COMP31111 Exercises

COMP31111 Exercises

These exercises are intended to give you familiarity with the elements of rigorous development using the approach encapsulated in Eschertech's *Perfect Developer* tool and methodology. *Perfect Developer* isn't the only such tool or methodology; there are many other comparable systems. But *Perfect* does have a convenient basis in object oriented design and development —which should tie in with your earlier experience— and it does readily generate Java source code. The Java can be combined either with a *Perfect*-generated front end (our preferred approach), or with bespoke Java written outside the *Perfect* system, and the result can be compiled through the *Perfect* tool, to yield a running system.[1] These notes have been prepared on the basis of a Windows installation of *Perfect*. Most of the exercises are not assessed, but the last exercise of this set has to be submitted for assessment. The submission mechanism is via *Blackboard* — log in to *Blackboard* and follow the instructions there to upload your solutions.

Preliminaries

The lab machines will have been set up with *Perfect Developer* and *Crimson Editor*. (N.B. *Crimson Editor* is a free text based editor, and Eschertech have built convenient config files to allow *Crimson Editor* to display *Perfect Developer* source files nicely.[2] Confirm that *Crimson* is set up for *Perfect Developer* when you work on a machine by checking the *Editors* dialog of the *Options* menu, and setting the path to the executable and clicking the *Crimson* button if necessary.)

If you are working on your own (Windows) machine, download and install *Perfect Developer* and *Crimson Editor* using (a CD burnt from) the ISO image link supplied during the lectures. Install using the individual installers for *Perfect Developer* and *Crimson Editor*.

Next, download and unzip the *Perfect Developer Examples* zipfile from the link in the Practical Work section of the COMP31111 web site, and put it somewhere convenient to work with. It's best if all your *Perfect Developer* projects are created in this folder, since some of the examples there contain critical bits of code that you will need to copy and replicate in your own projects, particularly when you want to create a running Java implementation.

Next, download and unzip the *Exercise Sources* zipfile from the link in the Practical Work section of the COMP31111 web site and put it somewhere convenient. It contains various folders, *Ex1*, *Ex2*, etc., containing source files for you to use in the corresponding exercises below. You are now ready to go.

Exercise 1

Create a folder called *SetEx* in the *Examples* folder. This will contain the working project for the first few exercises. Copy the files from your *Ex1* folder into *SetEx*. Start *Perfect Developer* and click on *New project....* in the *File* menu. Name the project *MySetEx* (just to be definite), and navigate to your *SetEx* folder. Click *Save*. Using either the *Add file...* entry from the *Project* menu, or the "plus" symbol in the toolbar, add the three files you copied to the project. Using either the *Save project* entry from the *File* menu, or the "floppy disk" symbol in the toolbar, save the project. Now, using either the *Re-build all...* entry from the *File* menu, or the "double thunderbolt" symbol in the toolbar, build the whole project.[3] It all works. Now, using either the *Verify* entry from the *Build* menu, or the **green** "tick" symbol in the toolbar, verify the whole project. It all verifies unproblematically.

1. In fact, the precise instructions on generating Java and executables referred to later in this document do not quite work as given, because of the way the School's filestore is organised. See the end of this document for some hints about how to get things to work.
2. *Perfect Developer* knows about a number of editors. The link between *Perfect Developer* and such an editor can be conveniently set up. A single click in the *Editors* dialog of the *Options* menu configures *Perfect Developer*, provided the path to the executable is set correctly. For any such editor, configuration files need to be copied into the editor installation. If necessary, follow the instructions in the relevant subfolder (eg. *Crimson*) of the *EditorCustomizations* folder of the *Perfect Developer* installation.
3. *Rebuilding* the whole project (instead of merely *build*ing an edited part of it) as a matter of course avoids any version clashes, and doesn't cost any appreciable time on the tiny projects dealt with in this course.

Let's now pause to examine what we have been doing by looking inside the project itself. Along the way we will see quite a few aspects of *Perfect* source in its context, useful as a supplement to the fairly sparse documentation about *Perfect* that is available.

