Nitpick

A Checker for Software Specifications
Reference Manual
Version 1.0 / January 1996

Nitpick Reference Manual

Daniel Jackson & Craig A. Damon
Carnegie Mellon University
Pittsburgh, Pennsylvania

Copyrights and credits

Reference manual and Nitpick tool
Copyright 1995, 1996, Carnegie Mellon University

Graph generator (source code compiled within Nitpick)
Copyright 1984–1993, Brendan C. McKay

Algorithms: Daniel Jackson, Craig Damon & Somesh Jha

Design: Craig Damon & Daniel Jackson

Implementation: Craig Damon & Jake McGuire

Documentation: Daniel Jackson & Craig Damon

Nitpick logo: Christina M. Grimes

Contacting us

Comments, questions & bug reports welcome

Email nitpick@cs.cmu.edu

Nitpick home page: http://cs.cmu.edu/~nitpick

Contents

1 Getting Started

2 User Guide

3 Command Reference

4 NP Language Description

5 Glossary

Chapter 1
Getting Started

1.1 What is Nitpick?

Nitpick is a tool for checking software specifications. You give it a specification along with a claim that the specification has some property. Nitpick evaluates the claim by searching for counterexamples. If it finds one, the specification does not satisfy the claim; if it does not, either the specification satisfies the claim, or Nitpick did not examine enough cases to find a counterexample.

Because errors can be missed, Nitpick is really a 'specification debugger'. But unlike conventional debugging and testing tools, Nitpick can examine billions of cases, so in practice it finds errors more reliably. And there's no need to specify individual cases to be tested. Instead you select a 'scope' that bounds the size of the cases to be examined, and Nitpick generates all cases within the scope.

Specifications and claims are written in Nitpick's own notation, NP. If you know a specification language such as Z, VDM or Larch, you should have no trouble learning it. If you already know Z, you'll find it even easier, since NP is almost a Z subset.

Nitpick can also perform a kind of limited simulation. It can generate instances of a state satisfying some invariants that you specify, or sample executions of an operation, even if you specified the operation implicitly.

1.2 The Nitpick Project

Nitpick is the product of an ongoing research project at Carnegie Mellon University. Both the tool and the NP language are still under development; we expect big improvements in both in the next few years.

The NP language is not nearly as expressive as Z, and the tool can only handle small scopes. So don't expect to take a large existing specification and just

run it through Nitpick. Think of an NP specification as a simplified model of the software you want to analyze, and the tool as a back-of-the-envelope calculator.

Despite its limitations, we think Nitpick is useful. It's the only tool we know of that can simulate implicit specifications and check properties completely automatically. We've found that it's much more fun writing specifications when you can simulate and check them as you go along. It's surprising how many errors you find this way.

Nitpick would make a nice teaching tool. The NP notation is much simpler than Z, and corresponds quite closely to the relational calculus. The interactive nature of Nitpick makes it easier to learn basic concepts, and to explore their subtleties in a way that is hard to do on paper. Students can use Nitpick to learn about sets, relations and functions; about invariants and relational specifications; and about theorems and counterexamples. The parts of Z missing would be higher-order types and the full schema calculus. We are beginning to develop course materials for teaching with Nitpick, and will be happy to share them with other educators.

1.3 Obtaining and Installing Nitpick

The Nitpick tool is currently available for the Macintosh only. It requires version 7.0 of the operating system or later. Instructions for obtaining the executable, a copy of this manual, and a collection of tutorial examples, may be found at http://cs.cmu.edu/~nitpick. Nitpick is not released in source form.

The distribution consists of two Nitpick executables – one for 68k and one for Power PC machines – and folders containing the manual and the examples. Just pick the appropriate executable and copy it with the other folders into a folder on your hard drive. The executable does not depend on any other files, except for a preferences file, which is created on the first execution and stored in the system preferences folder.

Specifications should be created in plain ascii text by an editor such as SimpleText. It is convenient to have both the editor and Nitpick open together, so that you can view and modify the specification as you check it.

1.4 How to Avoid Reading this Manual

If you've never written any formal specifications before, you are likely to find

that even after reading this manual you're not really sure what to do. You might want to read a short exposition of formal specification first, such as Chapter 1 of *The Z Notation: A Reference Manual*, J.M. Spivey, Prentice Hall, 1992, or "A specifiers introduction to formal methods", Jeannette Wing, IEEE Computer, 23(9), September 1990.

If you like to find things out for yourself, you might want to read just the first section of the user guide (2.1) and then dive straight into the tutorial examples (currently provided as files in the distribution, soon to be described fully in a tutorial guide). You might want to refer to the command reference (chapter 3) if you can't figure out how to use a Nitpick command.

If you're a Z guru, you might want to start with section 4.2, which compares NP to Z.

1.4

Chapter 2
User Guide

2.1 Short Overview

Using Nitpick is very straightforward. Write a specification in SimpleText and save it in a file. Open the Nitpick application; a transcript window will appear. Now select *Open Spec* from the *File* menu to load the file containing the specification. Nitpick will report any parsing errors in the transcript window. If there are none, two scrolling lists will appear, one containing the names of schemas in your specification file, and one containing the names of claims. If there are no claims in your specification, only the schema list will appear.

Now select *Edit File Scope* from the *Scope* menu. Nitpick will show you a list of given types in your specification, and for each, a default bound on the number of values it may assume. You can change the bounds for any given type. Close the scope dialogue. Now select a schema from the schema list or a claim from the claim list simply by clicking on it, and select *Run* from the *File* menu.

Nitpick will then start to generate cases. If you selected a schema, it will stop when it finds the first case satisfying the schema. We'll call such a case an *example* of the schema. To see the example, click once and examine the output of the transcript window. If you selected a claim, it will stop when it finds the first *counterexample*: a case in which the claim does not hold.

The term *instance*, used in some of Nitpick's dialog boxes and messages, refers to an example or counterexample when the distinction is irrelevant. The term *counter* is short for counterexample.

You can have Nitpick find more than one instance by changing the parameters in the *Limit Run* dialogue under the *Options* menu. You can select a scope for an individual claim or schema that overrides the scope for the file as a whole, and you can save and reload scopes. You can also turn off the reduction mechanisms that Nitpick uses to speed up the search. This will only make Nitpick go slower; it's primarily a research feature for measuring the effect of

the reductions. You can also display information about how Nitpick organized the search for a particular claim or schema.

2.2 Detailed Description

2.2.1 Specifications and Files

A specification to be analyzed and all its associated claims must be in the same file. You cannot put two different specifications in the same file, but by choosing different names for different versions of schemas, you can experiment with many versions of a single specification in the same file so long as they appear to Nitpick as parts of the same specification.

2.2.2 Selecting Scopes

All NP specifications have unbounded state spaces, since given types are assumed to be infinite sets, and so even a scalar of some given type can take on an infinite number of values. In contrast, the Nitpick tool always works within finite bounds selected by the user.

A *scope* associates each given type in a specification with a small positive integer. If a type *t* maps to the integer *k*, that means Nitpick will only consider cases containing no more than *k* elements of type *t*. The number of cases grows amazingly fast with the scope, so you should usually bound each type by 3 or 4; the default is 3.

Nitpick maintains a single scope for an entire file. By default, any claim or schema within the file will use this scope setting. However, you can override the settings for a particular schema or claim. If you subsequently change the file scope, any bounds explicitly set in the individual claim and schema scopes will not be reset; only claim and schema scopes that have not been edited will change to match the new file scope. If you want to reset all scopes, just quit and reopen Nitpick.

To change the scope for the entire file, select *Edit File Scope* from the *Scope* menu. The dialogue box presents a scrolling list of the given types along with their bounds. To change the bound for a type, highlight it (by clicking on it, or by using the up and down arrow keys), and then either click on a radio button or type a digit. You can also use the return key to move down the list.

To change the scope for an individual claim or schema, first select the claim or schema in one of the scrolling lists. Then select *Edit Schema/Claim Scope* from the *Scope* menu. It works just like the file scope dialogue box, but in addition

to bounding the size of the given types, you can restrict the enumeration of relation variables. Each relation variable appearing in the claim or schema is listed with the settings of four toggles: *F* means that only functions will be generated for the given variable; *I* means injections (relations whose transposes are functions); *T* means total relations; and *O* means onto, or surjective relations. To change any of these settings, first highlight a variable by clicking, or by typing a tab to move to the relation list and then using arrow keys, then either click on the toggle boxes to the right, or type one of the characters *F*, *I*, *T*, *O*.

