection terms are inconsistent, and every such lemma is listed as an obligation of the result. Consequently, the reply may instead be "`No intersection with the following proof obligations:`" even when no term has been closed explicitly with `empty`. Use `reset lemmas .` before `intersect` when those assumptions are not intended for the safety argument.

Variables from the two arguments are renamed independently before intersection. In particular, variables that occur only in constraints are existential variables local to their respective argument, even when the user wrote the same name on both sides. If terms remain, the output is still "`Intersection of constrained terms:`", and every term receives an identifier. The command `empty <id> .` may then close a term whose constraint is believed to be inconsistent; its constraint-to-`false` obligation is added to any lemma obligations already associated with the intersection. Disjointness is unconditional only after all displayed obligations have been proved.

4.6 Emptiness

A constrained pattern may denote the empty set because its constraint is unsatisfiable. Such a pattern can be discarded, but the unsatisfiability proof may require induction over the

equational theory. DM-Check automatically checks the constraints of children produced by symbolic operations. If it proves that a child's constraint implies `false`, the child is removed before the result is displayed. If the automatic check is inconclusive, the child is retained; it is not assumed to be satisfiable. A remaining emptiness argument may therefore require an explicit NuITP obligation. The same command may be used on an identified term in a non-empty intersection. In that setting, closing the term explicitly records its constraint-to-false obligation, so that a negative result cannot depend on an unreported emptiness assumption.

4.7 Where Proof Obligations Come From

DM-Check proof obligations arise when a symbolic operation depends on a semantic fact that is not purely syntactic. Typical examples are:

- a constraint implication needed for subsumption;
- unsatisfiability of a constraint;
- a lemma about an auxiliary function such as `tickets`, `allN`, or `gen`;
- a justification implication introduced by `substitution` or `variant unify`;
- a coverage implication introduced by `case`;
- a containment argument that is valid only modulo equations and structural axioms.

This explains why NuITP is part of the workflow. The model-checking step produces the right proof obligations; the inductive prover establishes the equational facts needed to close them. A lemma introduced by `add lemma` is a session assumption and is therefore also displayed among the obligations of every proof that uses it. Obligations generated by `empty`, `lemma`, `contained`, `substitution`, `variant unify`, or `case` belong only to the current proof. They are displayed in the verdict and by `show obligations`, but they are not reused as lemmas in later NuITP calls and are discarded when a new verification command begins.

The automatic NuITP strategy used by DM-Check is intentionally terminating and necessarily incomplete. Failure of the automatic strategy establishes neither satisfiability nor invalidity; it only leaves the corresponding semantic obligation open.

4.8 Semantic Transformations

The invariant-generation methodology uses transformations that preserve, or intentionally over-approximate, the set of states described by a constrained pattern [3]. They should be understood as proof steps over sets of states, not as textual edits to a term.

Table 5 summarizes the main transformation commands and their principal side conditions.

The equivalence-preserving transformations include constructor case splitting, variant unification, narrowing of selected auxiliary terms, variable abstraction, substitution, and Boolean splitting. Each of them has side conditions that make the step sound:

- *Constructor case splitting* replaces a pattern by a disjunction of cases for a selected variable. Every generator must be a constructor term of the selected sort or one of its subsorts. DM-Check computes constructor patterns not covered by the generator set and asks NuITP whether the node constraint excludes each pattern. Unproved exclusions become obligations; thus the split is applied without silently losing the omitted cases.
- *Variant unification* applies to a selected equation whose terms belong to an equational subtheory where the required variant-unification problem can be computed. Equations intended to participate in variant unification must be made available to Maude's variant machinery with the `[variant]` attribute. Equations intentionally left outside the variant

Table 5: *Main semantic transformation commands.*

Command	Effect	Main soundness condition
`case`	Splits a pattern according to constructor cases for a selected variable.	The generators must be constructor terms of the selected sort or its subsorts. DM-Check checks coverage and records any unproved exclusions as obligations.
`variant unify`, `vunify`	Uses variant unification to propagate a selected equation.	The node constraint must imply the equation. If the implication is not proved, the command is applied with an obligation. Defined non-variant subterms are abstracted automatically.
`narrowing`, `narrow`	Narrows a selected term, typically rooted by an auxiliary defined function.	The selected term must be suitable for the available narrowing step and expose enough constructor structure.
`variable abstraction`, `va`	Replaces a selected non-variable subterm by a fresh variable and records the defining equation.	The abstraction variable must be fresh and have the same sort as the selected subterm.
`substitution`, `sb`	Instantiates a selected variable by a well-sorted term.	The substitution must be non-cyclic, sort-compatible, and use only variables of the node. If its justification is not proved, it is applied with an obligation.
`split`	Performs a Boolean split on a selected Boolean term.	The module must declare `True` and `False` of sort `Bool`, following the NuITP convention.
`generalize`	Replaces a frontier pattern by a broader constrained pattern.	The proposed pattern must cover the original one. This is an over-approximation step, not an equivalence-preserving transformation.

subtheory are not used for this purpose. When this command is used as an equivalence-preserving transformation, the selected equation must follow from the constraint of the current node. DM-Check attempts to prove this implication and records it as an obligation if the proof is inconclusive. Variables in the equation must already occur in the node. Maximal subterms defined by non-variant equations are abstracted before unification so that their defining equations remain in the constraint.

- *Narrowing* is applied to a selected term, normally rooted by a defined function, whose arguments expose enough constructor structure for the narrowing step to be meaningful.
- *Variable abstraction* abstracts a selected non-variable subterm. The abstraction variable must be fresh and must have the same sort as the selected subterm; the defining equation is recorded in the constraint.
- *Substitution* instantiates a variable by a well-sorted term. The selected variable may occur in the pattern or only in its constraint. The replacement must not contain that variable, introduce variables absent from the node, or have an incompatible sort. DM-Check tries to prove the required constraint implication and otherwise applies the step with that implication as an obligation.
- *Boolean splitting* is applied to a term of sort `Bool` whose variables are available in the selected pattern or constraint. The active module must declare `True` and `False` of sort `Bool`; these are the NuITP constants, not the lowercase prelude Booleans.

Generalization is different. It deliberately replaces a selected frontier pattern by a broader constrained pattern. The proposed pattern must cover the original one, that is,

$$u \mid \varphi \subseteq v \mid \psi.$$

DM-Check requires a term matcher and uses NuITP, with the current session lemmas, to prove that each selected node constraint implies the generalized constraint. If coverage cannot be established, the command is rejected. Adding a lemma re-executes the preceding verification command, not the rejected generalization, so the user must issue `generalize` again. Generalization can make a finite fixpoint possible, but an overly broad generalization may also make the final safety property harder to prove.

4.9 Invariant Generation as a Fixpoint

Starting from the initial symbolic set A, DM-Check maintains an accumulated symbolic set P_d and a frontier F_d. Initially,

$$P_0 = F_0 = A.$$

At each level, the tool computes symbolic successors of F_d, removes those already covered by P_d, and obtains a new frontier F_{d+1}. The accumulated set is then updated as:

$$P_{d+1} = P_d \vee F_{d+1}.$$

If F_{d+1} is empty, the construction has reached a fixpoint and P_d is an inductive invariant. In infinite-state examples, reaching such a finite fixpoint may require user-guided generalization.

5 Verification Workflows

In this section, we describe the different approaches for invariant analysis. The standard workflow assumes that the user already has a candidate invariant B.

1. Write the set A of initial states and the candidate invariant B as disjunctions of constrained patterns.
2. Check the initial states already satisfy the candidate invariant, i.e., $A \subseteq B$ with "`check <A> subsumed by <B> .`"
3. Prove transition closure with "`check ind-invariant <B> .`"
4. Prove that the candidate invariant B is deadlock free with "`check <B> subsumed by <Enabled> .`", where *Enabled* is a disjunction containing one constrained left-hand side for each transition rule. An unconditional rule contributes `(LHS | true)`, whereas a supported conditional rule contributes `(LHS | condition)`. Omitting a conditional rule's condition admits states in which the rule is disabled and therefore does not establish absence of blocking.
5. Check the candidate invariant B does not incorrectly contain any bad state by using "`intersect <B> with <Bad> .`"

When the invariant is not known a priori, DM-Check automates, in Section 5.3, the discovery process from some initial state patterns.

5.1 Checking a Given Invariant

If the command "`check ind-invariant <B> .`" succeeds, transition closure is proved. If it does not, do not immediately rewrite the invariant. First inspect the open nodes. Open nodes are not necessarily counterexamples. They mean that the automatic procedure used by DM-Check did not close a proof branch. A failed child often points to the missing proof idea:

- if a child is not an instance of a previous state pattern but its constraint is unfeasible, close the child using the `empty` command (it generates a proof obligation for NuITP);
- if a child is not an instance of a previous state pattern but of the union of several patterns, close the child using the `contained` command (it generates a proof obligation for NuITP);
- if the child has the right shape to be an instance of a previous state but its constraint implication remains open, close it using the `lemma` command (it generates a proof obligation for NuITP);
- otherwise, use the transformation commands of Section 4.8.

This diagnostic step is important. It is common for an invariant to be mathematically correct while the current proof still lacks the auxiliary lemmas needed for the tool to close it automatically.

5.2 Proving Safety Properties

Apart of proving transitive closure of an invariant, two other proof styles can be useful.

The positive style proves that a disjunction of constrained patterns is contained in a target property expressed also as a disjunction of constrained patterns:

```
check <verified-invariant> subsumed by <property-patterns> .
```

The negative style proves that a disjunction of constrained patterns has no intersection with some bad states expressed also as a disjunction of constrained patterns:

```
intersect <verified-invariant> with <error-patterns> .
```

5.3 Generating an Invariant

When the invariant is not known in advance, use the following commands:

```
generate ind-invariant <initial-patterns> .
generate next level .
```

The shorter version `gen` of the `generate` keyword is a convenient alias for the corresponding generation commands. Invariant generation is best understood as a loop:

1. generate the next level;
2. inspect the remaining open children from the new frontier;
3. repeat until the frontier is empty.

The loop is finite only when the reachable symbolic set can be represented by the chosen patterns and generalizations. If the same family of states appears at every level with larger parameters, the next useful step is usually not another level, but a generalization.

5.4 When to Generalize

Generalization should be used deliberately. A good generalization explains a recurring pattern in the frontier and is strong enough to be closed under transitions. A poor generalization may be too weak and produce many new states that are not actually reachable, making the inductive proof harder.

Before generalizing, check whether the obstacle is instead a syntactic mismatch. Sometimes `vunify`, `va`, or `substitution` is enough to expose the equivalence between a frontier pattern and an existing disjunct. Generalization is appropriate when the frontier is not merely written differently, but is an instance of a broader symbolic region that should become part of the invariant.

5.5 Classifying the Status of a Result

A result should be described using one of the following statuses:

- *closed automatically*: no open nodes and no unproved obligations remain;
- *closed modulo obligations*: the symbolic tree is closed, but one or more recorded NuITP obligations are still pending;
- *open*: at least one proof-tree node has not been justified;
- *potential closure failure*: an uncovered symbolic successor remains after the available automatic and interactive steps;
- *genuine counterexample to closure*: a satisfiable successor outside the candidate invariant has been established;
- *inconclusive property query*: the set used in the final query has not first been established as an invariant containing the initial states.

Do not report a property as verified merely because the console contains no remaining symbolic children if proof obligations are still unproved.

5.6 Maintaining a Reproducible Proof

Because DM-Check sessions can include interactive decisions, a proof should be recorded as a transcript while it is being developed. A replayable `.dmc` script should also be saved with

`save proof` and checked with `load`. The transcript is not only a convenient way of replaying the same commands. It is part of the verification record: it documents which module was active, which symbolic set was checked, which lemmas were added, and how each open node was eventually handled.

A reproducible transcript should therefore include the complete sequence of DM-Check commands, starting with the module selection and ending with the final safety query. When the proof requires interaction, the transcript should also record the commands used to close open nodes, such as `empty`, `lemma`, and `contained`, together with any case split or semantic transformation applied to the generated patterns. If auxiliary lemmas are added during the session, the corresponding NuITP proof material should be stored with the same development.

This discipline is more than an administrative convention. Node identifiers, matching identifiers, and generated obligations depend on the exact version of the specification, the order of patterns, the active lemmas, and the tool release. A later change to the Maude module may invalidate a lemma, modify a generated frontier, or change the numbering of proof-tree nodes. A complete transcript gives a concrete point at which the new verification diverges from the previous one, making it possible to diagnose the change instead of reconstructing the proof from memory.

6 Working with NuITP

This section explains why an inductive theorem prover, such as NuITP, is necessary for discharging all the proof obligations generated by DM-Check.