Double click on the *ExsAlph.pd* line in the upper window of *Perfect Developer*. The *ExsAlph.pd* source file opens in *Crimson Editor*. It contains one line, the declaration of an enumeration class, `ExsAlph`, consisting of four elements `a,b,c,d`. There is not much more to say here, except that the elements of the enumeration class are **ordered**. In *Perfect*-speak this means `a~~b = below`, which is equivalent to `a < b`; also `c~~a = above` which is equivalent to `c > a`, and so on. In particular, enumerations have a minimum element.

Open the *ExsChar2elt.pd* file. Again it is a class file, this time containing the `ExsChar2elt` class. This is a constant mapping from individual characters (class `char`) as read from external input say, to the elements of `ExsAlph` which we want to work with. Note the syntax. The actual data is in the `abstract` section, since that is where (unrefined) data lives. It is a `const`ant mapping, since we will not need to change it dynamically. The constant is called `theC2Amap` and its class is: `map of (char -> ExsAlph)`. Following the definition symbol `^=` we have the data of the mapping itself. It is in the form of a constructor call, namely the class name followed by the data in curly brackets, thus: `map of (char -> ExsAlph){'a' ->` ExsAlph `a , ...}`. Note in particular the syntax for the data. The `'a'` refers to a single character of class `char` (note the back-quotes), and the `->` points to the value mapped to, which in this case is `ExsAlph a`, i.e. the element `a` of the class of interest, **preceded by** the class name (`ExsAlph` in this case). The latter is necessary to tell the compiler in which class the `a` is intended to be interpreted, and every occurrence of an element of an enumerated class, anywhere in the code, should be written similarly, i.e. as `ExsAlph a`. The `abstract` section is followed by the `interface` section, which in this case amounts just to `function theC2Amap`. This says that the abstract data of the class, i.e. `theC2Amap`, should be made visible to clients of the class. Note the syntax. A genuine function declaration would need parameter and result declarations, not to mention a body. The special form here is just for exposing abstract data.

Open the *ExsSet.pd* file. It is another class file, for class `ExsSet`. Unlike the previous class files, which were static, this has some dynamic data. The `abstract` section opens with a `var`iable declaration, so this class contains updatable state. The variable is called `theSet` and is of class `set of ExsAlph`. In fact the latter is an instance of the polymorphic class `set of` *X*, where *X* has been instantiated to `ExsAlph`. The remainder of the `abstract` section defines a helper function: `toSeq`.

The `toSeq` function turns the `set of ExsAlph` into a `seq of ExsAlph`, i.e. it sequentialises the set.[4] It provides a nice example of how to write a (mathematical, abstract) recursive function in *Perfect*. The input parameter is `ss : set of ExsAlph`, and the result is of class `seq of ExsAlph` as expected. The `decrease #ss` tells the verification system what natural number valued measure to use to guarantee the termination of the sequence of recursive calls when `toSeq` is called on some specific data value. (This is because *Perfect Developer* will not tolerate loops or sequences of recursive calls, that it cannot prove will terminate after a finite number of steps. More on this shortly.) The body of the function is a guarded expression. This is enclosed in parentheses, and consists of a comma separated sequence of two cases, each in turn consisting of a guard and a returned expression. The guards of the sequence of cases are tried in turn, until one that evaluates to `true` is reached. The `[ss.empty]` is the first guard, calling the built in `empty` member of the `set of ExsAlph` class. If this returns `true`, then the function invocation as a whole returns the empty sequence `seq of ExsAlph{}`, the syntax being that of a constructor call as above. If the `[ss.empty]` guard returns `false`, the next guard is tried. This is the default guard `[]`, equivalent to `[true]`. If control reaches here, `ss` is bound to be nonempty, and then the function invocation as a whole returns a value which is the concatenation (syntax `++`) of two pieces. The first piece is the minimum element of `ss` (which we noted above was well defined), delivered by `ss.min`, which is turned into a one-element sequence `seq of ExsAlph{ss.min}`. The second is the recursive call to `toSeq`, on the set that is obtained by `remove`ing the minimum element `ss.min` from `ss`. If you think about it now, `#ss`, the size of `ss`, is a good measure to choose for proving the termination of the sequence of recursive calls to `toSeq`, since we do actually make `ss` smaller at each iteration. By reasoning about what happens to the size of `ss` when we `remove` something from it, the verifier should be able to deduce that the sizes of `ss` at successive calls to `toSeq` form a strictly decreasing sequence of integers, bounded below (since the size of a set cannot be negative); and thus to deduce that the sequence of calls to `toSeq` as a whole must terminate. In fact you can be sure that all this is true since the verification you already did earlier (in which all this reasoning was done behind the scenes by the tool) succeeded.