The scope settings associated with a specification (both the file and individual claim and schema settings) can be saved by selecting *Save Scope* under the *Scope* menu, and reloaded by selecting *Load Scope*.

Note that these scope settings only affect the Nitpick enumeration; you should not think of them as part of the specification. If a variable *is* a function, then it should be declared as such in the specification. Nitpick will not let you choose a scope that is incompatible with the specification. For example, if a variable is a function, you cannot toggle *F* to make Nitpick enumerate non-function values.

2.2.3 Running Schemas and Claims

"Running" means generating examples of a schema or finding counterexamples to a claim. To run a claim or schema, select it in the scrolling list and then choose *Run* from the *File* menu. You can run all the claims in a file consecutively by selecting *Run All Claims*. If you want to stop Nitpick during an execution, select *Halt* from the *File* menu.

An execution window will appear with a progress bar showing how much of the space has been examined; this will give you some idea of how long the execution might take. When the run is complete, click anywhere to dismiss the execution window. The transcript window will show you the result of the execution.

2.2.4 Setting Run Limits

Nitpick's default is to run as far as the first example or counterexample and to display it. If none is found, it will run until the scope selected is exhausted. You can change this default behaviour by selecting *Limit Run* from the *Options* menu. In addition to selecting the number of instances to be found before termination, you can select how many of these are to be displayed. Nitpick will not let you specify that more instances are to be displayed than are to be found.

2.2.5 Reductions

When a run starts, the execution window shows (just above the progress bar) the total number of cases in the space. This is the number of cases that would need to be examined by a naive enumeration of every combination of values for the variables within the specified scope.

If you run a schema or claim to completion without finding an instance (or if you select 'find all' in the *Limit Run* dialog), you'll notice that the actual number of cases that Nitpick examines is usually much smaller than this total. The reduction factor increases with the scope and usually also with the complexity of the schema or claim; it typically ranges from 10 (that is, only 1 in 10 cases are examined) to 1 million.

Nitpick uses three mechanisms to reduce the space: derived variable analysis, short-circuit enumeration, and isomorph elimination.

All the reduction mechanisms are sound, so that if an instance exists within a scope, and would thus be found with the reduction mechanisms off, some instance will still be found with the mechanisms on. There is no guarantee that the same instance will be found, because one of the mechanisms – isomorph elimination – will skip instances that are relabellings of instances already seen, so only one instance out of an equivalence class will be found (and it may not be the one that would have been found first in a simple search).

The total number of cases shown above the progress bar already accounts for derived variable analysis, so the ratio of this total to the number of cases actually examined in a full search is the reduction factor achieved by short-circuiting and isomorph elimination.

You can turn any of the three reduction mechanisms on and off under the *Options* menu. They never increase the search space, and usually reduce it drastically, so the default mode has them all switched on.

2.2.5.1 Derived Variable Analysis

Derived variable analysis identifies variables that do not need to be enumerated but can instead be derived from the values of other variables. To see how this works, look at this claim:

$$(x = y) \Rightarrow (x = z)$$

It is obviously invalid: a counterexample can be made by making x and y the same value and z different. From the form of the claim, Nitpick can tell that in any counterexample, x must equal y. So instead of enumerating all combina-

tions of the three variables, it will enumerate just x and z and derive y from x. A list of derived variables for a claim or schema, along with expressions showing how they are obtained from enumerated variables, can be displayed by selecting *Derived Variables* from the *Display* menu.

2.2.5.2 Short-circuit Enumeration

Short-circuit enumeration prunes subtrees of the search tree that are known not to contain instances. Take a claim like

$$F(x, y) \Rightarrow G(x, y, z)$$

where F is some formula that uses only the variables x and y, and G uses only x, y and z. For this claim, a case is a value of x, a value of y and a value of z. To enumerate these cases, Nitpick walks a tree whose first level branches cover all the values of x, second level branches cover y and third level cover z. Each leaf of the tree gives one case. Now suppose that during the search, Nitpick has reached a node at the second level in which values have been assigned to x and y, but not yet to z, and that for these values, $F(x, y)$ is false. Obviously, the claim will be true whatever value of z is chosen, so there is no point enumerating any of them, and the whole subtree can be pruned. The trick in getting short-circuiting to work well is to pick a good order in which to enumerate the variables. In the short delay after selecting a claim or a schema, Nitpick is determining an enumeration order. The result can be seen by selecting *Enumerated Vars* from the *Display* menu; the order in which the variables are listed is the order in which they are enumerated.

Short-circuiting often works much better at some points in the search than at others, so you may notice some jumpiness in the progress bar, when a large subtree is pruned.

2.2.5.3 Isomorph Elimination

Isomorph elimination skips cases that are essentially relabellings of cases that have already been considered. Suppose we are checking the claim

$$e \in S \Rightarrow (S \setminus \{e\} \subset S)$$

which says that if an element belongs to a set, taking it away produces a smaller set. If the element type is T, and the scope bounds it to have a size of at most 2, Nitpick will generate values for e such as $t0$ and $t1$, and values for S such as $\{\}$. $\{t0\}$, $\{t1\}$, $\{t0, t1\}$. There will be 8 cases corresponding to the 4 values of S and 2 values of e. But Nitpick will only examine half of these cases, because the cases in which e is $t1$ are just relabellings of the cases in which e is $t2$. All that matters is the relative value of e and the elements of S, which

means there are only four cases: *S* is empty, *S* contains one element matching *e*, *S* contains one element not matching *e*, and *S* contains two elements.

Like short-circuiting, isomorph elimination happens at run-time, but most of the work is done at compile-time. When you select a claim or schema, Nitpick applies a type inference algorithm to determine the most general type of each variable, which is then used to decide which relative labellings matter. This inference accounts for some of the delay. To see the results of the inference, select *Wiring Types* from the *Display* menu. At run-time, isomorph elimination uses an isomorph-free repertoire of values for each variable: that is, no value in the set can be obtained from another one by relabelling it. This repertoire is generated at compile-time when the scope is selected. That is why you'll notice some delay when a large scope is selected.

2.2.6 Saving Results

There are three ways to save the output of the transcript window. You can select a portion of the text in the transcript window and copy it to the clipboard. You can switch logging on and off by selecting *Logging* under the *File* menu; when you switch it on for the first time, you will be prompted to give the name of the file into which the log will be placed. Also, when you quit Nitpick, you are given the choice of saving the entire transcript to a file.

2.3 Troubleshooting

When I type command-R to run a claim, nothing happens. Also, the Run command is greyed out in the File menu.

Make sure you have selected a claim or schema first.

When I open my specification, the schema and claim windows do not appear.

Check that there are no parsing errors reported in the transcript window, and that your specification contains schemas and claims.

When I run my claim, the progress bar barely moves.

Nitpick cannot handle a state space this size. Try to reduce the scope or simplify the specification. If you use Edit Claim/Schema scope, the dialog will show you as you set the scope how big the space is. A space of 1 million values should be no problem. Up to about 1 billion is usually fine too, but sometimes the reductions to not work as well.

Why does the progress bar barely move and then jump suddenly?

Because of short-circuiting. The jump occurs when a large subtree is pruned.

The progress bar zips to the end so fast, and reports that so few cases have been examined that I am not confident my claim was properly checked.

Check that your schemas have examples by running them; this often happens because a transition schema is over-constrained and short-circuiting can eliminate almost all cases.

I expected a variable to be identified as derived, but it was enumerated.

The derived variable mechanism is quite primitive. It will not treat a variable as derived, for example, if it is defined by two equations, each of which is the consequent of an implication. To avoid this problem, simplify the specification by separating the operation into cases, and making claims about them separately.

2.4 Known Problems

The current release of Nitpick has a couple of known bugs that we expect to fix soon:

· When all the variables in a formula can be derived (usually because variables are equated with constants), no enumeration occurs. Turn off derived variables to get around this.

· There is a memory leak, so it is a good idea to exit and restart Nitpick now and then.

· Occasional crashes occur due to (we guess) to memory corruption bugs.

All bug reports are welcome, especially if you can reproduce the scenario leading to the error. Our email address is nitpick@cs.cmu.edu.