6.1 Why NuITP Is Needed

DM-Check delegates inductive proof obligations to NuITP [9]. These obligations appear whenever a symbolic verification step depends on a semantic fact that cannot be justified by syntactic matching alone. For example, after a child pattern has been matched against an invariant disjunct, the tool may still need to prove that the constraint of the child implies the constraint of the matched pattern. Similarly, when a branch is declared empty, the reason is often that its constraint is unsatisfiable in the equational theory of the specification, which may itself require an inductive argument.

Obligations also arise from the auxiliary functions used in invariants and constraints. In examples with queues, lists, multisets, counters, or intervals, the invariant usually refers to functions that summarize the state. These functions make the symbolic invariant readable, but their algebraic properties must be available to the prover. Lemmas about functions such as `allN`, `l2ms`, `tickets`, or `gen` taken from the Bakery example (see Section 9.4) are therefore not external additions to the verification; they are part of the mathematical argument that justifies the symbolic proof.

Small obligations may be discharged automatically by NuITP. Larger or more structural obligations may require an interactive proof, usually by case analysis, induction, narrowing, or by introducing additional lemmas. In either case, the important point is that a DM-Check proof should not be considered fully closed until the obligations reported by the tool have also been proved in NuITP or explicitly recorded as pending.

6.2 How Obligations Should Be Read

A DM-Check obligation should be read as part of the verification argument. It is not an accidental artifact of the implementation. For example, if a child pattern is folded into an invariant disjunct only under a constraint implication, the implication is exactly the

mathematical reason why that folding is sound.

There are three common kinds of obligations:

- *constraint implications*, usually produced by subsumption;
- *inconsistency obligations*, produced when a pattern is claimed to be empty;
- *auxiliary-function lemmas*, produced when the invariant uses defined functions in constraints.

When developing a case study, keep these obligations in a separate proof file. If an obligation appears several times with small variations, it is often better to prove a more general lemma and add it to the session.

6.3 NuITP Proof Material

This manual records the DM-Check obligations that must be proved, but it does not reproduce the corresponding NuITP proof files. Each distributed example with obligations provides an `<example>.nuitp` script. From the Maude prompt, load the example specification, load `NuITP.maude`, start its interface, and then load the script:

```
Maude> load examples/<example>/<example>.maude
Maude> load NuITP.maude
Maude> erew in NuITP : init .
NuITP> load examples/<example>/<example>.nuitp
```

The alpha 12 scripts distributed for ABP, Bakery, QLOCK, conditional QLOCK, fair Readers–Writers, and Token Ring have been executed with the bundled NuITP release, and every goal in them reaches `qed`. In particular, the QLOCK script uses `0 ;; P:NzNat` as its generator set for `Nat`; this set covers the sort, and the script proves its second list-to-multiset lemma. The Token Ring script includes proofs of both obligations generated by DM-Check.

Each NuITP command in that file must occupy one physical line; the backslash continuation syntax of DM-Check scripts is not accepted by the NuITP loader. The NuITP manual remains the reference for proof commands and proof-engine behavior.

6.4 Recommended Discipline

When a DM-Check session produces obligations, it is usually better to prove reusable algebraic lemmas first, internalize them, and then return to the specific generated obligation. This avoids repeatedly proving the same facts about auxiliary functions.

6.5 Designing Useful Lemmas

Good lemmas are neither too specific nor too broad. A lemma that exactly repeats one generated obligation may solve the current branch but fail to help with the next similar branch. A lemma that is too general may be difficult to prove and may not simplify the generated obligations.

In practice, useful lemmas often describe how an auxiliary function behaves with respect to one constructor. For the Bakery example (see Section 9.4):

- how `allN` behaves on a non-empty set;
- how `tickets` behaves when a waiting process is added to a process set;
- how an interval function behaves at its lower or upper boundary;
- how a queue projection behaves under enqueue or dequeue operations.

These lemmas match the structure of the transition rules. Since proof obligations are generated from symbolic successors, they often expose exactly the constructor case that a useful lemma should cover.

6.6 Keeping DM-Check and NuITP Material Consistent

The DM-Check transcript and the external NuITP proof material should evolve together. When a new lemma is added to DM-Check, record the generated obligation and keep the corresponding NuITP proof file with the same example. In the current session-oriented workflow, `add lemma` re-executes the preceding command that required the lemma; therefore, the transcript should retain both the original failed attempt and the automatic re-execution. When a lemma is generalized or renamed, update the DM-Check transcript and the external NuITP material consistently.

The saved `.dmc` script and the `.nuitp` script have complementary roles: the first reproduces the symbolic proof steps, whereas the second proves the semantic obligations. Neither replaces the unedited transcript that records the actual results and identifiers of a particular run.

Part III:
Reference

7 Understanding DM-Check Output and Proof Trees

The console output is not ancillary diagnostic text. It is the concrete record of how a symbolic proof was decomposed, which branches were closed automatically, and which semantic arguments remain to be proved.

7.1 What a Proof Tree Represents

For an inductive-invariant check, a parent node normally identifies an invariant disjunct and a transition rule. Its children are constrained patterns denoting symbolic one-step successors. The tool then attempts to close each child by one of the following mechanisms:

- direct subsumption by an invariant disjunct;
- subsumption by another generated child when child-to-child folding is enabled;
- proof that the child is empty;
- containment in a union of patterns;
- a user-directed transformation followed by a new folding attempt;
- generation of a semantic obligation for NuITP.

The following excerpt, taken from the QLOCK invariant check (see Section 9.3), shows the typical shape of a proof tree with open proof obligations. Each parent corresponds to a disjunct of the candidate invariant. Each child is a symbolic successor generated by one rewrite rule. In this case, matching succeeds, but the associated constraint implication remains open and must be discharged by NuITP.

```
  Invariant could not be proved.

Parent 1
Term:
  < $2:MSet | $3:MSet | mt | $1:List >
Constraint:
  non-mt($2:MSet $3:MSet) = tt /\ set($2:MSet $3:MSet) = tt /\ l2ms($1:List) =
    $3:MSet

Parent 2
Term:
  < $6:MSet | $5:MSet | $7:Nat | $7:Nat ; $4:List >
Constraint:
  non-mt($6:MSet $5:MSet $7:Nat) = tt /\ set($6:MSet $5:MSet $7:Nat) = tt /\
    l2ms($7:Nat ; $4:List) = $5:MSet $7:Nat

Child 4 (Parent 1)
Last transformation: narrowing
Rule: r2w
Substitution:
  $1:List --> $15:List
  $2:MSet --> $12:Nat $13:MSet
  $3:MSet --> $14:MSet
Term:
  < $13:MSet | $12:Nat $14:MSet | mt | $15:List ; $12:Nat >
Matching parent #1: 1
Matching substitution #1:
  $1:List --> $15:List ; $12:Nat
  $2:MSet --> $13:MSet
```

```
  $3:MSet --> $12:Nat $14:MSet
NuITP implication #1 (open):
  tt = non-mt($12:Nat $13:MSet $14:MSet) /\ tt = set($12:Nat $13:MSet $14:MSet)
    /\ $14:MSet = l2ms($15:List) -> l2ms($15:List ; $12:Nat) = $12:Nat $14:MSet

Child 5 (Parent 2)
Last transformation: narrowing
Rule: c2r
Substitution:
  $4:List --> $16:List
  $5:MSet --> $18:MSet
  $6:MSet --> $17:MSet
  $7:Nat --> $19:Nat
Term:
  < $19:Nat $17:MSet | $18:MSet | mt | $16:List >
Matching parent #1: 1
Matching substitution #1:
  $1:List --> $16:List
  $2:MSet --> $19:Nat $17:MSet
  $3:MSet --> $18:MSet
NuITP implication #1 (open):
  tt = non-mt($19:Nat $17:MSet $18:MSet) /\ tt = set($19:Nat $17:MSet $18:MSet)
    /\ $19:Nat $18:MSet = $19:Nat l2ms($16:List) -> $18:MSet = l2ms($16:List)
```

The block for a child records both its immediate source and the last operation that produced it. A narrowing child is headed `Child N (Parent P)`; a node created by an auxiliary transformation is headed `Child N (from child P)`. The fields `Last transformation`, `Rule`, and `Substitution` are followed, when appropriate, by an operation-specific `Operation term` or `Operation data` block. The `Term`, matching information, and `Constraint` then describe the resulting child. In particular, `Substitution` is the current field name; older output called it `Unifier`.

7.2 Message Classes

DM-Check uses four visually distinct message classes. An Error is printed in red with the prefix `Error:`; the command was not applied, the proof state is unchanged, a running script is aborted, and `save proof` does not record the command. A Warning is printed in yellow with the prefix `Warning:`; the command was applied, but an obligation or another condition requires review. A Warning neither aborts a script nor prevents the command from being saved. Successful actions and informational summaries are printed in green, while headers that introduce structured material are cyan. Positive verdicts are green and negative or open outcomes are yellow.

The first line of a message is indented by two spaces. Material introduced by a header—lemmas, obligations, or a generated invariant—is indented by four spaces. Parent, child, term, and constraint blocks remain at the left margin. These conventions are semantically relevant when reading a transcript: the prefix, rather than the color alone, determines whether a script continues.

7.3 Node, Parent, and Match Identifiers

Commands such as `empty`, `lemma`, `contained`, `case`, and `vunify` refer to identifiers printed in the last result. These identifiers are session-local. They may change after modifying the specification, the order of invariant disjuncts, the active lemmas, or the tool release.

A reproducible transcript must therefore never present a command such as "`empty 5 .`"

without also presenting the output that establishes what node 5 is in that particular execution.

Identifiers are not reused during a proof session. Every auxiliary command, including substitution and generalization, assigns a fresh identifier to each resulting node. A new `check ind-invariant`, `generate ind-invariant`, `check ... subsumed by ...`, or `set module` restarts the identifier counter. By contrast, `generate next level` promotes frontier children without renumbering them and continues the generated-variable sequence.

Table 6: *Information that should be retained from a proof-tree node.*

Field	Why it matters
Node identifier	Used by subsequent interactive commands; unstable across executions.
Immediate source identifier	Distinguishes narrowing parents from children transformed by an auxiliary command.
Applied rule	Explains the symbolic transition that generated the child.
Child pattern	Describes the successor set that must be folded, split, or discarded.
Matcher identifier	Selects one of several possible subsumption matches.
Substitution	Shows how the child pattern aligns with a candidate parent pattern.
Constraint implication	States the semantic fact that must hold for the folding to be sound.
Status	Closed, empty, contained, transformed, or still open.

7.4 Conventions in Tool-Generated Output

The tool may rename user variables and introduce auxiliary variables when it prints proof trees. Names such as `$13:iNat` are generated variables local to the displayed result. Their numbers are not stable across runs. Names beginning with `#`, such as `#N:iNzNat`, usually come from generator patterns introduced by a user-provided case split.

Node identifiers may contain gaps. Identifiers are assigned before automatic constraint checking, and a node whose constraint is proved unsatisfiable is removed without renumbering the remaining nodes. Therefore, a missing identifier does not indicate that the output is incomplete. Nor is a closed identifier later reused for a different node in the same proof.

The field `Substitution` describes how the selected rewrite rule or symbolic step was applied to produce a child. A reply `Matching substitution` describes how a child is aligned with a candidate parent or invariant disjunct during subsumption. These are different substitutions and should not be conflated. After an auxiliary transformation, `Rule: none` appears together with the operation-specific details.

After every transformation or closure command, a green line summarizes the effect before the updated tree is printed. It identifies the source, the fresh node identifiers, and the number of automatically closed cases when relevant. The same convention applies to `empty`, `lemma`, and `contained`.