4. Eventually it will become clear why we want to do this.

The `interface` section starts by exposing the `theSet` variable using `function theSet`. The next member, `function check`, says whether the input element is an element of the current value of `theSet`. The next member, `function deliver`, converts `theSet` into a `string` using the built in member function `toString`. The next member, `function theSeq`, converts `theSet` into a sequence, using the helper function discussed above. Note the different syntax used in the bodies of these two functions. In the `deliver` function, the syntax is *object_name.member_function_name* (where the object is `theSet`, being of class `seq of ExsAlph`) since `toString` is a member function of the `seq of ExsAlph` class by virtue of its being a member function of the `anything` class, from which every other class inherits it. In the case of the `theSeq` function, the syntax is *function_name(parameter_name)*, because the function, in this case `toSeq`, is an internal function, not part of the collection of interface members of the `ExsSet` class; its parameter is thus the object `theSet`.

The next member is a schema, `schema !add`. Its being a schema —indicated by the keyword `schema` and the `!` decoration on the name `add`— means that it updates the object's state. It takes an `ExsAlph` parameter, which it adds to the set. The state update is accomplished by the `post` line. The keyword `post` indicates that what is being described is a change in the current object's state, namely a change in the value of `theSet`. The assignment operator in *Perfect* is `!=` (note that you can have space between the two characters). The value to be assigned is the result of `append`ing the `ExsAlph` parameter `x` to `theSet`, which is legitimate, since the fact that `theSet` is an object which is a set of something, implies that it has the member function `append`. The next member is also a schema, `schema !del`, which also takes an `ExsAlph` parameter, which it removes from the set. This schema has a precondition `pre x in theSet`. A precondition is an assumption by the schema that it will never be called in circumstances in which the precondition is not true — viewed alternatively, it places an *obligation* on the calling environment to *ensure* that the precondition is true before a call is made. So the precondition of `!del` allows `!del` to assume that it will never be asked to remove any element that is not an element of the current value of `theSet`. The actual change of state is accomplished in the `post` line (which can also be written using the alternative syntax `theSet! = theSet -- set of ExsAlph{x}`). Either way, this is a command to update the value of `theSet` to the set difference (syntax `--`) between its original value and `set of ExsAlph{x}`, the latter being a constructor call that turns the individual element `x` into a singleton set — note that we could also have written this using `remove` (by all means try it). Note also that the precondition here was not strictly necessary, since both set difference and `remove` are well defined, even if what is being taken out of the set is not actually in it to start with. That is not the point though. Preconditions are not limited to just policing what *must* be true for something to work. They are equally legitimate for policing what you *desire* to be true (for whatever reason) when a call is made. In general, it might be very beneficial to insist on a much stronger set of conditions holding at a call, than what is strictly needed in some abstract sense. For example, the stronger conditions might lead to a much more efficient implementation of the most frequently used cases, if we insist on just those cases using the precondition (thereby throwing the **responsibility of ensuring that only such cases arise** onto the calling environment). The final item is the `build{}` constructor. A call in the external environment by a client of a class *className* taking the form *className{parameters}*, causes a call to the `build{}` constructor of *className* to be made with the supplied parameters, creating an instance of the class as specified by the parameters. So the `build{}` constructor tells the system how to fulfill such constructor calls. Since it changes the (overall) system state, it is described using a `post` command, like other state change commands. In our case, an empty `set of ExsAlph{}` is built.