2.8

Chapter 3
Command Reference

3.1 Menus

Almost every action that can be accomplished using Nitpick is accessible through menu commands. This is a complete listing of all of the menu commands and their effects.

3.1.1 File Menu

3.1.1.1 Open Spec

Opens a text file containing an NP specification and its associated claims. The file must contain a complete specification. The command will bring up a standard dialog box for opening a file. If a subsequent specification is opened, the environment for the previous specification will be lost, although the scope settings will be retained.

3.1.1.2 Log Session

Toggles logging of all output to the log file. The first time logging is enabled, the user will be prompted for a file name. When logging is enabled, transcript output is written to the log file. Some output, such as the large debugging traces are written *only* to the log file and do not appear in the transcript window. Logging is initially off.

3.1.1.3 Run All

Evaluates all claims using the current options and scopes. The progress window must be clicked after each claim has been completed before the next claim will be evaluated.

3.1.1.4 Run

Evaluates the currently selected claim or schema, yielding counter examples if a claim is currently selected and sample executions if a schema is currently

selected. The currently selected claim or schema is the one highlighted in the claims or schemas windows.

3.1.1.5 Halt

Halts the current evaluation. If the evaluation was started using the *Run All* command, all further evaluations entailed by that command will be cancelled.

3.1.1.6 Quit

Exits the Nitpick tool. The user will be prompted with an option to save the contents of the transcript window before exiting.

3.1.2 Edit Menu

This menu is currently disabled.

3.1.3 Scope Menu

This menu contains commands to bound the execution of schemas and claims, by selecting a scope that associates a maximum size with each given set, and possible also a constraint for each relation variable on the kinds of values it may take. None of these commands are enabled until a specification has been loaded.

3.1.3.5 Load Scope

Loads a scope previously saved in a file. The user will be prompted for the name of the file to be loaded.

3.1.3.5 Edit File Scope …

Opens a dialog box for editing the global scope associated with the entire specification/file. This lets the user set the sizes for all of the given types in the specification.

3.1.3.5 Edit Claim Scope …

Opens a dialog box for editing the scope associated with the currently selected claim or schema. This lets the user set the sizes for all of the given types used in that claim or schema as well as restrict the class of relations considered in enumerating possible values for the claim or schema's variables. A subsequent change to the file scope will not reset any bound that has been edited in a specific claim or schema scope.

3.1.3.5 Save Scope …

Saves the current scope to a file for possible future loading. The settings

saved include not only the global file scope, but also the scopes associated with individual schemas and claims. The user will be prompted for a name of a file to save the scope in. This defaults to *filename*.Σ, where *filename* is the base name of the currently loaded specification file.

3.1.4 Options Menu

This menu contains commands that control how the execution of schemas and claims is conducted, by toggling reduction mechanisms and by placing a limit on the number of instances to be found before execution terminates.

3.1.4.1 Isomorph Elimination

Toggles the use of the isomorph elimination in reducing the total search space required to execute a claim or schema. Isomorph elimination is initially enabled.

3.1.4.2 Short Circuiting

Toggles the use of short circuiting in reducing the total search space required to execute a claim or schema. Short circuiting is initially enabled.

3.1.4.3 Derived Variables

Toggles the use of derived variable analysis in reducing the total search space required to execute a claim or schema. Derived variable detection is initially enabled.

3.1.4.4 Limit Run …

Opens a dialog box that enables the user to control the maximum number of instances to be found as well as the number to be displayed. If the number to be found is set to *i*, the checker will evaluate cases until either the space determined by the scope is exhausted, or *i* instances have been found. If *find all* is selected, the evaluation will terminate only when the space is exhausted (or the *halt* command is issued). The display setting determines how many of the instances found are to be displayed; if the number *i* is chosen, the first *i* of the instances found are displayed in the transcript window.

3.1.4.5 Debug Nitpick

This selection allows adds additional menu commands which allow debugging the Nitpick tool itself. These are expected to be used only in conjunction with the Nitpick group when trying to identify a problem in Nitpick.

3.1.5 Window Menu

This menu displays the name of each open window. Selecting an entry will bring the window of that name to the front, opening the window first if it has been closed. The *Claims* and *Schemas* entries are greyed out when a specification has not been loaded. If the specification has no claims (or no schemas), the *Claims* (or *Schemas*) entry will remain greyed out after the specification is loaded.

3.1.6 Display Menu

These commands display information about Nitpick's evaluation strategy. All of these commands apply to the currently selected claim or schema, and are disabled when no claim or schema is selected.

3.1.6.1 Enumeration Vars

Displays a list of the enumeration variables for the currently selected claim or schema. If the short circuiting reduction is currently enabled, the variables will be listed in their enumeration order. Following each variable, in square brackets, is the number of values to be enumerated for this variable. The size of the entire space is the product of these numbers.

3.1.6.2 Derived Vars

Displays a list of the derived variables for the currently selected claim or schema. This command is only enabled when derived variable analysis is enabled. The derivation is given along with each variable.

3.1.6.3 Wiring Types

The isomorph elimination reduction exploits the structure of the claim or schema being evaluated. If two variables can be shown to have different types, then their values are not compared, and relative permutations of one with respect to the other can be ignored. Nitpick therefore applies type inference to the claim or schema to determine the most general type of each variable. This command displays the results. For each type inferred, a list of variables of that type is given. Since a relation *r* has two types, one for the left side (the domain elements) and one for the right side (the range elements), it will appear twice in the display, as *LHS r* and *RHS r*. The more types are inferred, the better the reduction will be.

3.1.6.4 Executable

The claim or schema is compiled into an 'executable' that is evaluated for every combination of variable values. This command displays the executable.

3.2 Dialog Windows

Many actions either require interaction with one of the Nitpick dialogs, or are simply easier to accomplish using the dialog. There are four basic Nitpick dialogs: Scope dialog, Limit Run dialog, Schemas and Claims windows and an individual claim or schema dialog.

3.2.1 Scope Dialog

Logically, there is a scope dialog for the entire specification as well as one for each claim and schema in the current specification. As many of these dialogs can be open at once as desired (or none), but they all require a specification to be loaded before they can be opened. A scope dialog is modeless, so any other action can be taken while a scope dialog window is open. Any action performed in the scope dialog is effective immediately, although the cancel button can revert all such changes made since the particular scope dialog was opened. Like any dialog, the tab key will move between various portions of the dialog.

The file wide scope window can be accessed through the Edit File Scope menu item in the Scope menu. A specific claim or schema scope dialog can be accessed by selecting the claim or schema in the Claims or Schemas window and using the Edit Claim Scope menu item in the Scope menu. Or it may be brought up by clicking on the Edit Scope button in a claim or schema dialog.

3.2.1.1 Types List

The Types list shows all given types found in the specification for the file level scope dialog, or just those used in a given claim or schema for a claim/schema scope dialog. The numbered radio buttons will change the number of elements of the currently selected entry in the list. The current number of elements is shown in square brackets following the name.

If the Types list is currently active (indicated by a bold outline around the list), the number keys 1-8 will also set the number of elements for the currently selected entry. The up and down cursor keys will change the selection appropriately. In addition, the return key will move the selection down to the next entry if any.

3.2.1.2 Relations List

The Relations list shows all variables which are typed to be relations and which will be enumerated when evaluating the claim or schema represented by the claim or schema scope dialog. This list is only enabled for

claim/schema specific scopes. This list allows further restricting the values which will actually be enumerated when evaluating the claim or schema.

The characters inside the square brackets following the variable name indicate what will be enumerated given the current settings. The presence of an F indicates that only functions will be enumerated, an I indicates injections, an O surjections (onto) and a T indicates that only total relations will be enumerated.

These settings can be changed by clicking on any of the four corresponding check boxes next to the Relations list. Note that a restriction from the specification itself, such as functinal, cannot be unset through the scope dialog.

If the Relations list is the current active element (again indicated by a bold boundary around the list), typing the letters F,I,T and O will toggle the corresponding values. The return and up and down cursor keys will also change the selection as in the Types list.

3.2.1.3 Cancel Button

The Cancel button will undo any changes made to the scope since opening this window. It will also dismiss the window.

3.2.1.4 Done Button

The Done button dismisses the window.

3.2.2 Limit Run Dialog

This dialog, accessed through the Limit Run menu item in the Options menu, is used to control how many instances will be found found and displayed. It is a modal dialog, meaning that either cancel or done must be selected before any action elsewhere can be taken. The dialog can be accessed during a run, in which case its limits will be applied to that run.