7.5 Paths from the Initial Pattern

The commands `show path i .` and `show path states i .` reconstruct the path from the initial pattern to a parent or child in the last inductive-invariant check or generation result. They are not available after subsumption or intersection. The full form includes the data of each step; the states form keeps only the nodes and the step descriptions. Both paths cross generalization and frontier promotion. For example, after the first Bakery generalization:

```
DM-Check> show path 5 .

  Path to child 5 (3 steps from the initial pattern):

Parent 1 (initial pattern)
  Term:
    $2:iNat | $2:iNat | $1:ProcIdleSet
  Constraint:
    true
  Renamed variables:
    IS:ProcIdleSet --> $1:ProcIdleSet
    N:iNat --> $2:iNat

  --> Step 1: narrowing with rule wake
      Substitution:
        $1:ProcIdleSet --> $3:ProcIdleSet ; [idle]
        $2:iNat --> $4:iNat

Parent 2 (child promoted by generate next level)
  Term:
    (1 + $4:iNat) | $4:iNat | $3:ProcIdleSet ; [wait($4:iNat)]
  Constraint:
    true

  --> Step 2: narrowing with rule wake
      Substitution:
        $3:ProcIdleSet --> $6:ProcIdleSet ; [idle]
        $4:iNat --> $5:iNat

Child 3
  Term:
    (1 + 1 + $5:iNat) | $5:iNat | $6:ProcIdleSet ; [wait($5:iNat)] ; [wait(1 +
    $5:iNat)]
  Constraint:
    true

  --> Step 3: generalization

Child 5
  Term:
    ($9:iNat + $10:iNat) | $10:iNat | $11:ProcWaitSet
  Constraint:
    tickets($11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat)
```

The compact form prints the same path without renamed variables or step data:

```
DM-Check> show path states 5 .

  Path to child 5 (3 steps from the initial pattern):

Parent 1 (initial pattern)
  Term:
    $2:iNat | $2:iNat | $1:ProcIdleSet
  Constraint:
    true

  --> Step 1: narrowing with rule wake

Parent 2 (child promoted by generate next level)
```

```
  Term:
    (1 + $4:iNat) | $4:iNat | $3:ProcIdleSet ; [wait($4:iNat)]
  Constraint:
    true

  --> Step 2: narrowing with rule wake

Child 3
  Term:
    (1 + 1 + $5:iNat) | $5:iNat | $6:ProcIdleSet ; [wait($5:iNat)] ; [wait(1 +
    $5:iNat)]
  Constraint:
    true

  --> Step 3: generalization

Child 5
  Term:
    ($9:iNat + $10:iNat) | $10:iNat | $11:ProcWaitSet
  Constraint:
    tickets($11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat)
```

7.6 Reading the Main Outcomes

7.6.1 Successful closure

A successful invariant check means that every symbolic successor has been justified. The result may be fully automatic, or it may be closed modulo proof obligations that still have to be proved in NuITP. The console message should therefore be read carefully.

The Readers–Writers example, fully described in Section 9.1, gives the simplest fully automatic case:

```
DM-Check> check ind-invariant \
>      < N:Natural , 0 > | true \
>   \/ < 0 , s(0) > | true .

  Invariant satisfied.
```

This output means that the supplied symbolic set is closed under all transitions made available to DM-Check, including the supported conditional rules and the unconditional rules marked with `[narrowing]`. No additional proof obligation is reported.

In contrast, the QLOCK example, fully described in Section 9.3, shows a successful closure modulo explicit proof obligations:

```
DM-Check> add lemma l2ms(L:List) = S:MSet -> \
>   l2ms(L:List ; X:Nat) = (S:MSet X:Nat) .

  Invariant satisfied with the following proof obligations:

    S:MSet = l2ms(L:List) -> l2ms(L:List ; X:Nat) = X:Nat S:MSet
    l2ms(X:Nat ; L:List) = X:Nat S:MSet -> S:MSet = l2ms(L:List)
```

This second result should not be reported merely as an unconditional automatic proof. The symbolic invariant check has been closed, but the two displayed obligations are part of the final verification argument and must be proved in NuITP.

7.6.2 Open implication

An open implication generally means that a structural matcher was found but the associated constraint implication was not discharged. The appropriate next step is usually `lemma`, `add lemma`, or a more general reusable theorem in NuITP; it is not necessarily a new invariant disjunct.

The following QLOCK fragment from Section 9.3 shows this situation. The child generated by rule `r2w` matches parent 1, but the corresponding implication about `l2ms` remains open.

```
Child 4 (Parent 1)
Last transformation: narrowing
Rule: r2w
Substitution:
  $1:List --> $15:List
  $2:MSet --> $12:Nat $13:MSet
  $3:MSet --> $14:MSet
Term:
  < $13:MSet | $12:Nat $14:MSet | mt | $15:List ; $12:Nat >
Matching parent #1: 1
Matching substitution #1:
  $1:List --> $15:List ; $12:Nat
  $2:MSet --> $13:MSet
  $3:MSet --> $12:Nat $14:MSet
NuITP implication #1 (open):
  tt = non-mt($12:Nat $13:MSet $14:MSet) /\ tt = set($12:Nat $13:MSet $14:MSet)
    /\ $14:MSet = l2ms($15:List) -> l2ms($15:List ; $12:Nat) = $12:Nat $14:MSet
```

The important point is that the syntactic part of the match has succeeded. What remains is a semantic property of the auxiliary function `l2ms`. In the QLOCK analysis of Section 9.3, this is later handled by adding suitable lemmas about the conversion from lists to multisets.

7.6.3 No matcher

No matcher means that direct subsumption by the considered disjunct failed. The child may nevertheless be empty, contained in a union, equivalent after a transformation, or genuinely outside the candidate invariant.

The QLOCK deadlock-freedom check provides a typical example. After the invariant is compared with the symbolic set of enabled states, one constrained term is not directly subsumed:

```
  Constrained terms on the left that could not be subsumed:

Term 4:
  < $17:MSet | $18:MSet | mt | $16:List >
Matching: no matching found
Constraint 4:
  non-mt($17:MSet $18:MSet) = tt /\ set($17:MSet $18:MSet) = tt /\ l2ms(
    $16:List) = $18:MSet
```

This output is not yet a counterexample to deadlock freedom. It only says that the current subsumption attempt did not find a direct matcher. In the QLOCK analysis, the remaining term is then transformed by variant unification, case splitting, narrowing, automatic constraint pruning, variable abstraction, and variant unification until the subsumption proof succeeds.

7.6.4 Intersection result

An empty intersection proves the requested negative property only when the first argument has already been established as an invariant containing all initial states. A non-empty intersection

is different: even if the first argument is an inductive invariant containing the initial states, it may still be a strict over-approximation of the reachable states. Therefore, a non-empty intersection means that the invariant is not strong enough for this negative proof; it is not by itself a reachable counterexample.

The Bakery example, fully described in Section 9.4, gives a concrete empty-intersection result. After the invariant has been generated and related to the compact ticket invariant, mutual exclusion is checked by intersecting it with the symbolic error pattern containing two critical processes.

```
DM-Check> intersect \
>       ($9:iNat + $10:iNat | $10:iNat | $11:ProcWaitSet) | \
>         tickets($11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat) \/ \
>       ($20:iNat + $21:iNat | $21:iNat | $19:ProcWaitSet ; [crit($21:iNat)]) | \
>         tickets($19:ProcWaitSet ; [wait($21:iNat)]) = \
>         i($21:iNat, $20:iNat + $21:iNat) \
> with \
>       (M:iNat | N:iNat | [crit(K:iNat)] ; [crit(L:iNat)] ; PS:ProcSet) | true .

  No intersection.
```

The reply "`No intersection.`" means that no state described by the verified symbolic invariant has two critical processes. Since the invariant-generation session has already established the invariant, modulo the displayed NuITP obligations, this gives the negative safety proof for mutual exclusion.

A non-empty result displays the intersection term, the independently renamed left and right terms, their substitutions, and the combined constraint. For example, in QLOCK:

```
DM-Check> intersect ((< U:MSet | W:MSet | mt | Q:List ; X:Nat >) | \
>   (l2ms(Q:List ; X:Nat) = mt)) with \
>   ((< U2:MSet | W2:MSet | mt | Q2:List >) | true) .

  Intersection of constrained terms:

Intersection Term 3:
  < %3:MSet | %4:MSet | mt | %1:List ; %2:Nat >
Left Term:
  < $2:MSet | $3:MSet | mt | $1:List ; $4:Nat >
Left Substitution:
  $1:List --> %1:List
  $2:MSet --> %3:MSet
  $3:MSet --> %4:MSet
  $4:Nat --> %2:Nat
Right Term:
  < $6:MSet | $7:MSet | mt | $5:List >
Right Substitution:
  $5:List --> %1:List ; %2:Nat
  $6:MSet --> %3:MSet
  $7:MSet --> %4:MSet
Constraint:
  l2ms(%1:List ; %2:Nat) = mt
```

No lemma or proof obligation is used to compute this result. If its displayed constraint is inconsistent, the identified term can be closed explicitly:

```
DM-Check> empty 3 .

  Term 3 closed as empty (obligation added: its constraint -> false).
```

```
  No intersection with the following proof obligations:

    l2ms(%1:List ; %2:Nat) = mt -> false

DM-Check> show obligations .

  Current proof obligations:

    l2ms(%1:List ; %2:Nat) = mt -> false
```

This is a conditional negative result until the displayed obligation has been proved. The term identifier also allows the user to close one term at a time when the intersection contains several constrained terms.

7.7 A Diagnostic Decision Sequence

For each open child, the first question is whether the child denotes any concrete state at all. If the constraint is inconsistent, the branch should be closed by an emptiness argument rather than by weakening the invariant. If the child is not empty, the next step is to inspect the available matchers and determine whether the failure is purely syntactic or whether a genuine semantic implication remains open.

When a matcher exists but the implication is open, the right response is usually to prove or add a lemma. When no matcher exists, the child may still be contained in the current symbolic description after a case split, a variant-unification step, a narrowing step on an auxiliary term, or a variable abstraction. Only after these possibilities have been ruled out should the user consider adding or generalizing an invariant region.

This ordering avoids weakening the invariant merely to compensate for a missing lemma or a syntactic mismatch. It also keeps the proof transcript closer to the mathematical structure of the argument: impossible branches are closed as empty, semantic implications are sent to NuITP, and genuinely new symbolic regions are added only when they are needed.

8 Command Reference

This chapter documents the session-oriented syntax used throughout the manual. The spellings below are the current public command forms used in the examples. Legacy spellings and migration rules for older transcripts are collected separately in Appendix B.

8.1 Command Summary

Command family	Accepted forms	Use and notes
Session exit	`quit` `q`	Leaves DM-Check and returns to the Maude prompt. These commands are entered without a final period.
Proof scripts	`load file` `save proof file .`	Replays or saves a `.dmc` command script. `load` has no final period; `save proof` has one. Both require `-allow-files` or `-trust`.

Help	`help .` `h .`	Prints a structured command reference. Read-only.
Module selection	`set module M .`	Selects the active verification module and resets the stored proof state.

Inspection	`show module .`
	`show lemmas .`
	`show last .`
	`show obligations .`
	`show path i .`
	`show path states i .`
		Displays the current module, session lemmas, last result, pending obligations, or a node path. Read-only.

Invariant checking	`check ind-invariant B .`	Checks whether `B` is closed under one-step symbolic successors. Starts a new proof using the current session lemmas.

Invariant generation	`generate ind-invariant B .`
	`gen ind-invariant B .`
	`generate next level .`
	`gen next level .`
		Initializes generation and computes subsequent symbolic frontiers. Updates the accumulated set and frontier.

Subsumption	`check A subsumed by B .`	Checks symbolic containment. Used for initial coverage and positive safety properties.
Intersection	`intersect A with B .`	Computes symbolic intersection using the current session lemmas. Used for negative safety properties; the lemmas are reported as obligations.

Proof management	`add lemma L .`
	`reset lemmas .`
	`empty i .`
	`lemma i .`
	`lemma i match M .`
	`contained i .`
		Adds or clears session lemmas, closes open children or intersection terms, records obligations, or marks containment in the current proof tree.

Deprecated rerun	`recheck .`	Deprecated. Re-execution is now performed automatically when required.
Case splitting	`case to i on X with G .`	Splits node `i`; checks generator sorts, constructor form, and coverage.

Variable abstraction	`variable abstraction X with V in i .`
	`va X with V in i .`
	`select variable abstraction n .`
	`select va n .`
		Abstracts a selected term `X` by a variable `V`; selection commands choose one displayed abstraction.

Substitution	`substitution to i on X with t .` `sb to i on X with t .`	Instantiates `X`; rejects cyclic, fresh-variable, and sort errors and may add a justification obligation.
Variant unification	`variant unify t1 =?  t2 in i .` `vunify t1 =?  t2 in i .`	Propagates a justified equation; may add an obligation and preserves the node if there is no unifier.

Narrowing	`narrowing i on t .` `narrow i on t .`	Narrows auxiliary term `t` in node `i`.
Generalization	`generalize i ... to P .`	Generalizes one or more frontier nodes after checking coverage by the single pattern `P`.
Boolean splitting	`split i with t .`	Splits node `i` according to the Boolean term `t`.

8.2 Session and Information Commands

8.2.1 `set module`

Syntax

```
set module <module-name> .
```

Selects the module whose equations, axioms, rules, sorts, and operators will be used in later commands. Selecting a module resets the current session state.

A successful module selection prints a confirmation message. For example:

```
DM-Check> set module BAKERY-AUX .

  Module BAKERY-AUX is now active.
```

No proof obligation is generated by this command. It only selects the active module and resets the session state.