The above illustrates briefly how it goes with *Perfect Developer* at the class level, when the application being described has a clean abstract design. The class descriptions are succinct and clear, the build process works unproblematically, and in the vast majority of cases, verification is completely automatic if you have not made mistakes in the design. Obviously, if you **have** made mistakes in the design that lead to the design being inconsistent in some way, then verification will throw up failures of one kind or another; we'll leave aside that possibility for now.

Exercise 2

Copy the file *Main.pd* from your *Ex2* folder into *SetEx*. Predictably enough, *Main.pd* contains the code that will actually cause running class instances to be created and executed. Add it to the project in the usual way and open it in the editor. Make sure you can comfortably view lines that are about 110 characters wide for best results below. Before we actively **do** anything with *Main.pd*, we will go through it line by line since it illustrates many useful points that are poorly documented elsewhere or not documented at all.

The first line is commented out: `// import "?.pd"` etc. Ignore this, it is only needed when separate compilation is contemplated. The next nonblank line starts: `schema main(...)`. This declares that main is a schema like we saw above. It says that `context` is a variable of class `Environment`, the `!` decoration on `context` indicating that it is an **updatable** parameter of `main`, and the `limited` keyword indicating that the generality of the `Environment` class can be curtailed for convenience, (in general `context` will contain the file handles and similar items that allow communication with the host operating system). The `args` parameter refers to the comand line parameters, and the updatable `ret` parameter is intended for the result code returned by the execution of `main`. The next line has a precondition that just stipulates that there is a nonzero number of `args`. So far everything is standard, that is to say, when you use *Perfect Developer* to generate a template *Main.pd* of your own (which is an option when you generate a template class file from the *Project* menu) these lines appear as described and you should not alter them.

The detailed stuff starts with the next line. Being a schema, `main` must specify its effects via a postcondition. The `post` line gives this. Here, it is given using the most general clause for specifying update in *Perfect*, the `change` *`variables`* `satisfy` *`conditions`* clause. The only updatable abstract variables around right now are the parameters of `main`, i.e. `context` and `ret`. These are therefore the variables mentioned in the `change` *`variables`* part. We need to say how these are to be changed, (else *Perfect Developer* will complain). What should we say? Well, the real job of a `main` program is to control the main outer loop of interactions with the user. This does not really involve the abstract variables mentioned so far in any sensible way, since we cannot predict what the user will actually want to do. So the best thing is to say something which is as noncommital as possible here. The least commital statement we can make is to say that the values of these variables should be members of their respective classes. Unfortuntely *Perfect* does not provide syntax to say this directly, so we use a slightly roundabout route. The `satisfy` *`conditions`* clause of the `post` line contains the term `undefRel(context' , ret')`, which, on looking towards the end of the file, we see is a polymorphically defined relation, defined in both binary and ternary versions, which just returns `true` whenever its parameters are well typed — it will do the job we want. We also have the condition `ret' = 0` saying that the returned result code should be zero; this is good enough for us. Note the presence of primed variables in the `satisfy` *`conditions`* clause. They refer to the after-values of the variables (i.e. after the update has completed) in syntactic contexts which demand a predicate. When the update is actually specified via a predicate (as in the satisfy conditions clause), it is obligatory to mention the after-values.

So far we have specified the behaviour of `main` abstractly and loosely, but also uselessly, since the specification makes no contact with what needs to happen operationally in the main loop. This defect is remedied by **refining** the abstract definition, and in *Perfect* this can be done by giving the abstract definition an **implementation**, contained in a `via` clause. The `via` clause contains a sequential program, of a style similar to many programing languages, but adorned with additional data to support verification.