3.2.2.1 Find At Most

There is both a check box (labelled All) and an editable text item that combine to make up the Find At Most control. If the check box is checked, no text will appear in the text box. This means that evaluations will not stop until the entire case space has been searched for interesting instances. If the check box is not checked, the evaluation will halt once the indicated number of instances have been found.

3.2.2.2 Display at Most

Like the Find At Most control, the Display at Most control is a combination of

a check box (labelled All) and an editable text item. If the check box is checked, all instances that are found will be displayed. Otherwise, only the first n instances will be displayed where n is the value in the text field.

3.2.2.3 Cancel Button

The Cancel button will dismiss the window without making any changes to the current limits.

3.2.2.4 Done Button

The Done button will dismiss the window after making its values the current limits. If neither control has the All box checked, the number given in the Find at Most text field must be greater than or equal to the number given in the Display at Most field.

3.2.3 Claims and Schemas Windows

These windows, which appear automatically upon loading a specification file, list all of the claims and schemas defined in the specification. If there are no claims, the Claims window will not appear. If either window is closed (using the go away box), it may be re-opened using the Claims or Schemas menu item in the Window menu.

Clicking on an item in either window selects that item, making it the currently selected claim or schema, and enabling many of the commands. Only one claim or schema can be selected at a time. Double clicking on any of the items will bring up the claim or schema dialog corresponding to the claim or schema name which was double clicked.

3.2.4 Claim or Schema Dialog

This dialog can be used to control all evaluation of a single claim or schema as well as to view the results of that evaluation. The dialog can be accessed by double clicking on the name of a claim or schema in the Claims or Schemas window.

The resulting dialog contains five basic sections: formulas, reductions, show buttons, action buttons and prior run statistics. The formulas section is at the top and simply lists up to the first three formulas that define the claim or schema being processed.

3.2.4.1 Reduction Check Boxes

These check boxes display the state that was used for the last evaluation of this claim or schema as well as providing a convenient place to change the

settings for the next run.

3.2.4.2 Show Buttons

These four buttons can be used to view various structures useful in understanding how Nitpick is processing the claim or schema. They are found in a vertical line in the middle of the window. Clicking on one of the four Show buttons is equivalent to selecting one of the four menu items found in the Display menu, after selecting the appropriate claim or schema if necessary. The generated output is written to the transcript window, or the log file if logging is currently enabled.

3.2.4.3 Action Buttons

The Action buttons are two buttons found at the right hand side of the dialog. They allow the user to edit the claim or schema specific scope (the Edit Scope button) as well as an easy way to evaluate a given claim or schema (the Run Claim button). If the Claim or Schema represented by this dialog is the currently selected one, the Edit Scope button is equivalent to the Edit Claim Scope menu item and the Run Claim button is equivalent to the Run menu item.

3.2.4.4 Run Statistics

The bottom of the dialog displays a number of statistics from the current or previous run. This information includes the total number of cases in the state, the number of cases actually checked, the number of interesting instances found and the total time spent on the search. The text also indicates whether the current settings have been run and whether that run was full (all states checked) or partial (halted either through the Halt menu item or because the search limit was reached).

Chapter 4
NP Language Description

4.1 Overview

NP is a small specification language for describing state machines.

The state is described by a collection of variables, each of which is either a scalar, or a set, or a binary relation, and a logical formula constraining the values of the state variables. Other data structures, such as mappings, trees and lists, are treated as special kinds of relations.

The transitions of the machine are described by operations. Each operation corresponds to a class of transitions, and is specified as a formula relating the state after to the state before. Different variables are used to refer to the before and after values of the state, so variables are like mathematical variables and not program variables: there is no notion of a variable's value changing when a transition occurs. Nevertheless, we sometimes talk loosely about a variable's value 'changing', meaning simply that the values of the before- and after-variable differ. The formula can be written in an explicit style – giving an equality for each after-variable defining its value in terms of before-variables – or in an implicit style – constraining the before and after-variables to be related by some property – or in a mixture of the two.

The basic unit of specification is the *schema*. It consists of some variable declarations and a formula. Schemas are used for both state and operation descriptions; the only difference is that operation descriptions have two variables for each state component, one for the before and one for the after-value, and may also have arguments. Descriptions can be built up incrementally, by using the names of one schema in another to import its variable declarations and formula. This makes it easy to factor out a common property of several operations; you define a schema for the property and import it into each operation.

A claim looks much like a schema. It consists of some variable declarations and a formula that is claimed to be true for all values of the variables. Many

different kinds of properties can be asserted in a claim; the two most common are that a state description implies some invariant, and that an operation preserves an invariant.

NP comes in two lexical versions: an ascii version, used for interacting with the Nitpick tool and communicating by email, and a more typographically appealing version for displaying specifications. The two differ only in the symbols used for constructing relation and set expressions.

4.2 Comparison to Z

If you know Z, you should have no trouble learning NP. Bear in mind that Nitpick is not intended to be a general mathematical notation, but is designed for describing and checking state machine descriptions. Its tractability has a major cost in expressive power.

The principal respects in which Nitpick differs from Z are:

- NP has no built-in concrete types; in particular, there are no integers. This means that NP is pretty useless if you want to write specifications of numerical applications. But for most applications, integers are not necessary. The most common use of integers in Z occurs with sequences, which are modelled as functions from integers to elements. NP models a sequence as an order instead, with a function that maps each element to its successor.
- NP has no quantifiers. Having no scalar quantifiers means that you sometimes have to be clever in constructing relational expressions. The section on idioms below shows a number of examples. Having no relational quantifiers means that you cannot write claims about preconditions. We hope to remedy this soon.
- NP's notion of schema is slightly different. An NP schema should be thought of as a named collection of variable declarations along with a formula. In NP, you can't use a schema as a type.

  There's no need for the Δ convention in NP because primed versions of variables declared in a transition schema are present implicitly. NP's keyword *const* is like Ξ in Z, but it can be applied to individual variables (and schemas) even in a schema that does not describe an operation. When a variable x is declared to be constant in a state schema, the equation $x' = x$ is included implicitly in any transition schema that imports the state schema; this mechanism takes the place of Z's global

variable declarations. Primed variables are never declared explicitly in Nitpick.

It's easy to distinguish state and transition schemas, because every transition schema has a (possibly empty) argument list. Arguments are very useful, because you can bind a formal to an expression, which is not possible with Z's renaming.

NP has no implicit global schema, no axiom schemas, and no generics.

The signature matching rules for Nitpick are more stringent than Z's. When two declarations of a variable are merged, they must match exactly. As in Z, the function $f\colon S \to T$ and the relation $f\colon S \leftrightarrow T$ have the same type, but they are not compatible declarations.

- NP's scoping rules are a bit stricter than Z's. We don't like the way variable declarations pop-up to the top in Z. If you write a schema or claim body like $[S1 \mid x = y \land S2]$, the variables x and y have to be declared in $S1$; declaring them in $S2$ alone is no good.

- Claims in NP are just formulae. That's why they are written in a form so similar to schemas. In Z, a property of a specification is written in a meta-language with turnstiles for talking about inference. NP is much simpler: a claim just says that a formula is valid, and its counterexamples are analogous to the examples of a state or transition schema.

  Constraints that are implicit in declarations are treated differently for schemas and claims. Suppose we have a schema S with variable declarations SD and a formula SF. The schema $X = [S \mid P]$ is equivalent to $[SD \mid SF \land P]$, but the claim $C :: [S \mid P]$, whose body looks exactly the same, is equivalent to $[SD \mid SF \Rightarrow P]$. If the claim were interpreted like the schema, any counterexample of S would also be a counterexample of the claim, which is not at all what you would expect.

- Special declaration forms are more significant in NP. Like Z, NP lets you define various special kinds of relation; instead of the funny arrow symbols, however, it uses modifier keywords. For example,

  $f\colon fun\ S \leftrightarrow T$

  declares f to be function from S to T; this can be written equivalently as

  $f\colon S \to T$

  Note that the plain arrow denotes a *partial* function in NP. To declare a total function, we would write

  $f\colon tot\ S \to T$

An injection in NP is just a relation whose transpose is a function (whereas in Z an injection is a function in both directions). So

f: *inj tot* $S \leftrightarrow T$

does not make f one-to-one, but

f: *inj tot* $S \rightarrow T$

does.