Before activation, the flattened module is checked against the supported specification class. For example, the following diagnostic reports, in one module, four unsupported forms:

```
DM-Check> set module UNSAFE .

  Error: module UNSAFE was not activated.
  DM-Check cannot reason soundly about these statements:
    - equation f(N:Nat): owise equation
    - equation p(N:Nat): uses the built-in _==_ or _=/=_
    - membership g(z): membership axiom
    - rule bad: condition other than t = t'
  Maude evaluates the built-in _==_ and _=/=_ and owise equations
  syntactically on terms with variables (N == 0 reduces to false), so
  narrowing and NuITP can lose states. Use predicates defined with
  variant equations instead (e.g. eq N .=. N = True [variant] .).
  Conditional rules may only have equational conditions (t = t').
  Unification ignores membership axioms (mb, cmb), so narrowing can also
  lose states: declare those sorts with operator declarations (mb f(X:S) : S'
  is op f : S -> S') or use a predicate in the constraints.
```

The active module, if any, is preserved after this Error, and another `set module` command may be issued immediately. Successful activation may add a Warning that lists sorts for which no constructor operator was found; this Warning does not cancel the activation.

8.2.2 help and h

Syntax

```
help .
h .
```

The commands `help .` and `h .` print a structured command reference at the DM-Check prompt. Commands are organized into session, inspection, verification, pattern-transformation and generalization, and proof-management categories. The help output also indicates which arguments must be replaced by user-provided values and which commands require a terminating period.

The full help output is not reproduced in this manual, because the command reference below gives the same information in a more structured form and can add the methodological comments needed to interpret each command. Both help commands are read-only: they do not change the active module, the stored lemmas, the last proof tree, or the current proof obligations.

8.2.3 load and save proof

Syntax

```
load <file>
save proof <file> .
```

The `load` command has no final period, whereas `save proof` is terminated by a period. Both require Maude to have been started with `-allow-files` or `-trust`. The following Readers–Writers session saves the accepted proof commands and then replays them:

```
DM-Check> save proof /tmp/dmcheck-manual-rw.dmc .

  Proof script saved to /tmp/dmcheck-manual-rw.dmc.

DM-Check> load /tmp/dmcheck-manual-rw.dmc

  Begin running script /tmp/dmcheck-manual-rw.dmc.

DM-Check> set module R&W .

  Module R&W is now active.

DM-Check> check ind-invariant < N:Natural , 0 > | true  \/ < 0 , s(0) > | true
    .

  Invariant satisfied.

  End running script /tmp/dmcheck-manual-rw.dmc.
```

Commands read from a script are echoed before their output. The loader ignores the comment and blank-line forms described in Section 2.3 and joins lines ending in a backslash. It rejects a nested `load`. The first Error aborts the script with `Error: script ... aborted: the last command could not be parsed or executed.`; a Warning does not abort it.

The saved file contains only accepted commands that change the proof state, with one blank line between commands and a descriptive `--` header. Repeated spaces are normalized,

and long disjunctions and subsumption commands are split with backslash continuations. If no such command has been recorded, the save is rejected and no file is written. A pending variable-abstraction choice is saved only together with the `select va` command that completes it.

8.2.4 `show module`, `show last`, `show lemmas`, and `show obligations`

```
show module .
show last .
show lemmas .
show obligations .
```

The inspection commands do not modify the proof state. They are useful for checking the active module, replaying the last result, and recording the lemmas and obligations that belong to the current verification. For example:

```
DM-Check> show module .

  Module R&W is active.

DM-Check> show last .

  Error: no previous result available.
```

After a successful invariant check, `show last .` repeats the last result:

```
DM-Check> check ind-invariant \
>      < N:Natural , 0 > | true \
>   \/ < 0 , s(0) > | true .

  Invariant satisfied.

DM-Check> show last .

  Invariant satisfied.
```

When no lemma has been added, `show lemmas .` reports an empty list:

```
DM-Check> show lemmas .

  Current lemmas:

    No lemmas saved.
```

After the QLOCK lemmas have been added, the same command records them explicitly:

```
DM-Check> show lemmas .

  Current lemmas:

    S:MSet = l2ms(L:List) -> l2ms(L:List ; X:Nat) = X:Nat S:MSet
    l2ms(X:Nat ; L:List) = X:Nat S:MSet -> S:MSet = l2ms(L:List)
```

Similarly, `show obligations .` distinguishes an empty obligation set from a session that is closed only modulo pending proof obligations:

```
DM-Check> show obligations .

  Current proof obligations:
```

```
  No proof obligations pending.
```

```
DM-Check> show obligations .

  Current proof obligations:

    null = e($28:iNat), tickets($29:ProcWaitSet) -> false
    e($33:iNat), tickets($34:ProcWaitSet) = e($33:iNat + $32:iNzNat), i(
    $33:iNat, $33:iNat + $32:iNzNat) -> tickets($34:ProcWaitSet) = i(1 +
    $33:iNat, 1 + $33:iNat + $32:iNzNat)
    e($30:iNat) = e($30:iNat), tickets($31:ProcWaitSet) -> null = tickets(
    $31:ProcWaitSet)
    tickets(($30:ProcWaitSet ; [wait($29:iNat)]) ; [wait($29:iNat)]) = i(
    $29:iNat, $28:iNat + $29:iNat) -> false
```

A proof that depends on obligations printed by this command must be reported as conditional until the corresponding NuITP proof material has been checked.

`show lemmas` and `show obligations` deliberately report different sets. The first lists the current session lemmas and, after `reset lemmas`, any removed lemmas still used by the current proof. The second lists every assumption of the current proof: its lemmas, generated obligations, and open nodes. Before any verification command it reports `No proof obligations pending.` Immediately after `intersect`, it does the same only if the intersection used no session lemmas; otherwise it lists those lemmas. If an intersection term is then closed with `empty`, the resulting constraint-to-false implication is added to the same list. Clearing the session with `reset lemmas` does not remove lemma assumptions from the current intersection result.

8.2.5 `show path` and `show path states`

Syntax

```
show path <id> .
show path states <id> .
```

Both commands are read-only and are available for nodes in the last `check ind-invariant` or `generate ind-invariant` result. The full output and the compact states-only form are illustrated in Section 7. A request without an active module, without a preceding result, after subsumption or intersection, or for an identifier absent from the last result is rejected without changing the proof state.

8.2.6 `reset lemmas`

Syntax

```
reset lemmas .
```

This command removes all lemmas introduced by `add lemma` without re-executing a verification command or changing the current result. A proof that has already used those lemmas retains them until a new proof starts:

```
DM-Check> reset lemmas .

  Session lemmas cleared.

DM-Check> show lemmas .
```

```
Current lemmas:

  No lemmas saved.

Lemmas still used by the current proof (removed with reset lemmas):

  true -> N:Natural = N:Natural
```

The next verification command starts without the removed lemmas. The command is state-changing and is therefore recorded by `save proof`.

8.2.7 `quit` and `q`

```
quit
q
```

Leaves the DM-Check interface and returns to the Maude prompt. These two commands are entered without a terminating period. The tool prints `Bye.` before control returns to Maude:

```
DM-Check> quit

  Bye.
```

```
DM-Check> q

  Bye.
```

8.3 Verification Commands

8.3.1 `check ind-invariant`

Syntax

```
check ind-invariant <disjunction> .
```

Checks whether the denotation of the supplied disjunction is closed under every transition rule available to constrained narrowing. The command computes symbolic children and attempts to fold them into the candidate invariant or other admissible covering patterns.

Preconditions. An active module must be selected. Relevant unconditional rules must carry the `[narrowing]` attribute; supported conditional rules are transformed internally.

Successful result. Every child has been closed, possibly modulo recorded proof obligations.

Typical open result. An uncovered child, an open constraint implication, an unproved emptiness claim, or a pattern requiring transformation. Failure of the automatic NuITP strategy does not establish that the implication is false; it only leaves the semantic obligation open.

8.3.2 check ... subsumed by ...

Syntax

```
check <disjunction-1> \
subsumed by \
<disjunction-2> .
```

Checks whether the first symbolic set is contained in the second using the tool's subsumption procedure and generated semantic obligations. The command is used for initial-state coverage and for positive safety proofs.

Subsumption by one disjunct is sufficient but not complete for containment in a union. A failed direct match should therefore be interpreted using the decision sequence in Chapter 7. If a matcher is found but its implication is not discharged automatically, the implication remains an open NuITP obligation; it is not a proof that containment fails.

8.3.3 intersect ... with ...

Syntax

```
intersect <disjunction-1> \
with \
<disjunction-2> .
```

Computes a symbolic intersection. If it is empty and the first argument is an established invariant containing the initial states, the command proves that no reachable state belongs to the second set.

The computation starts with the current session lemmas introduced by `add lemma`. NuITP uses them to discard intersection terms whose constraints are inconsistent, and DM-Check reports the lemmas as proof obligations. Therefore an intersection may report `No intersection with the following proof obligations:` without any preceding `empty` command. If terms remain, the output is still `Intersection of constrained terms:`. Adding a lemma after `intersect` repeats that intersection with the new lemma, as it does for the other proof commands. Use `reset lemmas .` before the intersection when session lemmas are not intended to be assumptions of the negative safety proof. Resetting them afterwards clears the session but leaves them attached to the current proof result.

Variables from the two arguments are renamed independently before the intersection is computed; in particular, variables occurring only in a constraint are existential and do not become shared merely because their names coincide. If Maude reports that the unifier set may be incomplete, the command is rejected and the preceding session state is retained.

Each term in a non-empty result has an identifier. The user may apply `empty <id> .` to an intersection term; this removes that term and adds its constraint-to-false implication to the lemma obligations already associated with the result. When every term has been removed in this way, the output is `No intersection with the following proof obligations:`, followed by all obligations that must still be discharged. The command `show obligations .` lists both kinds; without session lemmas or explicit closures it reports `No proof obligations pending.` The commands `lemma` and `contained` do not apply to intersection terms.

The Bakery mutual-exclusion query (see Section 9.4) illustrates the empty-intersection:

```
DM-Check> intersect \
>       ($9:iNat + $10:iNat | $10:iNat | $11:ProcWaitSet) | \
>         tickets($11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat) \/ \
```

```
>       ($20:iNat + $21:iNat | $21:iNat | $19:ProcWaitSet ; [crit($21:iNat)]) | \
>         tickets($19:ProcWaitSet ; [wait($21:iNat)]) = \
>         i($21:iNat, $20:iNat + $21:iNat) \
> with \
>       (M:iNat | N:iNat | [crit(K:iNat)] ; [crit(L:iNat)] ; PS:ProcSet) | true .

  No intersection.
```

This is a successful negative result. A non-empty intersection should be treated with more care. It identifies a symbolic overlap between the invariant and the error pattern, but an inductive invariant may over-approximate the reachable states. Therefore, the overlap only shows that the current invariant does not suffice for the negative proof. A real counterexample additionally requires a reachability argument for some state in the intersection.

8.4 Invariant-Generation Commands

8.4.1 `generate ind-invariant` and `gen ind-invariant`

Syntax

```
generate ind-invariant <initial-disjunction> .
gen ind-invariant <initial-disjunction> .
```

Both forms initialize the accumulated symbolic set and frontier from the supplied initial states. The long form is more explicit; the `gen` form is the short alias used in compact command sequences. The input should represent the actual initial set unless the verification argument explicitly starts from a larger over-approximation.

8.4.2 `generate next level` and `gen next level`

Syntax

```
generate next level .
gen next level .
```

Both forms compute symbolic successors of the current frontier, remove states already covered by the accumulated set, and store the remaining frontier. If no frontier remains, a symbolic fixpoint has been reached. Node identifiers and the fresh-variable counter continue from the preceding level, so paths and variable provenance remain meaningful. The command can continue only a result produced by `generate ind-invariant`; in any other proof state it reports an Error. If Maude reports that the narrowing result may be incomplete, the command is rejected and the previous frontier is retained.

The following Bakery fragment shows a generated level before any generalization has been applied. The new frontier consists of two uncovered children: a second waiting process and one process entering the critical section.

```
DM-Check> generate next level .

  Invariant could not be proved.

Parent 1
Term:
  $2:iNat | $2:iNat | $1:ProcIdleSet
Constraint:
```

```
  true

Parent 2
Term:
  (1 + $4:iNat) | $4:iNat | $3:ProcIdleSet ; [wait($4:iNat)]
Constraint:
  true

Child 3 (Parent 2)
Last transformation: narrowing
Rule: wake
Substitution:
  $3:ProcIdleSet --> $6:ProcIdleSet ; [idle]
  $4:iNat --> $5:iNat
Term:
  (1 + 1 + $5:iNat) | $5:iNat | $6:ProcIdleSet ; [wait($5:iNat)] ; [wait(1 +
    $5:iNat)]
Matching: no matching found
Constraint:
  true

Child 4 (Parent 2)
Last transformation: narrowing
Rule: crit
Substitution:
  $3:ProcIdleSet --> $8:ProcIdleSet
  $4:iNat --> $7:iNat
Term:
  (1 + $7:iNat) | $7:iNat | $8:ProcIdleSet ; [crit($7:iNat)]
Matching: no matching found
Constraint:
  true
```

In this situation, computing more levels would continue to expose the same family of waiting-process patterns. The useful step is to generalize Child 3 into a symbolic region described by the `tickets` invariant.