The implementation starts by declaring a variable `myset`, declared (`:`) to be of class `ExsSet`, and initialised to the only buildable `ExsSet` via the constructor call `! = ExsSet{}`. When `main` is executed, this line causes an instance of the `ExsSet` class to be brought into existence. The next line shows how to print something to the command line using the wealth of facilities provided by the `Environment` class (as instantiated in our case by the variable `context`). Note the syntax *object_name***!***member_schema_name* for calling a schema to update an object (in this case `context` is the object and `print` is the schema that updates its state by appending something to the standard output stream).[5]

The next line gets to the heart of the matter by starting the description of a `loop`. The next line declares a variable (whose scope is just the loop itself) called `count`, declared (`:`) to be of class `nat`, and initialised to 100 `! = 100`. It will serve as a loop counter, preventing nontermination of the loop.[6] The next line says `change context , myset`. This tells *Perfect* which of the updatable variables currently in scope are to be updated in the loop body, so that the *Perfect* verification system can formulate the right verification conditions. If you change an in-scope variable in the loop body without declaring it in the `change` clause, *Perfect* will complain. The next line says `keep undefRel(context' , myset' , count')`. The keyword `keep` introduces the loop invariant. The loop invariant describes a property (of all the variables to be altered in the loop body) that persists from iteration to iteration. Insisting on having such a property makes a lot of sense in genuine pro-

5. You may consider the bounteous provision of different syntax forms for calling things, i.e. *object_name!member_schema_name*, *object_name***.***member_function_name*, *function_name(parameters)*, to be generously indulgent. Me? I couldn't possibly comment.
6. For a genuine top level control loop, we don't really want predetermined termination; we want the user to be in control of termination. However in formalisms like *Perfect*, loops which do not terminate are taboo, so since we are using *Perfect* to do the top level control, we have to guarantee termination whether we like it or not. A better option is to do the top level control using Java. There are some remarks about that later.

gram development, where it gives rise to verification conditions that help to ensure the correctness of the whole enterprise. However, in the context of the top level control loop, there is little of use that can be said. So we again make use of our friend `undefRel` (this time in its ternary version), to just say that the variable values are elements of their class. Note that we have to state the loop invariant in terms of the primed variables.

The next line says `until count' = 0`. The `until` expression is checked after every iteration, and if it evaluates to `true`, the loop is terminated. The next line says `decrease count'`. The `decrease` expression forms the heart of a verification condition that enables the *Pefect* verification system to deduce that the loop will terminate (without actually executing the loop). It has to be an expression which the *Pefect* prover can **prove** decreases at every iteration, and is bounded below. (So the mechanism is the same as for proving the termination of chains of recursive calls, that we saw above.) As for other predicates about the loop, it has to be stated in terms of the primed variables. We see that there are two aspects controlling loop termination: the `until` clause which can cause loop termination at runtime, and the `decrease` clause which ensures that there are bound to be only a finite number of iterations anyway. The two mechanisms interact. If, as in our case, the `until` clause merely restates the minimum value of the expression policed by the `decrease` clause, the `until` clause can be omitted and the runtime check generated from the `decrease` clause takes care of termination. The `decrease` clause ends with a semicolon, after which we finally see the loop body.

The next line starts the loop body. Ours starts by declaring two variables, `line` and `rret`. The loop body is an inner scope. A consequence of this is that any variables declared here are created afresh at each loop iteration and destroyed when that loop iteration completes. So, since they do not persist between iterations, they do not need to figure in the `change`, `keep`, `decrease` clauses discussed above. Note that `line` and `rret` are not initialised. The next line says `count! - 1`, which could also be written `count! =count - 1`. Obviously, this decrments the `count` variable. As this is one of only two places that `count` is mentioned in the loop body, it is not going to be too hard for the *Pefect* prover to deduce that the value of `count` decreases monotonically at each iteration, and since `count` is a `nat`, it has a minimum permissible value, namely zero. Thus the runtime check for a zero value of `count` will guarantee termination.