Modifier keywords can be used in formulae; *fun*(r), for example, is true when the relation r is a function. As in Z, it is good specification style to constrain a variable in its declaration whenever possible. More importantly, Nitpick can exploit constraints in declarations to make searching much faster; if you declare as a variable to be a function, only function values will be enumerated.

NP has two special declaration forms that do not appear in Z, for partitions and sequence-like orders. These are explained below.

- Image and application. Application of the function F to the argument x is written $F.x$ in NP. If x is not in the domain of F, the elementary formula containing the term $F.x$ is false. The formula $F.x = y$ is thus equivalent to $x \in \mathit{dom}\ F \wedge F.x = y$. In Z, the formula would have undetermined value when x is outside the domain of F. In practice, the difference does not matter much because specifications should be written so that their meaning is independent of how these bad applications are treated. The negated comparison operators $\neq$ and $\notin$ are not elementary, so $x \neq y$ and $\neg\ x = y$ are equivalent, and so are so $x \notin y$ and $\neg\ x \in y$.

  As in Z, you can apply a relation as a function too; $R.x$ is the application of the relation R, as a function, to x. If R does not map x to exactly one value, the application is undefined, and the containing elementary formula is false.

  Relational images are so common that is convenient to use the same notation: $R.S$ is the image of the set S under the relation R. Nitpick distinguishes image and application by examining the type of the second variable.

- NP is lexically very close to the horizontal form of Z. We have replaced some of the symbols to make them match their ascii counterparts, thus $=$ instead of $\widehat{=}$. We have picked ascii representations for the relational operators and logical connectives.

4.3.1 Specifications and Files

The Nitpick tool requires a one-to-one association between specifications and files. You cannot write two specifications in the same file, nor split a specification amongst several files.

4.3.2 Ascii Representation

The specification loaded by Nitpick must be in standard ascii format. There are about 20 special symbols in the language that look pretty ugly in ascii, so in the description here, typographic versions are used instead. The table in section 4.9.9 gives the ascii character sequence corresponding to each special symbol.

4.3.3 Lexical Considerations

A specification consists of a sequence of tokens and separators. A *token* is a sequence of printing ascii characters, representing a reserved word, a punctuation symbol, an operator or an identifier. Upper and lower case are distinguished.

A *separator* is a space, a tab, a carriage return or a newline. In general, tokens may be separated by any number of separators, including zero, except for adjacent identifiers, which require at least one separator.

Identifiers name types, variables, schemas and claims. Reserved words may not be used as identifiers. An identifier may contain upper and lower case alphabetic and numeric characters, underscores, ? and !, must start with an alphabetic character or an underscore, and is limited to 128 characters. Type, variable, schema and claim names must form distinct sets.

Comments start with /* and end with */ and may occur anywhere on the line. They may not be nested.

4.3.4 Scoping Units

The names of the given types have a scope that extends to the end of the file. The names of schemas extend from immediately after their definition to the end. Claims are named, but only so they can be selected during interactive analysis; their names have no scope.

4.3.5 General Structure

specification

type_decls schemas_and_claims

schemas_and_claims
schema_declaration
claim_declaration
schemas_and_claims schema_declaration
schemas_and_claims claim_declaration

A specification consists of a declaration of one or more given types, followed by a sequence of schemas and claims.

4.3.6 Type Declarations

type_decls
[*given_type_list*]

given_type_list
given_type
given_type_list , *given_type*

given_type
identifier

The given types are declared as a comma-separated sequence of identifiers within square brackets.

4.3.7 Basic Type Expressions

type_expr
given_type
set_type
rel_type
sequence_type

set_type
set given_type
one *given_type*

rel_type
given_type ↔ *given_type*
given_type → *given_type*
fun *rel_type*
inj *rel_type*
tot *rel_type*
suj *rel_type*

A type expression represents a scalar, set or binary relation, constructed from

given types. Higher-order types are not allowed, and names cannot be given to type expressions.

A set or relation type expression may be preceded by one or more *modifiers*. Modifiers constrain the values of a type just like explicit constraints, but allow Nitpick to run more efficiently. The order of modifiers is immaterial.

The set modifier *one* constrains the type to include only sets containing one or zero elements.

The relation modifiers have the following meanings:

- *fun* the relation is a *function*
- *inj* the relation is *injective*: its transpose is a function
- *tot* the relation is *total*; it maps every element of the domain type
- *suj* the relation is *surjective*; it maps to every element of the range type

The expression $S \rightarrow T$ is short for *fun* $S \leftrightarrow T$.

4.3.8 Sequence Type Expressions

sequence_type
    seq *named_type*
    tot seq *named_type*

The expression *seq S* describes the type whose values correspond to sequences without repeated elements. Each such value is represented by a function mapping each element to its successor in the sequence, with the last element mapped to itself.

A sequence type expression may be prefixed with the modifier *tot*; a value of the sequence type then describes an order in which every value of the element type has a successor.

4.3.9 Variable Declarations

schema_name (*arg_list*)

var_decl
    variable_list : *type_expr*
    variable_list : *partition_type*

variable_list
    identifier
    variable_list , *identifier*

partition_type
    identifier part *given_type*

A variable declaration consists of a list of variable names and a type expression; the variables in the list all have the declared type. Although primed variables may appear in formulae, they are never explicitly declared.

A *qualified declaration* is preceded by the keyword *const*. Intuitively, the variable is a constant whose value cannot change.

A *partition declaration* has the syntactic appearance of a standard variable declaration, but adds an implicit constraint that relates the values of the variables. In contrast, the modifiers of a standard declaration constrain each of the declared variables independently. The declaration

*s*1, *s*2, ..., *si*: *f part X*

declares *s*1, *s*2, ..., *si* to be sets of elements of type X, which together form a partition of *X*. They are disjoint and exhaustive; every value of type *X* belongs to exactly one of the sets. The declaration also introduces a variable *f* corresponding to the characteristic function of the partition, with a type as if declared by

f: $X \rightarrow Label$

where *Label* is some (inaccessible) type that indexes the sets *s*1, *s*2, ..., *si*.

4.3.10 Expressions

expr
- *scalar_expr*
- *set_expr*
- *relation_expr*

All expression operators are left associative, except for the unary operators $^{\sim}$, $^{+}$ and *. The precedence order, from highest to lowest, is:

- unary operators ($^{\sim}$, $^{+}$, *)
- application and image
- unary operators (dom, fdom, ran, first, last, next)
- restriction operators ($\lhd \rhd \lhd\!\!\!\rhd$)
- relational composition (;)
- intersection ($\cap$)
- relational override ($\oplus$)
- difference ($\setminus$)
- union ($\cup$)
- comparison operators ($\in$, $=$, $\subseteq$, $\subset$)

4.3.10.1 Scalar Expressions

scalar_expr
- *variable*
- *relation_expr* . *scalar_expr*
- first *relation_expr*
- last *relation_expr*
- (*scalar_expr*)

A scalar expression is either a variable, or an application; there are no scalar operators. An application *R.x* denotes the result of applying the relation-valued expression to the scalar-valued expression *x*. If the relation denoted by *R* maps the element denoted by *x* to exactly one element, its value is the value of the expression; otherwise the expression is undefined. Section 4.6 explains this in more detail.

The expressions *first s* and *last s* are applications of the built-in sequence operators to the relation s, which must have been declared to be a sequence. They define the first and last elements of the sequence respectively.

4.3.10.2 Set Expressions

set_expr
- { }
- { *scalar_exprs* }
- *variable*
- *given_type*
- *relation_expr* . *set_expr*
- *set_expr* ∪ *set_expr*
- *set_expr* ∩ *set_expr*
- *set_expr* ∖ *set_expr*
- dom *relation_expr*
- ran *relation_expr*
- (*set_expr*)

A set expression is either a variable, a combination of two set expressions using the standard set operators (union, difference and intersection), a relational image, a projection, a display, or the empty set constant {}. The expression *R.S* denotes the relational image of the set *S* under the relation *R*; its value is the set containing the elements in the range of *R* that are mapped to by the elements of *S*.

Relational image and application expressions are distinguished by the types of their constituent expressions. Thus, if *R* has the type $S \leftrightarrow T$, the expression *R.X* is an application when *X* has type *S* and an image when *X* has type *set S*.