8.5 Lemmas and Proof-Tree Closure Commands

8.5.1 `add lemma`

Syntax

```
add lemma <lemma> .
```

Adds a reusable semantic lemma to the active session. In the current workflow, the command automatically re-executes the preceding command that required the lemma. There is therefore no need to invoke the deprecated `recheck` command. Before any proof command, `add lemma` only stores the lemma and reports that it was added. Session lemmas are assumptions: every lemma used by a successful proof is consequently listed among that proof's obligations.

The following ABP fragment shows the intended behavior. After the lemma is added, DM-Check re-executes the invariant check that was previously open and reports the remaining NuITP obligation.

```
DM-Check> add lemma (allN(M:iNat ; NS:iSet, N:iNat) = True) -> \
>               (M:iNat = N:iNat) /\ (allN(NS:iSet, N:iNat) = True) .
```

```
Invariant satisfied with the following proof obligations:

  True = allN(M:iNat ; NS:iSet, N:iNat) -> True = allN(NS:iSet, N:iNat) /\
  N:iNat = M:iNat
```

8.5.2 empty

Syntax

```
empty <id> .
```

Closes the selected child or intersection term by asserting that its constraint is unsatisfiable. If unsatisfiability is not decided automatically, retain and prove the generated NuITP obligation. An unsuccessful automatic emptiness check leaves the branch open; it does not prove that the constraint is satisfiable. The generated obligation belongs to the current proof and is not reused as a lemma by a later proof command.

8.5.3 lemma

Syntax

```
lemma <id> .
lemma <id> match <match-id> .
```

Converts the selected open constraint implication into a proof obligation. The match-qualified form selects the implication associated with one particular matcher of the same open node. As with `empty`, the obligation closes the current tree but is not added to the reusable session lemmas.

8.5.4 contained

Syntax

```
contained <id> .
```

Records that a child is contained in a union of patterns even though no single disjunct closed it by direct subsumption. The transcript and generated obligation must make the union-containment argument auditable. The pending check remains an obligation of the current proof rather than a reusable lemma.

8.6 Pattern-Transformation Commands

Transformations should be applied only after inspecting the selected node and understanding why automatic folding failed.

8.6.1 Constructor case splitting

Syntax

```
case to <id> on <variable> with <generators> .
```

Splits a pattern according to a constructor-based generator set for the sort of the selected variable. Every generator must have that sort or a subsort and must be a constructor term. DM-Check computes the constructor patterns not covered by the supplied generators and asks NuITP whether the current constraint excludes them. Any case not discharged becomes a proof obligation, and the applied split is accompanied by a warning. Thus omitted cases are never silently discarded. A generator variable that is absent from the current node is fresh even if the same name occurs only in the recorded history of that node; accepted fresh variables are renamed to the tool's generated variable names.

For example, the following deliberately incomplete split is applied, but its uncovered zero case is made explicit:

```
DM-Check> case to 3 on $3:Natural with s(#N:Natural) .

  Case split on child 3: every resulting case is closed.

  Warning: the generator set may not cover every value of
  $3:Natural in node 3. Not covered:
    0.Natural
  The case split was applied and this implication was added
  to the proof obligations for each pattern not covered:

    $3:Natural = 0 -> false

  Invariant satisfied with the following proof obligations:

    $3:Natural = 0 -> false
```

8.6.2 Variable abstraction

Syntax

```
variable abstraction <term> with <variable> in <id> .
va <term> with <variable> in <id> .
select variable abstraction <number> .
select va <number> .
```

Replaces the selected non-variable subterm by the given variable and records the defining relationship in the constraint. The variable must be fresh in the selected node and must have the same sort as the abstracted subterm. The `va` form is the short alias. When the tool displays several possible abstractions, use one of the `select` commands to choose the desired alternative. Candidates are printed under `Possible variable abstractions:` as numbered blocks of the form `N.`, followed by `Term:` and `Constraint:`. The list ends with the line `Choose one with select va <n> .  (any other command cancels the choice)`. While this choice is pending, `select`, `show`, and `help` preserve it; any other command cancels the choice before that command is executed. A `select va` command issued when no choice is pending reports an Error. A variable of a proper subsort is rejected because it would lose instances of the selected term outside that subsort. Freshness is checked against the current

node, not against variables that occur only in its recorded history.

8.6.3 Explicit substitution

Syntax

```
substitution to <id> on <variable> with <term> .
sb to <id> on <variable> with <term> .
```

Instantiates a variable occurring either in the term or only in its constraint. The replacement must be sort-compatible, must not contain the selected variable, and may use only variables already present in the node. NuITP checks that the node constraint implies the equality represented by the substitution. If it cannot prove that implication, the substitution is still applied with a warning and the implication becomes a proof obligation. The `sb` form is the short alias.

8.6.4 Variant unification

Syntax

```
variant unify <term-1> =? <term-2> in <id> .
vunify <term-1> =? <term-2> in <id> .
```

Propagates variant-unification substitutions and constraints to the selected pattern. The selected equation should belong to a part of the equational theory where the required variant-unification problem can be computed. Equations meant to participate in this step must be available to Maude's variant machinery, using the `[variant]` attribute where required. Equations intentionally left outside the variant subtheory are not used by this command. The `vunify` form is the short alias. When the command is used as an equivalence-preserving transformation, NuITP checks that the selected equation follows from the node constraint. If it cannot prove the implication, the step is applied with a warning and the implication becomes an obligation. Variables in the equation must already occur in the node. Defined subterms outside the variant subtheory are first abstracted, with their defining equations retained in the constraint. If there is no unifier, the command is rejected and the original node is kept; the diagnostic suggests closing the node with `empty` when the equation follows from its constraint. An incomplete unifier set also causes rejection.

An equation containing a variable that does not occur in the node is rejected. The required justification would state that every state represented by the node satisfies the equation for *some* value of that new variable; it therefore contains an existential quantifier that DM-Check can neither check nor record as an ordinary proof obligation. The diagnostic gives the sound alternative when one side t_1 contains only variables of the node and the other side t_2 is a constructor term:

1. abstract t_1 with `va t1 with X in <id> .`, unless t_1 is already a variable;
2. cover the possible values of the new node variable with `case to <id2> on X with t2 .`; and
3. apply `vunify t1 =? t2 in <id3> .` after the variables of t_2 have become variables of the node.

For example, the following command was rejected in Readers–Writers after node 3 had been generated from `< s(R:Natural), 0 > | true`:

```
DM-Check> variant unify $3:Natural =? Z:Natural in 3 .

  Error: the equation has variables that do not occur in the node.
  Its justification (every state of the node satisfies the equation for SOME
  value of those variables) can be neither checked nor recorded as a proof
  obligation. The command was not applied. If one side, t1, has only variables
  of the node and the other, t2, is a constructor term, do the step in three
  commands:
    1. va t1 with X in <id> .       (skip it if t1 is already a variable)
    2. case to <id2> on X with t2 . (checks that t2 covers every value of X)
    3. vunify t1 =? t2 in <id3> .   (now t2 only has variables of the node)
```

8.6.5 Narrowing an auxiliary term

Syntax

```
narrowing <id> on <term> .
narrow <id> on <term> .
```

Applies narrowing to the selected auxiliary term in the open node identified by `<id>`. The selected term should normally contain a defined symbol whose arguments expose constructor structure. Every child produced by the narrowing step is subsequently checked for constraint unsatisfiability. Consequently, some narrowing results may be removed automatically and may not appear in the displayed proof tree. Constants defined by equations may be selected. Equational conditions of conditional rules are supported below the root of the selected term; matching, rewrite, and membership conditions in equations do not generate such cases. If the selected operator has no applicable narrowing step, or if the result may be incomplete, the command is rejected and the node is retained. The `narrow` form is the short alias.

8.6.6 Generalization

Syntax

```
generalize <id> [<id> ...] to <constrained-pattern> .
```

Generalization deliberately enlarges the symbolic description in order to capture one or more recurring frontier families. Exactly one constrained pattern is accepted. For every selected node, its term must match the pattern and NuITP, using the session lemmas, must prove that the node constraint implies the pattern constraint. Otherwise the whole command is rejected and names the uncovered nodes. A successful generalization creates a fresh child identifier. Generalization must be followed by new closure and property checks; adding a lemma reruns the last proof command, not a rejected `generalize` command.

8.6.7 Boolean splitting

Syntax

```
split <id> with <Boolean-term> .
```

Performs Boolean splitting on the selected open node, producing the cases in which the given expression is true and false. The selected expression must have sort `Bool`, and all its

variables must occur in the selected pattern or its constraint. The active module must declare `True` and `False` of sort `Bool`, following the NuITP convention; the prelude values `true`/`false` are not substitutes for these constants.

8.7 Deprecated Command: `recheck`

Syntax

```
recheck .
```

The command `recheck` is deprecated because commands that need to be rerun are now re-executed automatically, notably after `add lemma`. It is retained only for compatibility with old transcripts and command examples.

```
DM-Check> recheck .

  Warning: command deprecated. Recheck is now performed automatically when
    required.
```

9 Examples and Case Studies

This section presents representative DM-Check examples using the modules, command transcripts, and example material supplied with the tool distribution. The goal is not to reproduce every line of each specification, but to show the verification structure that the user will find in the corresponding Maude and DM-Check files.

The examples cover two complementary uses of DM-Check. Readers–Writers, ABP, and QLOCK illustrate invariant checking from a user-provided symbolic invariant. Bakery illustrates invariant generation, where the invariant is built incrementally from the initial symbolic states and then generalized.

9.1 Readers–Writers

The basic Readers–Writers example is specified in the module `R&W`. A state has the form

$$\langle r, w \rangle$$

where r is the number of active readers and w is the number of active writers. Natural numbers are represented by `0` and `s`. The transition rules model writer entry, writer exit, reader entry, and reader exit.

```
mod R&W is
  sort Natural .
  op 0 : -> Natural [ctor rpo 0] .
  op s : Natural -> Natural [ctor rpo 1] .
  sort Config .
  op <_,_> : Natural Natural -> Config [ctor rpo 2] .

  vars R W : Natural .

  rl [enter-w] : < 0, 0 > => < 0, s(0) > [narrowing] .
  rl [leave-w] : < R, s(W) > => < R, W > [narrowing] .
  rl [enter-r] : < R, 0 > => < s(R), 0 > [narrowing] .
  rl [leave-r] : < s(R), W > => < R, W > [narrowing] .
```

```
endm
```

Listing 8: *Readers–Writers module.*

The DM-Check session first selects the module and then checks the inductive invariant

$$\langle N, 0\rangle \mid true \quad \vee \quad \langle 0, s(0)\rangle \mid true.$$

This symbolic set states that either there are no active writers, or there is exactly one active writer and no active reader.

```
DM-Check> set module R&W .

  Module R&W is now active.

DM-Check> check ind-invariant \
>      < N:Natural , 0 > | true \
>   \/ < 0 , s(0) > | true .

  Invariant satisfied.
```

Listing 9: *Readers–Writers invariant check.*

No NuITP proof is needed in this example. The invariant is closed automatically under the four transition rules.

The same session then checks a symbolic containment property. The right-hand side is the symbolic set of states matching the left-hand sides of the rules; therefore, this query can be read as a deadlock-freedom check for the states covered by the invariant.

```
DM-Check> check \
>      < N:Natural , 0 > | true \
>   \/ < 0 , s(0) > | true \
>  subsumed by \
>      < 0, 0 > | true \
>   \/ < R:Natural, s(W:Natural) > | true \
>   \/ < R:Natural, 0 > | true \
>   \/ < s(R:Natural), W:Natural > | true .

  Subsumption satisfied.
```

Listing 10: *Readers–Writers containment check.*

The example illustrates the simplest fully automatic pattern: select the module, check transition closure, and then use subsumption to prove a property from the verified symbolic set.