The next line reads a line of text from the command line. Details of interacting with the command line are held in the variable `context`, and its `readLine` schema does what is necessary to do the reading. Note the two parameters, `line!` and `rret!`. The `!` decoration tells *Perfect* that you expect these variables to be updated by the schema. Also, note that we could not just reuse the abstract `ret` variable for the return value, since `ret` and `rret` belong to different classes (check their declarations).[7]

Now that we have an instruction from the user, we can react to it. The next part of the loop body is a big conditional statement, doing different things according to the user's wishes. Note the slightly different syntax for conditional expressions (which we saw earlier) and conditional statements. Conditional expressions are enclosed in parentheses, and separate their cases using commas. Conditional statements are enclosed in `if ... fi` and separate their cases using semicolons. There are many aspects of the *Perfect* source language where the declarative syntax (used in expressions and predicates and predominantly in the abstract parts of classes) differs slightly from the command syntax (used in schema postconditions and in implementations). Keep an eye out for these.

The `if` statement works rather like the conditional expression. The guards are tried in order, and for the first guard that evaluates to `true`, its command is executed. We look at the cases. The first four just take note of the character that was input, and append the corresponding element of `ExsAlph` to `myset` using the `add` schema. The next case causes early termination by immediately setting `count` to zero. The next case causes the current value of `myset` to be printed to the command line by calling the `deliver` function of `ExsSet`. Note that we have to supply the newline explicitly. The next case also causes the current value of `myset` to be printed to the command line, but this time it does it using a custom designed format instead of the default provided by using `toString` on the `myset` variable (which is what the previous case does). It accomplishes this by calling the `theSeq` function of `ExsSet` and then manipulating the elements of the resulting sequence. This illustrates a number of aspects of *Perfect* programming, so let's look at it in more detail. Firstly, the string that is printed is the concatenation of three pieces: the first is a constant announcement, the third is a newline, and the second is more interesting so let's focus on it. It has two sets of parentheses, one nested inside the other. Look at the inner parentheses. They contain a `for` *`bound_variable`* `::` *`collection`* `yield` *`expression`* construct. This is one of the construct types for processing collections (sets, bags, sequences) in *Perfect*. The *`bound_variable`* ranges over all the elements of the *`collection`*, and for each such element so generated, the *`expression`* is evaluated and forms an element of the result collection (which is of the same kind, i.e. set,

7. Question: Suppose the classes of `ret` and `rret` were not different. What would be the consequences of using the abstract variable in the `readLine` schema?

bag, sequence). In our case `x` ranges over the elements of the sequence `myset.theSeq` — this is just the sequentialised version of `myset`, as we discussed before. Each element `x` of this sequence is then processed by the constant map `theA2Cmap` to convert it from an element of `ExsAlph` to a `char`acter. Note the square brackets used for indexing a map. The `theA2Cmap` constant map for doing the conversion appears near the end of the file — observe that it is just the inverse of the map `theC2Amap` discussed earlier. Constants like this can go in any file of the project. We are not finished. We have a `char`, but we will need a `string` if we eventually want to use the `print` schema, so the builtin function `toString` is therefore called,[8] giving the final form of the expression `theA2Cmap[x].toString`. So far we have obtained a sequence of (one-character) strings. However we want a single string to give to the `print` schema, so we concatenate the sequence of one-character strings together using an *`operator over collection`* construct. In our case the operator is `++` (concatenation), and the collection is the thing we just discussed — this is the content of the outer parentheses. In the `over` construct, the collection has to be such that the operator is associative (and commutative if it makes sense) when mapped over the collection — so you can't just have any old operator with any old collection. For concatenation to yield an unambiguous answer, it has to be applied over a **sequence**, and ***that*** is the reason why we designed the `theSeq` function in the `ExsSet` class: `(++ over some_set_of_char)` would simply not work. Observe how much work we did to achieve this simple result. In particular, observe how fussy *Perfect* is about types. No types are ever converted automatically[9] — *Perfect* wants you to be aware of any needed type convertions by making you write them explicitly. This precision also helps to ensure that the verification conditions generated and proved behind the scenes are indeed the right ones, and not ones appropriate to some other problem than the one you have in mind.