The expressions *dom R*, *ran R* and *fdom R* project the relation *R* into sets denoting the domain of *R* (all values that are mapped by *R*), the range of *R* (all values mapped to) and the functional domain of *R* (all values mapped to exactly one value).

A display expression of the form {*e*1, *e*2, ..., *ek*}, where *e*1..*ek* are scalar-valued expressions, denotes the set whose elements are the values of those expressions.

4.3.10.3 Relation Expressions

relation_expr
- { }
- Id
- Un
- *variable*
- *relation_expr* ∪ *relation_expr*
- *relation_expr* ∩ *relation_expr*
- *relation_expr* ∖ *relation_expr*
- *relation_expr* ⊕ *relation_expr*
- *relation_expr* ; *relation_expr*
- *set_expr* ◁ *relation_expr*
- *set_expr* ⩤ *relation_expr*
- *relation_expr* ▷ *set_expr*
- *relation_expr* ⩥ *set_expr*
- *relation_expr* ~
- *relation_expr* +
- *relation_expr* *
- next *relation_expr*
- (*relation_expr*)
- { *scalar_maplets* }

scalar_maplets
- *scalar_expr* ↦ *scalar_expr*
- *scalar_maplets*, *scalar_expr* ↦ *scalar_expr*

A relation expression may be:

- constructed with set operators. The expressions $P \cup Q$, $P \cap Q$, and $P \setminus Q$ denote the union, intersection and difference of the relation expressions *P* and *Q* viewed as sets of pairs.

- constructed with relational operators. *P* ; *Q* is the relational composition of *P* and *Q*. *P* * is the reflexive and transitive closure of *P*. P^+ is the transitive closure of *P*. $P^\sim$ is the transpose of *P*.

- constructed with restriction operators. $S \lhd R$ denotes the domain restriction of the relation R to the set S; its value is the subrelation of R consisting of pairs whose domain elements are in S. $S \lhd\!\!\!\!- R$ is the negative domain restriction; its value is the subrelation of R whose pairs have domain elements not in S. $R \rhd S$ is the range restriction of R to S; its value is the subrelation of R whose pairs have range elements in S. $R \rhd\!\!\!\!- S$ is the negative range restriction of R to S; its value is the subrelation of R whose pairs have range elements not in S.

- derived from a sequence. The expression *next s* is the relation obtained from the sequence s that denotes the successor function of the sequence; it is equivalent to $s \setminus Id$.

- a constant. The constant {} denotes the empty relation as well as the empty set. There are two additional relation constants: *Id*, the identity relation, and *Un*, the universal relation. Each constant should be viewed as a family of constants indexed by type. The types of these constants are inferred from the context. In a context in which *Id* has the type $S \leftrightarrow S$, its value is the relation mapping every value of the type S to itself. Likewise, if *Un* has the type $S \leftrightarrow T$, its value is the relation mapping every value of the type S to every value of the type T.

- a display. The expression $\{e1 \mapsto f1, e2 \mapsto f2, ..., ek \mapsto fk\}$, where $e1..ek$ and $f1..fk$ are scalar valued expressions, denotes the relation mapping $e1$ to $f1$, $e2$ to $f2$, and so on as far as ek to fk. Note that the maplet $ei \mapsto fi$ is not a legal expression; there are no tuple-valued expressions.

4.3.11 Formulae

A formula has the value true or false for a binding of its variables to values. It is never the case that a formula has neither or both truth values for some binding.

4.3.11.1 Elementary Formulae

elem_formula
- *expr* $\in$ *expr*
- *expr* $\notin$ *expr*
- *expr* $=$ *expr*
- *expr* $\neq$ *expr*
- *expr* $\subseteq$ *expr*
- *expr* $\subset$ *expr*
- *relational_predicate*
- *set_predicate*

relational_predicate
fun *expr*
inj *expr*
tot *expr*
suj *expr*
seq *expr*

set_predicate
one *expr*

Modifiers may be used in formulae as well as in declarations. For a set-valued expression *S*, the formula *one S* is true when *S* has one or fewer elements. For a relation-valued expression *R*, the formula *fun R* is true when *R* denotes a function, *tot R* is true when *R* is total, and so on. Note that *part* is not a modifier and may not be used in this way.

The formula $A = B$ is true when *A* and *B* have the same value; they may be scalar, set or relation expressions.

For set-valued expression *S* and scalar-valued expression *e*, the formula $e \in S$ is true when *e* is an element of *S*.

The formula $A \subseteq B$ ($A \subset B$) is true when *A* is a subset (proper subset) of *B*. *A* and *B* may be set-valued or relation-valued, in which case the relations are interpreted as sets of pairs.

If a formula contains a function application that is undefined for some binding of variables to values, the elementary formula in which the application appears has the value *false*.

4.3.11.2 Compound Formulae

formula
(*formula*)
$\neg$ *formula*
formula $\wedge$ *formula*
formula $\vee$ *formula*
formula $\Rightarrow$ *formula*
formula $\Leftrightarrow$ *formula*
invocation
invocation ; *invocation*
elem_formula

invocation
schema_ref (*arg_list*)
schema_ref ( )

schema_ref

arg_list
  expr
  arg_list , *expr*

schema_ref
  schema_name
  schema_ref '

Compound formulae are built with the standard logical connectives from elementary formulae. Negated comparisons are always shorthands and are thus regarded as compound; $e \notin S$, for example, is short for $\neg\, e \in S$.

A schema name may also play the role of a formula, being short for the formula of the referenced schema.

The precedences for the logical operators are, from highest to lowest:

- negation ($\neg$)
- conjunction ($\wedge$)
- disjunction ($\vee$)
- implications ($\Rightarrow$, $\Leftrightarrow$)

4.3.12 Schemas

schema_declaration
  state_schema_decl
  trans_schema_decl

Schemas are used to structure specifications. A schema is essentially a named formula that can be incorporated, along with its declarations, in another schema. There are two kinds of schemas. State schemas are used to introduce state variables and to specify invariants on states. Transition schemas are used for specifying operations or partial properties of operations (such as frame conditions).

A schema may include a reference to another schema. The meaning of such a reference is described by rules for elaborating the referencing schema with the declarations and formula of the referenced schema.

Each schema should be elaborated in full...

4.3.12.1 State Schema Declarations

state_schema_decl
  schema_name = *schema_body*

declaration_part
declaration
declaration_part declaration
declaration_part ; declaration

declaration
schema_name
const *schema_name*
var_decl
const var_decl

A state schema declaration consists of a name and a schema body. In its simplest form, a schema body consists of a sequence of variable declarations and a formula. Every variable that appears explicitly in the formula must be declared.

The formula may be written syntactically as a list of formulae, but the list is treated semantically as a single formula that conjoins the formulae of the list. The formula may be missing, in which case the pathological formula *true* is implicit.

Any of the variable declarations may be replaced by the name of a state schema declared earlier in the specification. In this case, the variable declarations of the referenced schema are added implicitly to this schema's declarations, and the formula of the referenced schema is implicitly conjoined to this schema's formula.

No variable may be declared twice; if a declaration already exists for a variable, an implicit declaration for the same variable has no effect. When a variable is declared more than once in this way, the type expressions must match, but the modifiers and presence of a *const* qualifier may differ; the modifiers and possible qualifier are accumulated, so that the effective declaration includes all the modifiers of the explicit and implicit declarations, and is qualified with *const* if any explicit or implicit declaration is.

A state schema may also be referenced by name in the formula of the schema. In this case, the formula of this schema stands for a longer formula in which the referenced schema name is replaced by that schema's formula. The declarations of the referenced schema are not added implicitly to this schema's declarations.

If no variables appear explicitly in the schema's formula, no declarations are necessary, and the square brackets are omitted.

trans_schema_decl
schema_name (*param_list*) = *schema_body*
schema_name ( ) = *schema_body*
schema_name = *schema_body*

schema_body
[*declaration_part* | *formula_part*]
[*declaration_part*]
formula

declaration_part
declaration
declaration_part declaration
declaration_part ; *declaration*

declaration
schema_import
const *schema_import*
var_decl
const var_decl

schema_import
schema_name
schema_name ( )
schema_name (*arg_list*)

A transition schema declaration consists of a name, an argument list, and a schema body. The argument list consists of a sequence of variable declarations, none of which may be qualified by *const*. The list may be empty.