9.2 Alternating Bit Protocol

The Alternating Bit Protocol example verifies a communication protocol over unreliable channels. The module `ABP` contains the transition rules, whereas `ABP-PRED` defines the auxiliary predicates used in the invariant. The session selects the combined module:

```
set module ABP-PRED-AUX .
```

States have the form

$$N : B > BPQ \;||\; BQ < B' : NL,$$

where N is the data value currently being sent, B is the sender bit, BPQ is the packet channel, BQ is the acknowledgement channel, B' is the receiver bit, and NL is the output list. The state constructor is declared as follows:

```
op _:_>_||_<_:_ : iNat Bit BPQue BQue Bit iList -> Sys [ctor rpo 18] .
```

The invariant used by the session has six disjuncts. It relates the sender bit, receiver bit, channel contents, and output stream by means of auxiliary functions such as `aP`, `allN`, `getNs`, and `gen`.

```
check ind-invariant \
      ((0 : on > Q:OnBPQue || P:OnBQue < on : nil) | (aP(Q:OnBPQue,0) = True)) \
   \/ ((1 + N:iNat : on > Q':OffBPQue Q:OnBPQue || P:OnBQue < on : L:iList) | \
         ((1 + N:iNat :: L:iList) = gen(1 + N:iNat)) /\ \
               (aP(Q:OnBPQue,1 + N:iNat) = True) /\ (aP(Q':OffBPQue,N:iNat) = True
    )) \
   \/ ((0 : off > Q':OffBPQue || P':OffBQue < off : nil) | (aP(Q':OffBPQue,0) =
    True)) \
   \/ ((1 + N:iNat : off > Q:OnBPQue Q':OffBPQue || P':OffBQue < off : L:iList) |
    \
         ((1 + N:iNat :: L:iList) = gen(1 + N:iNat)) /\ \
               (aP(Q':OffBPQue,1 + N:iNat) = True) /\ (aP(Q:OnBPQue,N:iNat) = True
    )) \
   \/ ((N:iNat : on > Q:OnBPQue || P:OnBQue P':OffBQue < off : L:iList) | \
         (L:iList = gen(N:iNat)) /\ (aP(Q:OnBPQue,N:iNat) = True)) \
   \/ ((N:iNat : off > Q':OffBPQue || P':OffBQue P:OnBQue < on : L:iList) | \
         (L:iList = gen(N:iNat)) /\ (aP(Q':OffBPQue,N:iNat) = True)) .
```

Listing 11: *ABP invariant check.*

The first attempt does not close automatically. The generated proof tree contains many open implications, all with the same underlying reason: after removing a packet from a queue, DM-Check needs a semantic fact saying that `allN` continues to hold for the remaining multiset of packet numbers. A representative open implication printed by the tool is:

```
NuITP implication #1 (open):
  True = allN($39:iNat ; getNs($40:OnBPQue), 0) ->
  True = allN(getNs($40:OnBPQue), 0)
```

Listing 12: *Representative ABP open obligation.*

The session records the required lemma as follows:

```
DM-Check> add lemma (allN(M:iNat ; NS:iSet, N:iNat) = True) -> \
>               (M:iNat = N:iNat) /\ (allN(NS:iSet, N:iNat) = True) .

  Invariant satisfied with the following proof obligations:

    True = allN(M:iNat ; NS:iSet, N:iNat) -> True = allN(NS:iSet, N:iNat) /\
    N:iNat = M:iNat
```

Listing 13: *ABP lemma added to the DM-Check session.*

Thus, the DM-Check part of the proof is closed modulo the displayed NuITP obligation. The corresponding NuITP proof material is distributed with the ABP example on the official DM-Check web site and is not reproduced here.

This example illustrates a common situation in deductive model checking: the symbolic model checker exposes the invariant structure, while the remaining semantic facts about recursively defined predicates are delegated to NuITP.

9.3 QLOCK

The QLOCK example models a queue-based mutual exclusion protocol. A state has the form

$$\langle U \mid W \mid C \mid Q \rangle,$$

where U, W, and C are multisets of process identifiers and Q is a queue. The function `l2ms` maps a list to the corresponding multiset, while `set` and `non-mt` express well-formedness conditions over multisets.

The session selects the module `QLOCK`:

```
DM-Check> set module QLOCK .

  Module QLOCK is now active.
```

The invariant has two disjuncts. The first describes states with an empty critical section; the second describes states with one process in the critical section and that process at the head of the queue.

```
DM-Check> check ind-invariant \
>     ((< U:MSet | W:MSet | mt | Q:List >) | \
>         (non-mt(U:MSet W:MSet) = tt) \
>          /\ (set(U:MSet W:MSet) = tt) \
>          /\ (l2ms(Q:List) = W:MSet)) \
>     \/ \
>     ((< V:MSet | T:MSet | i:Nat | i:Nat ; Q':List >) | \
>         (non-mt(V:MSet T:MSet i:Nat) = tt) \
>          /\ (set(V:MSet T:MSet i:Nat) = tt) \
>          /\ (l2ms(i:Nat ; Q':List) = T:MSet i:Nat)) .

  Invariant could not be proved.
```

Listing 14: *QLOCK invariant check.*

The open implications are about the auxiliary function `l2ms`. The first added lemma decomposes a list whose multiset image is known:

```
DM-Check> add lemma l2ms(X:Nat ; L:List) = (S:MSet X:Nat) -> \
                    l2ms(L:List) = S:MSet .

  Invariant could not be proved.
```

The second added lemma states that appending an element to the end of a list adds that element to the corresponding multiset:

```
DM-Check> add lemma l2ms(L:List) = S:MSet -> \
                    l2ms(L:List ; X:Nat) = (S:MSet X:Nat) .

  Invariant satisfied with the following proof obligations:

    S:MSet = l2ms(L:List) -> l2ms(L:List ; X:Nat) = X:Nat S:MSet
    l2ms(X:Nat ; L:List) = X:Nat S:MSet -> S:MSet = l2ms(L:List)
```

The corresponding NuITP proof material is distributed with the QLOCK example on the official DM-Check web site and is not reproduced here.

Initial containment is checked by proving that states of the form `< S:NeMSet | mt | mt | nil >`, under the condition `set(S) = tt`, are covered by the invariant.

```
DM-Check> check \
>     ((< S:NeMSet | mt | mt | nil >) | (set(S:NeMSet) = tt)) \
```

```
> subsumed by \
>     ((< U:MSet | W:MSet | mt | Q:List >) | \
>         (non-mt(U:MSet W:MSet) = tt) \
>          /\ (set(U:MSet W:MSet) = tt) \
>          /\ (l2ms(Q:List) = W:MSet)) \
>     \/ \
>     ((< V:MSet | T:MSet | i:Nat | i:Nat ; Q':List >) | \
>         (non-mt(V:MSet T:MSet i:Nat) = tt) \
>          /\ (set(V:MSet T:MSet i:Nat) = tt) \
>          /\ (l2ms(i:Nat ; Q':List) = T:MSet i:Nat)) .

  Subsumption satisfied with the following proof obligations:

    S:MSet = l2ms(L:List) -> l2ms(L:List ; X:Nat) = X:Nat S:MSet
    l2ms(X:Nat ; L:List) = X:Nat S:MSet -> S:MSet = l2ms(L:List)
```

Listing 15: *QLOCK initial containment.*

The two `l2ms` lemmas added before are kept as lemmas of the session, so the verdict lists them as proof obligations.

Deadlock freedom is then checked by showing that the invariant is contained in the symbolic set described by the left-hand sides of the rules. The first subsumption attempt leaves one open constrained term:

```
  Constrained terms on the left that could not be subsumed:

Term 4:
  < $17:MSet | $18:MSet | mt | $16:List >
Matching: no matching found
Constraint 4:
  non-mt($17:MSet $18:MSet) = tt /\ set($17:MSet $18:MSet) = tt /\ l2ms(
    $16:List) = $18:MSet
```

Listing 16: *QLOCK deadlock-freedom query before transformations.*

The remaining node is closed by semantic transformations. First, variant unification exposes the cases in which the multiset `$17:MSet $18:MSet` is non-empty:

```
DM-Check> vunify non-mt($17:MSet $18:MSet) =? tt in 4 .

  Variant unification in term 4: new terms 5, 6 and 7.

  Constrained terms on the left that could not be subsumed:

Term 5:
  < $19:NeMSet | mt | mt | $20:List >
Matching: no matching found
Constraint 5:
  set($19:NeMSet) = tt /\ l2ms($20:List) = mt

Term 6:
  < mt | $21:NeMSet | mt | $22:List >
Matching: no matching found
Constraint 6:
  set($21:NeMSet) = tt /\ l2ms($22:List) = $21:NeMSet

Term 7:
  < $23:NeMSet | $24:NeMSet | mt | $25:List >
Matching: no matching found
```

```
Constraint 7:
  set($23:NeMSet $24:NeMSet) = tt /\ l2ms($25:List) = $24:NeMSet
```

Constructor case splitting removes the cases in which the first process multiset is non-empty and therefore already enabled by the `r2w` rule:

```
DM-Check> case to 5 on $19:NeMSet with m:Nat ;; k:Nat S:NeMSet .

  Case split on term 5: every resulting case is closed.

  Constrained terms on the left that could not be subsumed:

Term 6:
  < mt | $21:NeMSet | mt | $22:List >
Matching: no matching found
Constraint 6:
  set($21:NeMSet) = tt /\ l2ms($22:List) = $21:NeMSet

Term 7:
  < $23:NeMSet | $24:NeMSet | mt | $25:List >
Matching: no matching found
Constraint 7:
  set($23:NeMSet $24:NeMSet) = tt /\ l2ms($25:List) = $24:NeMSet

DM-Check> case to 7 on $23:NeMSet with m:Nat ;; k:Nat S:NeMSet .

  Case split on term 7: every resulting case is closed.

  Constrained terms on the left that could not be subsumed:

Term 6:
  < mt | $21:NeMSet | mt | $22:List >
Matching: no matching found
Constraint 6:
  set($21:NeMSet) = tt /\ l2ms($22:List) = $21:NeMSet
```

The remaining term requires exposing the structure of the list whose multiset is non-empty. This is done by narrowing `l2ms($22:List)`. The narrowing step initially produces an empty-list branch whose constraint requires a non-empty multiset to be equal to `mt`. This branch is automatically proved unsatisfiable and discarded by DM-Check. Since node identifiers are assigned before constraint checking, only `Term 9` is displayed (identifier 8 belonged to the discarded branch). The remaining branch is then handled by variable abstraction and variant unification.

```
DM-Check> narrow 6 on l2ms($22:List) .

  Narrowing of a subterm in term 6: new term 9 (1 case closed).

  Constrained terms on the left that could not be subsumed:

Term 9:
  < mt | $23:NeMSet | mt | $24:Nat ; $25:List >
Matching: no matching found
Constraint 9:
  set($23:NeMSet) = tt /\ $24:Nat l2ms($25:List) = $23:NeMSet

DM-Check> va l2ms($25:List) with $26:MSet in 9 .
```

```
  Variable abstraction in term 9: new term 10.

  Constrained terms on the left that could not be subsumed:

Term 10:
  < mt | $26:NeMSet | mt | $27:Nat ; $28:List >
Matching: no matching found
Constraint 10:
  tt = set($26:NeMSet) /\ $29:MSet = l2ms($28:List) /\ $26:NeMSet = $27:Nat
    $29:MSet

DM-Check> vunify $26:NeMSet =? $27:Nat $29:MSet in 10 .

  Variant unification in term 10: every resulting case is closed.

  Subsumption satisfied with the following proof obligations:

    S:MSet = l2ms(L:List) -> l2ms(L:List ; X:Nat) = X:Nat S:MSet
    l2ms(X:Nat ; L:List) = X:Nat S:MSet -> S:MSet = l2ms(L:List)
```

Listing 17: *QLOCK final transformation sequence.*

This example shows that a failed subsumption check is not necessarily a failed property. Narrowing exposes the relevant list structure, while automatic constraint checking removes the unsatisfiable empty-list branch. The remaining pattern is closed by variable abstraction and variant unification. The two `l2ms` lemmas added for the invariant proof are still lemmas of the session, so the verdict lists them as proof obligations.