The remaining cases are for checking and deleting elements in `myset`. For this, the input line must specify (a) which action is intended, (b) which element is meant. So `line` must be at least two characters long. The first two cases handle `check`ing. The guard for the first of these: (a) ensures that `line` has at least two characaters, `#line > 1`; (b) checks that the first character of `line` is `C` (for check), `line.head='C'`; (c) checks that the **second** character of `line` (i.e. `line[1]` — note the indexing of sequences from zero) corresponds to an element of the `ExsAlph` set, `line[1] in ExsChar2elt theC2Amap.dom`. Note how this last test is accomplished. The string `line` is a sequence of `char`acters, so an element, such as `line[1]`, yields a character. We want to ensure that this character is one that we can apply the `theC2Amap` map to, so we check the domain of that map, `theC2Amap.dom`. However, `theC2Amap` is not the name of anything in the current class (since it is a constant of the `ExsChar2elt` class), so we have to prefix its use by the class name to tell the *Perfect* compiler where to go to find the definition of that name.[10] Finally, we can do the membership test using `in`. If the guard succeeds, we execute the command. This essentially prints the result of a call to the `check` function on `myset`, but again there is some type conversion to take care of. Thus, `line[1]` is first extracted; then `ExsChar2elt theC2Amap` converts it into an element of `ExsAlph`; this is then offered to the `check` function of `myset`, i.e. `myset.check`; this returns a `bool`; this is then converted into a `string` using the builtin `toString`, which together with the newline, can finally be `print`ed. If the guard fails, we try the next guard. The second guard is just the negation of the first (provided `#line > 1 & line.head='C'` holds). This means that we are trying to `check` some `char` that does not correspond to a element of `ExsAlph`. The corresponding command just prints a warning to the user, without calling any `myset` member.

Recall that earlier we talked about preconditions, when discussing various members of `ExsSet`. There, we said that it was the responsibility of the environment of the `del` schema (say) to ensure that its precondition was always met when a call to `del` was about to be made. Well, `check` also has a precondition, albeit a slightly concealed one. The fact is that its parameter is declared as `x : ExsAlph`, which implies the precondition that in every call to `check`, the parameter is indeed an element of `ExsAlph`. The environment for `check` is the `main` schema that we are currently discussing, since that is the place that `check` gets called from (and if `check` were ever to be called from any other place, then that other place would also become part of `check`'s environment). The code we have just been through exemplifies the discharge of the responsibility of ensuring that the precondition is met at every call, since the call to `check` is only permitted when the parameter is of the correct form. If it is not, no call to `check` is made. This arrangement is very typical. We cannot prevent the user from typing any old rubbish he wishes at the command prompt. However, we can filter it and deal with any bad cases at the outer level, only permitting a call to be made to a preconditioned `ExsSet` member when it is safe to do so.

8. We could have avoided this step if we had mapped elements of `ExsAlph` directly to one-character strings.
9. Except for a few very specific cases — see the documentation.
10. So contrast the uses of `theC2Amap` from class `ExsChar2elt` with uses of `theA2Cmap` from the current class.

The next three cases concern deletion. The input line must start with `D` to signal deletion, which must be followed by the character representing the `ExsAlph` element to be deleted from the `myset`. For deletion, the guard for the first case concerns the non-`ExsAlph` case. Why put this case first? Well, the **next** case uses `check` in its guard, and we have just dwelt at length on the need to ensure the precondition of `check` will hold whenever `check` is called. Catching the non-`ExsAlph` cases first therefore ensures this by virtue of the sequential testing of guards in conditionals in *Perfect*. The three deletion cases should now be easy to understand. The final guard is `[]` and has no command. It is the '*else do nothing*' case of the `if` statement.

The end of the `if` statement signals the end of the `loop`. The only other thing that the `main` schema does is to assign zero to `ret`, by the command `ret! = 0`, in order to satisfy the abstract postcondition `ret' = 0`.

That completes our survey of the `main` schema. Time to do something with it. Verify it. It verifies unproblematically.