The rules for constructing and interpreting the schema body are the same as those for a state schema, with the following exceptions. The variables declared in the argument list, called the *formals* of the schema, must not be redeclared in the body, implicitly or explicitly.

Every variable declaration in the body is accompanied by an implicit declaration that is identical, except that for a variable x it declares a variable x'. If the declaration of x is qualified with *const*, the equation $x' = x$ is added implicitly to the formula.

References to transition schemas are treated much like references to state schemas within state schemas: a reference in the declarations of the schema imports both the referenced schemas declarations and its formula; a reference in the formula imports only the formula. A transition schema with the

name *T* is referenced as *T*() if it has no formals. Otherwise, the name is accompanied by an argument list with an actual for each formal. If the reference occurs in the declaration part of the schema, each actual must have been declared as a formal of this schema. If the reference occurs in the formula, each actual must either be a formal or a variable declared in this schema. In either case, the formula that is implicitly included is obtained by taking the formula of the referenced schema and replacing the formals with the actuals, and it is illegal for the name of an actual to coincide with the name of declared variable in the referenced schema.

References to state schemas are treated as follows. A state schema with the name *S* may be referenced as *S* or *S′*. If the reference *S* appears in the formula, the reference is implicitly replaced by the referenced schema's formula. The reference *S′* is replaced by a version of the referenced schema's formula in which each variable *x* in the formula of *S* is renamed *x′*. Only an unprimed reference is permitted in the declaration part; both primed and unprimed versions of the declarations and formula are imported. If the reference is qualified by const, an equation $x' = x$ is imported for each variable of the referenced schema.

If only a single formula is given in a schema body and no variables appear explictly in that formula, no declarations are necessary and the square brackets and vertical bar may be omitted.

4.3.13 Claim Declarations

claim_declaration
    state_claim_decl
    trans_claim_decl

state_claim_decl
    claim_name :: *schema_body*

trans_claim_decl
    claim_name (*param_list*) :: *schema_body*
    claim_name ( ) :: *schema_body*
    claim_name :: *schema_body*

A claim declaration has the same syntactic form as a schema declaration, but with a double colon in place of the equals sign. Like schemas, claims come in two varieties: state claims and transition claims.

There is one difference of interpretation between claims and schemas. Whenever a schema's formula is imported by reference to the schema's name in the declaration part of the claim, the formula is not implicitly conjoined to

the claim's formula, but instead becomes the hypothesis of an implication whose consequent is the explicit formula.

Suppose we have a schema *S* with variable declarations *SD* and a formula *SF*. The schema $X = [S \mid P]$ is equivalent to $[SD \mid SF \wedge P]$, but the claim $C :: [S \mid P]$, whose body looks exactly the same, is equivalent to $[SD \mid SF \Rightarrow P]$.

Claims may not be referenced in schemas or in other claims.

4.4 Grammar

A complete grammar for NP is given below. A production of the form

X
    *Y*1
    ...
    Yk

defines each instance of the non-terminal *X* to be an instance of some *Yi*; it is read '*X* is either a *Y*1, or a *Y*2, or ...'. Non-terminals appear in italics; reserved words are in roman. All punctuation symbols are terminals of the language.

The grammar uses the typographic versions of the special symbols. For their ascii counterparts that the Nitpick tool requires, see the table below.

specification
    type_decls schemas_and_claims

type_decls
    [*given_type_list*]

given_type_list
    given_type
    given_type_list , *given_type*

given_type
    identifier

schemas_and_claims
    schema_declaration
    claim_declaration
    schemas_and_claims schema_declaration
    schemas_and_claims claim_declaration

4.18

schema_declaration
    state_schema_decl
    trans_schema_decl

trans_schema_decl
    schema_name (*param_list*) = *schema_body*
    schema_name ( ) = *schema_body*
    schema_name = *schema_body*

state_schema_decl
    schema_name = *schema_body*

claim_declaration
    state_claim_decl
    trans_claim_decl

state_claim_decl
    claim_name :: *schema_body*

trans_claim_decl
    claim_name (*param_list*) :: *schema_body*
    claim_name ( ) :: *schema_body*
    claim_name :: *schema_body*

schema_name
    identifier

claim_name
    identifier

param_list
    var_decl
    param_list ; *var_decl*

schema_body
    [*declaration_part* | *formula_part*]
    [*declaration_part*]
    formula

declaration_part
    declaration
    declaration_part declaration
    declaration_part ; *declaration*

declaration
    schema_import
    const *schema_import*

var_decl
const var_decl

schema_import
schema_name
schema_name ( )
schema_name (*arg_list*)

var_decl
variable_list : type_expr
variable_list : partition_type

variable_list
identifier
variable_list , identifier

type_expr
given_type
set_type
rel_type
sequence_type

set_type
set given_type
one *given_type*

rel_type
given_type ↔ *given_type*
given_type → *given_type*
fun *rel_type*
inj *rel_type*
tot *rel_type*
suj *rel_type*
sequence_type
seq *named_type*
tot seq *named_type*

partition_type
identifier part *given_type*

formulas
formula
formulas formula

formula
(*formula*)

4.20

formula
¬ *formula*
formula ∧ *formula*
formula ∨ *formula*
formula ⇒ *formula*
formula ⇔ *formula*
invocation
invocation ; *invocation*
elem_formula

invocation
schema_ref (*arg_list*)
schema_ref ( )
schema_ref

arg_list
expr
arg_list , *expr*

schema_ref
schema_name
schema_ref '

elem_formula
expr @*expr*
expr ∉ *expr*
expr = *expr*
expr ≠ *expr*
expr ⊆ *expr*
expr ⊂ *expr*
relational_predicate
set_predicate

relational_predicate
fun *expr*
inj *expr*
tot *expr*
suj *expr*
seq *expr*

set_predicate
one *expr*

variable
identifier

expr

scalar_expr
set_expr
relation_expr

scalar_expr
variable
relation_expr . *scalar_expr*
first *relation_expr*
tail *relation_expr*
(*scalar_expr*)

set_expr
{ }
{ *scalar_exprs* }
variable
given_type
relation_expr . *set_expr*
set_expr $\cup$ *set_expr*
set_expr $\cap$ *set_expr*
set_expr $\setminus$ *set_expr*
dom *relation_expr*
ran *relation_expr*
(*set_expr*)

relation_expr
{ }
Id
Un
variable
relation_expr $\cup$ *relation_expr*
relation_expr $\cap$ *relation_expr*
relation_expr $\setminus$ *relation_expr*
relation_expr $\oplus$ *relation_expr*
relation_expr ; *relation_expr*
set_expr $\lhd$ *relation_expr*
set_expr $\lhd\!\!\!-$ *relation_expr*
relation_expr $\rhd$ *set_expr*
relation_expr $-\!\!\!\rhd$ *set_expr*
relation_expr $^{\sim}$
relation_expr $^{+}$
relation_expr *
next *relation_expr*
(*relation_expr*)

{ *scalar_maplets* }

scalar_maplets
scalar_expr ↦ *scalar_expr*
scalar_maplets, *scalar_expr* ↦ *scalar_expr*

scalar_exprs
scalar_expr
scalar_exprs , *scalar_expr*

4.5 Ascii Versions of Typographic Symbols

Symbol Name	*Typo*	*Ascii*
Set membership	∈	in, :
Non-membership	∉	not in, !:
Non-equality	≠	!=
Subset	⊆	<=
Proper subset	⊂	<
Union	∪	U
Intersection	∩	&
Relational override	⊕	(+)
Domain restriction	◁	<:
Range restriction	▷	:>
Negative domain restriction	⩤	<;
Negative range restriction	⩥	;>
Function constructor	→	->
Relation constructor	↔	<->
Maplet constructor	↦	->
Implication	⇒	=>
Bi-implication	⇔	<=>

4.6 Semantics of Relational Operators

For each operator, the tables below give an expression using the operator, type constraints, and the meaning of the expression.