9.4 Lamport's Bakery Protocol

Lamport's Bakery protocol is used to illustrate invariant generation. The system module `BAKERY` defines the transition rules, and the module `BAKERY-AUX` combines the transition system with auxiliary functions for tickets and intervals. The session begins by selecting the auxiliary module:

```
DM-Check> set module BAKERY-AUX .

  Module BAKERY-AUX is now active.
```

Configurations have the form

$$N \mid M \mid PS,$$

where N is the dispenser counter, M is the currently served ticket, and PS is a multiset of processes. A process may be idle, waiting with a ticket, or in the critical section. The transition rules are the usual wake-up, critical entry, and exit rules:

```
rl [wake] : N | M | [idle] ; PS
 => 1 + N | M | [wait(N)] ; PS [narrowing] .

rl [crit] : N | M | [wait(M)] ; PS
 => N | M | [crit(M)] ; PS [narrowing] .

rl [exit] : N | M | [crit(M)] ; PS
 => N | 1 + M | [idle] ; PS [narrowing] .
```

The auxiliary part of the specification records the ticket information carried by waiting and critical processes. The invariant generated below is expressed using the interval function `i` and the summarizing function `tickets`:

```
op i : iNat iNat -> MSet [rpo 23] .

eq i(M + N,N) = null .
eq i(N,(1 + N + M)) = i(N,(N + M)), e(N + M) .

op tickets : ProcSet -> MSet [rpo 32] .
eq tickets(IS) = null .
eq tickets([idle] ; PS) = tickets(PS) .
eq tickets([wait(N)] ; PS) = e(N), tickets(PS) .
eq tickets([crit(N)] ; PS) = e(N), tickets(PS) .
```

9.4.1 Starting from the initial states

The invariant-generation session starts from the symbolic initial states in which both counters agree and all processes are idle:

```
DM-Check> generate ind-invariant \
>     (N:iNat | N:iNat | IS:ProcIdleSet) | true .

  Invariant could not be proved.

Parent 1
Term:
  $2:iNat | $2:iNat | $1:ProcIdleSet
Constraint:
  true

Child 2 (Parent 1)
Last transformation: narrowing
Rule: wake
Substitution:
  $1:ProcIdleSet --> $3:ProcIdleSet ; [idle]
  $2:iNat --> $4:iNat
Term:
  (1 + $4:iNat) | $4:iNat | $3:ProcIdleSet ; [wait($4:iNat)]
Matching: no matching found
Constraint:
  true
```

After one more generated level, the frontier contains a pattern with two waiting processes and a pattern with one critical process:

```
DM-Check> generate next level .

  Invariant could not be proved.

Child 3 (Parent 2)
Last transformation: narrowing
Rule: wake
Substitution:
  $3:ProcIdleSet --> $6:ProcIdleSet ; [idle]
  $4:iNat --> $5:iNat
Term:
  (1 + 1 + $5:iNat) | $5:iNat | $6:ProcIdleSet ; [wait($5:iNat)] ; [wait(1 +
    $5:iNat)]
Matching: no matching found
Constraint:
  true
```

```
Child 4 (Parent 2)
Last transformation: narrowing
Rule: crit
Substitution:
  $3:ProcIdleSet --> $8:ProcIdleSet
  $4:iNat --> $7:iNat
Term:
  (1 + $7:iNat) | $7:iNat | $8:ProcIdleSet ; [crit($7:iNat)]
Matching: no matching found
Constraint:
  true
```

The repeated waiting pattern is not accumulated literally. Instead, it is generalized into a symbolic region relating the process set with the interval of issued tickets:

```
DM-Check> generalize 3 to \
>       (M:iNat + N:iNat | N:iNat | WS:ProcWaitSet) | \
>              (tickets(WS:ProcWaitSet) = i(N:iNat, M:iNat + N:iNat)) .

  Generalization of child 3: new child 5.

  Invariant could not be proved.

Child 4 (Parent 2)
Last transformation: narrowing
Rule: crit
Substitution:
  $3:ProcIdleSet --> $8:ProcIdleSet
  $4:iNat --> $7:iNat
Term:
  (1 + $7:iNat) | $7:iNat | $8:ProcIdleSet ; [crit($7:iNat)]
Matching: no matching found
Constraint:
  true

Child 5 (from child 3)
Last transformation: generalization
Rule: none
Substitution:
  none
Operation data:
  1. ($9:iNat + $10:iNat) | $10:iNat | $11:ProcWaitSet
  2. tickets($11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat)
Term:
  ($9:iNat + $10:iNat) | $10:iNat | $11:ProcWaitSet
Matching: no matching found
Constraint:
  tickets($11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat)
```

9.4.2 Closing the generated frontier

The generation process then discovers the critical-region component of the invariant. The next level contains a critical process whose ticket is the current served ticket:

```
DM-Check> generate next level .

  Invariant could not be proved.
```

```
Parent 4
Term:
  (1 + $7:iNat) | $7:iNat | $8:ProcIdleSet ; [crit($7:iNat)]
Constraint:
  true

Parent 5
Term:
  ($9:iNat + $10:iNat) | $10:iNat | $11:ProcWaitSet
Constraint:
  tickets($11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat)

Child 9 (Parent 5)
Last transformation: narrowing
Rule: crit
Substitution:
  $10:iNat --> $21:iNat
  $11:ProcWaitSet --> $19:ProcWaitSet ; [wait($21:iNat)]
  $9:iNat --> $20:iNat
Term:
  ($20:iNat + $21:iNat) | $21:iNat | $19:ProcWaitSet ; [crit($21:iNat)]
Matching: no matching found
Constraint:
  tickets($19:ProcWaitSet ; [wait($21:iNat)]) = i($21:iNat, $20:iNat +
    $21:iNat)
```

One more level exposes the two remaining situations that require user action: an exit from the critical section, and an impossible state with two critical processes carrying the same ticket.

```
DM-Check> generate next level .

  Invariant could not be proved.

Child 11 (Parent 9)
Last transformation: narrowing
Rule: exit
Substitution:
  $19:ProcWaitSet --> $27:ProcWaitSet
  $20:iNat --> $25:iNat
  $21:iNat --> $26:iNat
Term:
  ($25:iNat + $26:iNat) | 1 + $26:iNat | $27:ProcWaitSet ; [idle]
Matching: no matching found
Constraint:
  tickets($27:ProcWaitSet ; [wait($26:iNat)]) = i($26:iNat, $25:iNat +
    $26:iNat)

Child 12 (Parent 9)
Last transformation: narrowing
Rule: crit
Substitution:
  $19:ProcWaitSet --> $30:ProcWaitSet ; [wait($29:iNat)]
  $20:iNat --> $28:iNat
  $21:iNat --> $29:iNat
Term:
  ($28:iNat + $29:iNat) | $29:iNat | $30:ProcWaitSet ; [crit($29:iNat)] ; [
```

```
    crit($29:iNat)]
Matching: no matching found
Constraint:
  tickets(($30:ProcWaitSet ; [wait($29:iNat)]) ; [wait($29:iNat)]) = i(
    $29:iNat, $28:iNat + $29:iNat)
```

The double-critical child is closed as empty. The exit child is split according to the possible constructor forms of the distance variable; one branch is empty, and the other two branches become proof obligations:

```
DM-Check> empty 12 .

  Child 12 closed as empty (obligation added: its constraint -> false).

DM-Check> case to 11 on $25:iNat with 0 ;; 1 ;; (1 + #N:iNzNat) .

  Case split on child 11: new children 13, 14 and 15.

DM-Check> empty 13 .

  Child 13 closed as empty (obligation added: its constraint -> false).

DM-Check> lemma 14 .

  Child 14 closed (obligation added: its open implication).

DM-Check> lemma 15 .

  Child 15 closed (obligation added: its open implication).

  The following invariant is satisfied:

    ((($9:iNat + $10:iNat) | $10:iNat | $11:ProcWaitSet) | (tickets(
    $11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat))) \/
    ((($20:iNat + $21:iNat) | $21:iNat | $19:ProcWaitSet ; [crit($21:iNat)]) |
    (tickets($19:ProcWaitSet ; [wait($21:iNat)]) = i($21:iNat, $20:iNat +
    $21:iNat)))

  with the following proof obligations:

    null = e($28:iNat), tickets($29:ProcWaitSet) -> false
    e($33:iNat), tickets($34:ProcWaitSet) = e($33:iNat + $32:iNzNat), i(
    $33:iNat, $33:iNat + $32:iNzNat) -> tickets($34:ProcWaitSet) = i(1 +
    $33:iNat, 1 + $33:iNat + $32:iNzNat)
    e($30:iNat) = e($30:iNat), tickets($31:ProcWaitSet) -> null = tickets(
    $31:ProcWaitSet)
    tickets(($30:ProcWaitSet ; [wait($29:iNat)]) ; [wait($29:iNat)]) = i(
    $29:iNat, $28:iNat + $29:iNat) -> false
```

The generated symbolic tree is therefore closed modulo the four displayed NuITP obligations. These obligations express the algebraic facts about tickets, intervals, and multiset cancellation that justify the emptiness and folding steps.

9.4.3 Discharging the obligations in NuITP

The Bakery session leaves four NuITP obligations. They are part of the full verification argument, but the corresponding NuITP proof material belongs to the external theorem prover and is therefore not reproduced in this manual. It is distributed with the Bakery example on

the official DM-Check web site.

9.4.4 Derived ticket invariant and mutual exclusion

After generation, the obtained symbolic invariant is related to a more compact ticket invariant. This step is a containment proof by subsumption:

```
DM-Check> check ($9:iNat + $10:iNat | $10:iNat | $11:ProcWaitSet) | \
>         tickets($11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat) \/ \
>       ($20:iNat + $21:iNat | $21:iNat | $19:ProcWaitSet ; [crit($21:iNat)]) | \
>         tickets($19:ProcWaitSet ; [wait($21:iNat)]) = \
>         i($21:iNat, $20:iNat + $21:iNat) \
> subsumed by (M:iNat + N:iNat | N:iNat | PS:ProcSet) | \
>         tickets(PS:ProcSet) = i(N:iNat, N:iNat + M:iNat) .

  Subsumption satisfied.
```

Finally, mutual exclusion is proved by intersecting the generated invariant with the symbolic error pattern containing two critical processes:

```
DM-Check> intersect \
>       ($9:iNat + $10:iNat | $10:iNat | $11:ProcWaitSet) | \
>         tickets($11:ProcWaitSet) = i($10:iNat, $9:iNat + $10:iNat) \/ \
>       ($20:iNat + $21:iNat | $21:iNat | $19:ProcWaitSet ; [crit($21:iNat)]) | \
>         tickets($19:ProcWaitSet ; [wait($21:iNat)]) = \
>         i($21:iNat, $20:iNat + $21:iNat) \
> with \
>       (M:iNat | N:iNat | [crit(K:iNat)] ; [crit(L:iNat)] ; PS:ProcSet) | true .

  No intersection.
```

This proves that the verified invariant is disjoint from the set of states with two critical processes. Since the invariant was generated from the symbolic initial states and the remaining obligations are discharged in the accompanying NuITP material, the Bakery mutual-exclusion property follows.

9.5 Other Examples

Each distributed example should contain the Maude specification, the DM-Check transcript or command listing, and, when needed, the external NuITP proof material. Include a plain-text transcript when the published manual refers to node identifiers.

The public material for DM-Check also includes additional examples, such as ordinary Readers–Writers, Fair Readers–Writers, the conditional-rule `qlock-cond` variant, Token-Ring, CHC, and Tree Processing. These examples follow the same structure: a Maude module defining the transition system, a `.dmc` command script containing the invariant or generation commands, and, when needed, a `.nuitp` file for the external semantic obligations.

The `qlock-cond` example places the guard `non-mt(C) = ff` on the `w2c` rule. Its invariant does not require the auxiliary `set` constraint; its deadlock-freedom target includes the guard. The supplied script checks this invariant with the two `l2ms` lemmas and then checks initial containment. Immediately before the mutual-exclusion intersection it executes `reset lemmas .`, because those lemmas are needed only for the invariant argument. The intersection therefore reports `No intersection.`, and the subsequent absence-of-blocking proof finishes with `Subsumption satisfied.`, without obligations.

In the current Fair Readers–Writers example, deadlock freedom no longer requires the auxiliary `enabled` predicate or a corresponding external NuITP proof. The remaining symbolic cases can be handled directly in DM-Check by applying two `case` commands.

Part IV:
Troubleshooting

10 Troubleshooting and Limitations

10.1 No Active Module

Several commands require an active Maude module. If no module has been selected with `set module`, DM-Check prints the same error independently of the command that triggered it.

For example, asking for the active module before selecting one gives:

```
DM-Check> show module .

  Error: module not loaded.
```

The same error is printed if a verification command is issued without an active module:

```
DM-Check> check ind-invariant < N:Natural , 0 > | true .

  Error: module not loaded.
```

The solution is to load the intended Maude specification and then select the module explicitly:

```
DM-Check> set module M .
```

10.2 The Module Is Not Activated

A loaded module is nevertheless rejected if its flattened form contains a use of the built-in `_==_` or `_=/=_`, an `owise` equation, or a conditional rule with a non-equational condition, or declares a membership axiom with `mb` or `cmb`. The diagnostic lists every offending statement. The preceding active module, if any, remains active, and a corrected module can be selected immediately without restarting DM-Check.

10.3 A Command Using == Is Rejected

The built-in comparison operators evaluate syntactically on terms containing variables and may therefore discard symbolic states. Commands containing them are rejected. Define an explicit predicate by variant equations and use that predicate in the constraint instead.

10.4 NuITP Could Not Process a Formula

If NuITP cannot process a formula needed by a command, the event is an error, not an open obligation. The command is undone and the previous proof state is restored. A running `load` script consequently stops at this point.