4.6.1 Standard set-like operators

Union

$P \cup Q$	$P, Q: S \leftrightarrow T$	$\{(a, b) \mid (a, b) \in P \vee (a,b) \in Q\}$

Intersection

$P \cap Q$	$P, Q: S \leftrightarrow T$	$\{(a, b) \mid (a, b) \in P \wedge (a,b) \in Q\}$

Difference

$P \setminus Q$	$P, Q: S \leftrightarrow T$	$\{(a, b) \in P \mid (a, b) \notin Q\}$

Subset

$P \subseteq Q$	$P, Q: S \leftrightarrow T$	$\forall (a, b) \in P.\ (a, b) \in Q$

Proper subset

$P \subset Q$	$P, Q: S \leftrightarrow T$	$P \subseteq Q \land P \neq Q$

4.6.2 Relation operators from Z

Universal relation

Un	$Un: S \leftrightarrow T$	$\{(a,b) \mid a \in S \land b \in T\}$

Identity relation

Id	$Id: S \leftrightarrow S$	$\{(a,a) \mid a \in S\}$

Domain

$dom\ P$	$P: S \leftrightarrow T$	$\{a \mid (a, b) \in P\}$

Range

$ran\ P$	$P: S \leftrightarrow T$	$\{b \mid (a, b) \in P\}$

Domain restriction

$s \lhd P$	$s: set\ S;\ P: S \leftrightarrow T$	$\{(a, b) \in P \mid a \in s\}$

Range restriction

$P \rhd s$	$s: set\ T;\ P: S \leftrightarrow T$	$\{(a, b) \in P \mid b \in s\}$

Negative dom restriction

$s \lhd\!\!\!- P$	$s: set\ S;\ P: S \leftrightarrow T$	$\{(a, b) \in P \mid a \notin s\}$

Negative range restriction

$P \rhd\!\!\!- s$	$s: set\ T;\ P: S \leftrightarrow T$	$\{(a, b) \in P \mid b \notin s\}$

Relational override

$P \oplus Q$	$P, Q: S \leftrightarrow T$	$(dom\ Q \lhd\!\!\!- P) \cup Q$

Forward composition

$P\ ;\ Q$	$P: S \leftrightarrow T;\ Q: T \leftrightarrow U$	$\{(a, c) \mid \exists b.\ (a, b) \in P \land (b,c) \in Q\}$

Transpose

$P^{\sim}$	$P: S \leftrightarrow T$	$\{(b,a) \mid (a, b) \in P\}$

Transitive closure

P^{+}	$P: S \leftrightarrow S$	$P \cup (P\ ;\ P) \cup (P\ ;\ P\ ;\ P)$

Reflexive/transitive closure

P^{*}	$P: S \leftrightarrow S$	$Id \cup P^{+}$

4.6.3 Extra relational operators

Application

$P.x$	$P: S \leftrightarrow T;\ x: S$	*the single y such that* $(x,y) \in P$

Image

P.s	$P: S \leftrightarrow T;\ s: \mathit{set}\ S$	$\{y \mid \exists x \in s.\ (x,y) \in P\} = \mathit{ran}\ s \lhd P$
Functional domain		
fdom P	$P: S \leftrightarrow T$	$\{a \mid (a, b) \in P \wedge (\forall c.\ (a,c) \in P \Rightarrow c = \mathrm{b})\}$
Function predicate		
fun P	$P: S \leftrightarrow T$	$\forall (a,b).\ (a,c) \in P.\ b = c$
Injection predicate		
inj P	$P: S \leftrightarrow T$	$\forall (b,a).\ (c,a) \in P.\ b = c$
Surjection predicate		
suj P	$P: S \leftrightarrow T$	$\mathit{ran}\ P = T$
Totality predicate		
tot P	$P: S \leftrightarrow T$	$\mathit{dom}\ P = S$
Sequence predicate		
seq P	$P: S \leftrightarrow S$	$\mathit{fun}\ s \wedge \mathit{inj}\ s \wedge \mathit{one}\ \mathit{dom}\ (s \cap \mathit{Id}) \wedge (s \cup s^{\sim} \cup \mathit{dom}\ s \lhd \mathit{Id})^{+} = \mathit{dom}\ s \lhd \mathit{Un} \rhd \mathit{dom}\ s \wedge (s \cap \mathit{Id} = \{\} \vee s = \{\})$
Singleton predicate		
one s	$s: \mathit{set}\ S$	$\#s \leq 1$

4.26

Chapter 5
Glossary

case A case is a binding that associates a value with each variable in a formula. The formula may be a *claim* or a *schema*. If the formula is true for the binding, the case is an *example* or *model*; if it is false, it is a *counterexample*.

characteristic function The characteristic function associated with a partition maps each element of the partitioned type to some set of labels. In the partition declaration

$$s1, s2, s3, \ldots : f \; part \; X$$

$s1$, $s2$, etc. are the equivalence classes; f is the characteristic function; and X is the partitioned type. There is one label for each equivalence class in the partition, so the label designates to which equivalence class the element belongs. The label set is not accessible in NP, but the function is nevertheless useful; the formula $f' = f$, for example, says that the characteristic function f does not change, and thus neither do any of the equivalence classes.

claim A claim is an assertion that a *specification* satisfies some intended property. Formally, however, a claim is just a formula, and need not be related to the *schemas* of the specification. An instance of a claim is a *counterexample*: a case for which the claim is false.

closure The x-closure of a relation R is the smallest relation containing R that satisfies the property p. In particular, the transitive closure of R, written R^+ is the smallest relation containing R satisfying the law $R \; ; \; R \subseteq R$, and the reflexive/transitive closure of R is the smallest relation containing R that is transitive and additionally includes the *identity* relation.

constant A constant is a variable whose value is not affected by the execution of an operation. To declare a variable as constant in NP, precede its declaration with the qualifier *const*. When the qualifier precedes a schema name, all variables in the schema are treated as constants.

counter Short for *counterexample*.

counterexample A counterexample is a *case* for which a formula is false. The counterexample of a claim is a case in which the claim does not hold.

derived variable A derived variable is a variable that has been identified by static analysis as not requiring enumeration, because it can be obtained from other variables by a straightforward calculation.

enumerated variable A variable whose values are explicitly enumerated in the search for instances of a schema or a claim.

example An example of a *schema* is a binding of its variables to values for which its formula is true.

given type A type that has no structure or semantics.

injection A relation that never maps two domain elements to the same range element is an injection. Note that an injection is not necessarily a function.

instance An instance is a binding of values to variables for which a schema is true, or for which a claim is false. See *case*.

invariant An invariant is a formula that is either preserved, or claimed to preserved by an *operation*. The term is also used more loosely to refer to any explicit constraint on the state space.

isomorph elimination One of Nitpick's reduction mechanisms; it works by avoiding the enumeration of isomorphic *cases*.

logging A mechanism that saves transcript output to a file.

model A model of a formula is a binding of values to variables for which the formula is true. An *instance* of a *schema* is a model of the schema's formula.

Nitpick The checking tool.

NP The language in which Nitpick's specifications are expressed. NP is neither hard nor complete.

onto A relation is onto if every element of the range type is mapped to; a synonym of *surjective*.

partition A partition of a type is a family of sets such that each element of the type belongs to exactly one of the sets.

reduction A reduction in Nitpick is a mechanism used to reduce the space of *cases* examined in the search for *instances* of a *schema* or *claim*.

run A run of a *schema* or a *claim* is an attempt to find *instances* by enumerating values of variables.

scalar A scalar is a variable whose type is one of the given types. There are no predefined scalar types in NP.

schema Schemas are the basic structuring mechanism in NP. A schema is a named pairing of a collection of variable declarations and a formula constraining the variables declared.

scope A scope associates a small positive integer with each *given type*, and constrains Nitpick to examine only those *cases* constructed from at most that number of values of the given type. Scopes may also be associated with *claims* and *schemas*, and may constrain the values of a relation variable (by limiting them to functions, for example).

sequence A sequence in NP is a relation that orders the elements of a type into a contiguous list without repetitions. The relation is a function that maps each element to its successor; the last element is mapoed to itself.

short-circuiting One of Nitpick's reduction mechanisms; it works by pruning subtrees of the enumeration that are known in advance not to yield *instances* because the value of the *claim* or *schema* can be determined from the values of only a subset of the variables.

specification A description of some system being modelled. Used loosely to include both *schemas* (the specification proper) and *claims* (redundant assertions about the schemas).

state schema A *schema* describing a set of states.

surjective A relation is surjective if every element of the range type is mapped to; a synonym of *onto*.

total A relation is total if it maps every element of the domain type.

transition schema A *schema* describing a set of state transitions, distinguished by a (possibly empty) parameter list.

Z A specification language developed at Oxford University.