10.5 The Command Produced an Ill-formed Result

An internal result that fails the tool's well-formedness check is reported as an error and is not installed as the current result. Preserve the diagnostic and the command sequence when reporting the problem; the session itself remains available.

10.6 Unification May Be Incomplete

When Maude reports that it may not have computed all unifiers or narrowing steps—for example, modulo an `assoc` operator without `comm`—the affected command is rejected. No conclusion may be drawn from a partial result. Reformulate the equational theory, for example with an appropriate identity or commutativity attribute, or choose another representation.

10.7 A Script Was Aborted

`load` stops at the first error, including parse failures, rejected commands, unprocessed NuITP formulae, and ill-formed or incomplete results. Warnings do not abort a script. Correct the failing line and rerun the script from the beginning; nested `load` commands are not allowed.

10.8 A Case Split Adds Coverage Obligations

A warning after `case` means that the split was applied but the generator set did not establish complete coverage. Each displayed missing constructor pattern has an associated exclusion obligation. A sort for which the active module declares no constructors necessarily produces such obligations; review the constructor declarations or discharge every exclusion explicitly.

10.9 The Invariant Is Not Proved

An unsuccessful `check ind-invariant` command does not by itself mean that the invariant is false. It may mean that a generated child is empty, that an auxiliary lemma is missing, that a case split is needed, or that the invariant is not yet strong enough.

Inspect the result and classify the open node. If the printed child contains an obviously contradictory constraint, the next step is an emptiness proof. If the child resembles an existing invariant disjunct but with a different variable naming or hidden equality, inspect the matcher and consider a transformation. If the child has a genuinely new symbolic shape, the invariant may need a new disjunct or a generalization.

10.10 Subsumption Fails

Failure of subsumption should be interpreted carefully. If there is no matcher, the pattern may still be contained in a union of patterns. If there is a matcher but the implication is open, the missing argument is probably an inductive theorem about the constraint.

10.11 The Parser Rejects a Pattern

Parsing problems are common when constrained patterns contain user-defined mixfix operators. The first suggestions to try are:

- add parentheses around the state pattern;
- add parentheses around Boolean constraints;
- annotate variables with their sorts;
- check that the active module is the one declaring the operators;
- reintroduce disjuncts one at a time to check which of them is failing to parse.

Because Maude syntax is user-definable, the parser may need more parentheses than the mathematical notation suggests.

10.12 A Rule Is Not Used

If a transition that should be explored does not seem to generate any child, first distinguish between unconditional and conditional rules. An unconditional rule must be marked with `[narrowing]`; otherwise DM-Check activates the module with a warning and does not use that rule for symbolic exploration. Conditional rules must have only equational conditions; matching, rewrite, and membership conditions cause `set module` to reject the module rather

than silently ignore the rule. Membership axioms declared with `mb` or `cmb`, whether local or imported, likewise cause module activation to be rejected because Maude's unification ignores them. Also check that the rule is in the flattened active module and that its left-hand side has the expected state sort.

10.13 Invariant Generation Does Not Terminate

Invariant generation may keep producing new frontier patterns. This typically means that the current symbolic description is too concrete. Consider using semantic transformations or generalization when:

- the same shape reappears with larger counters;
- queues or multisets grow in a regular way;
- equality information is hidden in constraints;
- several frontier patterns are instances of a common abstraction.

Do not keep generating levels blindly. If the same pattern family reappears with larger parameters, the useful information has already been found: the proof needs an abstraction. At that point, inspect the recurring structure and decide whether the next step is a semantic transformation or a genuine generalization.

10.14 The Generated Invariant Is Too Weak

Sometimes the generated approximation is inductive but does not imply the desired property, because it over-approximates too much. This can happen after an aggressive generalization. The remedy is to strengthen the symbolic description: add a constraint, split a pattern, or replace a broad generalization by a more precise one.

This is the main tradeoff in invariant generation. A stronger invariant may be harder to make inductive, but a weaker invariant may be useless for the final safety proof.

10.15 The Deprecated `recheck` Command Is Used

`recheck` is retained only for compatibility diagnostics. Current commands that require repetition perform it automatically. Remove `recheck` from new transcripts and record the exact deprecation warning in Appendix B.

10.16 The Generated Invariant Is Too Strong

A candidate invariant is too strong when it excludes initial or reachable states. This may appear either as a failure of initial coverage, $A \not\subseteq B$, or as a transition-closure failure, $R[B] \not\subseteq B$. In a generation workflow, it appears when a proposed symbolic region does not cover a frontier pattern that must be included.

The usual fix is to add the missing symbolic region or weaken a constraint. The open child is the best starting point.

Part V:
Appendices

A Quick Reference

A.1 Minimal Checking Workflow

```
set module <module-name> .

check <initial-patterns> subsumed by <candidate-invariant> .

check ind-invariant <candidate-invariant> .

show last .
show obligations .

intersect <candidate-invariant> with <error-patterns> .
```

A.2 Minimal Generation Workflow

```
set module <module-name> .

generate ind-invariant <initial-patterns> .
--- or: gen ind-invariant <initial-patterns> .

generate next level .
--- or: gen next level .

--- Repeat generation, transformation, lemma, case, emptiness,
--- containment, or generalization steps until the frontier is empty.

show last .
show obligations .
```

A.3 Questions to Ask When a Proof Stops

1. Is the active module correct?
2. Are all relevant unconditional transition rules marked with `[narrowing]`, and do conditional rules use the supported equational condition syntax?
3. Is the open child empty?
4. Is it covered by another child or by a union rather than one disjunct?
5. Was a matcher found but its constraint implication left open?
6. Does the pattern need variant unification, abstraction, substitution, or a case split?
7. Is a reusable lemma about an auxiliary function missing?
8. Is the frontier a recurring family that should be generalized?
9. Does the final safety query require a stronger invariant?

A.4 Reproducibility Checklist

- Exact Maude, DM-Check, and NuITP releases.
- Operating system and architecture.

- Maude module with constructor declarations, equations, and transition rules.
- Explicit list of unconditional rules carrying the `[narrowing]` attribute and of supported conditional rules.
- Initial constrained pattern or disjunction.
- Candidate or generated invariant.
- Error pattern or positive target property.
- Complete DM-Check command transcript or command listing.
- A saved `.dmc` proof script, replayed with `load`; remember that it records the route to the result but does not certify assumed lemmas.
- Unedited console transcript for commands that create node identifiers.
- List of generated proof obligations.
- A replayable `.nuitp` file, with one command per line, for every non-automatic obligation.
- Explanation of each case split, transformation, containment claim, and generalization.
- Final proof status: automatic, modulo discharged obligations, open, or inconclusive.

B Legacy Syntax and Migration

Older material may use command forms that are no longer the preferred current syntax. This appendix records the main migrations needed to update old transcripts and command examples.

The following forms have been removed; they are not aliases accepted by the current grammar.

Article or legacy syntax	Current syntax
`check-inv in M : B .`	`set module M .` followed by `check ind-invariant B .`
`check in M : A subsumed-by B .`	`set module M .` followed by `check A subsumed by B .`
`intersect in M : A with B .`	`set module M .` followed by `intersect A with B .`
`add lemma in M : L .`	`set module M .` followed by `add lemma L .`
`case i of p on X with G .`	`case to i on X with G .`
`empty i of p .`	`empty i .`
`lemma i of p .`	`lemma i .` or `lemma i match M .`
`contained i of p .`	`contained i .`

B.1 Module-Qualified Legacy Commands

Older transcripts may place the module name inside each command:

```
check-inv in M : <disjunction> .
check in M : <disjunction-1> subsumed-by <disjunction-2> .
intersect in M : <disjunction-1> with <disjunction-2> .
```

These module-qualified forms are no longer accepted. Select the module once and then use the session-oriented commands:

```
set module M .
check ind-invariant <disjunction> .
check <disjunction-1> subsumed by <disjunction-2> .
intersect <disjunction-1> with <disjunction-2> .
```

B.2 Accepted Aliases and Deprecated Spellings

The current interface accepts several long and short command forms. New transcripts should prefer the current forms used in Chapter 8; the legacy forms below are documented to help migrate older transcripts.

```
quit
q
help .
h .
show module .
show lemmas .
show last .
show obligations .
set module M .
```

The preferred invariant-checking command is:

```
check ind-invariant B .
```

The older spelling is still accepted:

```
check invariant B .
```

Invariant generation admits both the long and short forms:

```
generate ind-invariant B .
gen ind-invariant B .
generate next level .
gen next level .
```

Proof-tree management uses node identifiers, with an optional matcher identifier for `lemma`:

```
empty i .
lemma i .
lemma i match M .
contained i .
```

Older help text or transcripts may use parent-qualified forms such as:

```
empty i of p .
lemma i of p .
contained i of p .
```

These parent-qualified forms are no longer accepted; the node identifier is sufficient in the current proof tree.

The current variable-abstraction syntax is:

```
variable abstraction X with V in i .
va X with V in i .
select variable abstraction n .
select va n .
```

The following older forms are still accepted, although the forms above are preferred in new material:

```
variable abstraction to i on X with V .
va to i on X with V .
```

The current substitution syntax is:

```
substitution to i on X with t .
sb to i on X with t .
```

The current variant-unification syntax is:

```
variant unify t1 =? t2 in i .
vunify t1 =? t2 in i .
```

The following older forms are still accepted, although the forms above are preferred in new material:

```
variant unify to i with t1 =? t2 .
vunify to i with t1 =? t2 .
```

The current narrowing syntax is:

```
narrowing i on t .
narrow i on t .
```

The following older forms are still accepted, although the forms above are preferred in new material:

```
narrowing to i on t .
narrow to i on t .
```

The current generalization and Boolean-splitting forms are:

```
generalize i to P .
split i with t .
```

The `recheck` command is deprecated. Commands such as `add lemma` repeat the required operation automatically:

```
DM-Check> recheck .

  Warning: command deprecated. Recheck is now performed automatically when
    required.
```

B.3 Migration Rules

- Select the module once with `set module`.
- Replace `check-inv` and `check invariant` by `check ind-invariant`.
- Replace `subsumed-by` by the two-token form `subsumed by`.
- Use either `generate ind-invariant` or its short alias `gen ind-invariant`.
- Replace old `vunify`, `va`, and `narrow` forms by the current forms documented above.
- Use `lemma i match M .` when only one matcher implication of a node should be recorded as a lemma.
- Replace the removed parent-qualified forms by `empty i .`, `lemma i .`, and `contained i ..`

- Remove `recheck`.
- Re-record every proof-tree transcript because node identifiers may change.

References

[1] K. Bae, S. Escobar, R. López-Rueda, J. Meseguer, and J. Sapiña. "Verifying Invariants by Deductive Model Checking". In: *Rewriting Logic and Its Applications*. Vol. 14953. Lecture Notes in Computer Science. Springer, 2024, pp. 3–21. DOI: 10.1007/978-3-031-65941-6_1.

[2] K. Bae, S. Escobar, R. López-Rueda, J. Meseguer, and J. Sapiña. "DM-Check: Verifying Invariants of Concurrent Systems by Deductive Model Checking". In: *Journal of Logical and Algebraic Methods in Programming* 149 (2026), p. 101107. DOI: 10.1016/j.jlamp.2025.101107.

[3] K. Bae, S. Escobar, R. López-Rueda, J. Meseguer, and J. Sapiña. "Generating Invariants by Deductive Model Checking". In: *Pre-proceedings of the 16th International Workshop on Rewriting Logic and its Applications*. Pre-proceedings. 2026, pp. 96–114. URL: https://wrla2026.github.io/preproceedings/preproceedings.pdf.

[4] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott. *Maude Manual (version 3.5.1)*. Tech. rep. SRI International Computer Science Laboratory, 2025. URL: http://maude.cs.illinois.edu.

[5] F. Durán, S. Escobar, J. Meseguer, and J. Sapiña. *NuITP: An Inductive Theorem Prover for Maude Equational Theories*. Alpha 36. Universitat Politècnica de València. Mar. 2026. URL: https://nuitp.webs.upv.es/download/NuITP_Manual.pdf.

[6] DM-Check Team. *DM-Check Web Site*. https://dmcheck.webs.upv.es/. Official web site. 2026.

[7] NuITP Team. *NuITP Web Site*. https://nuitp.webs.upv.es/. Official web site. 2026.

[8] The Maude Team. *Maude Web Site*. https://maude.cs.illinois.edu/. Official web site. 2026.

[9] F. Durán, S. Escobar, J. Meseguer, and J. Sapiña. "NuITP: An Inductive Theorem Prover for Equational Program Verification". In: *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming*. 2024, 6:1–6:11. DOI: 10.1145/3678232.3678236.