A `main` schema like this one can also be used to generate runnable code. Consult the *Command Line Java Interface* document in the General Resources section of the COMP31111 web site. Just follow the instructions in the sensible way. (What this means is that whereas the document assumes that you are starting completely from scratch, you have in fact already completed some of the steps, eg. the creation of the *Main.pd* file, so just skip over those points.) Also a couple of minor points are not described quite correctly (eg. the location of the runnable `jar` file right at the end) but it's obvious what to do. You will have to invent a name for the generated Java package. Call it *setex*. Once you have completed all the steps, when you next build the application, the postbuild step will be executed, which will generate object code from all the Java files. You can then open a `cmd` window, navigate to the `output` subfolder of your *SetEx* project repository, and run the application using the command `java -jar setex.jar`.

Run and play with the application a little. Try some small modifications: eg. add a message just before exiting; eg. change the `count` initialisation to a small number and watch the application exit after that many steps; eg. modify `theA2Cmap` to yield `string`s directly, and so optimise the `"P"` case; eg. reorder the cases for deletion to put the second case first, then rebuild and run the application, then try to delete a letter not in `ExsAlph` and see the application crash as it fails the runtime check on the `check` precondition. Etc.

So much for command line interaction with a *Perfect*. In the General Resources section of the COMP31111 web site you will no doubt have also noticed the *Graphical Java Interface* document. That document tells you how to construct a much more attractive interface to a *Perfect* application, with buttons and other nice UI gadgets. You have to build the top level control code directly in Java instead of in *Perfect* (one consequence of which is that you are not plagued by loop correctness and termination baggage at the top level); you have to use the `Swing` library and instantiate the `actionListener` pattern, etc. While this is all outside the scope of this course, you are of course free to experiment with it if you wish.

Exercise 3

So far, we have a simple class `ExsSet`, where everything has been specified at the abstract level, and a `main` schema (which is global, i.e. it does not belong to any class). The abstract part of the `main` schema says next to nothing of any use, so we had to **refine** it to an **implementation**, which was essentially a sequential program, in order to express what we wanted the top level control to do. The programming here was rather messy, in that it was full of type conversion mappings and so on. In a sense, in the context of the `main` schema, this is somewhat inevitable. The `main` schema interacts with the environment, so must deal with I/O and `character`s and `string`s. On the other hand the `main` schema must also deal with the internal classes, so must deal with the types of their members. Doing both necessitates lots of translation back and forth between incompatible types.

Now, we will look at refining the class `ExsSet` to an implementation. We might want to do such a thing when the default translation to executable code provided by *Perfect* for some class is not efficient enough, for example. In the context of classes which do not contain the top-level main loop and interact with the environment, the types are typically much more compatible with each other, so the programming is typically much cleaner.

The basic idea is as follows. At the abstract level of `ExsSet` we have a dynamic set `theSet` containing elements of `ExsAlph` which is itself `{a,b,c,d}`. At the concrete level we will encode everything in a single natural number `theNat` constructed as follows. The presence of `a` in `theSet` will be encoded as exact divisibility of `theNat` by 2. The presence of `b` in `theSet` will be encoded as exact divisibility of `theNat` by 3. The presence of `c` in `theSet` will be encoded as exact divisibility of `theNat` by 5. The presence of `d` in `theSet` will be encoded as exact divisibility of `theNat` by 7. And `theNat` will contain no prime factors other

than 2, 3, 5, 7. Therefore `theNat` will be of the form $2^{n_a}.\ 3^{n_b}.5^{n_c}.7^{n_d}$ where all of the exponents n_a, n_b, n_c, n_d are non-negative, and $n_a \geq 1$ if and only if `a in theSet`, $n_b \geq 1$ if and only if `b in theSet`, $n_c \geq 1$ if and only if `c in theSet`, $n_d \geq 1$ if and only if `d in theSet`.

Let us say at the outset, that from a practical point of view, such an implementation strategy is crazy in the extreme. Our main purpose in pursuing it is not so much to promote it as a practical approach, but rather to use it to illustrate some of the machinery involved in refinement in a succinct manner.

In your *Ex3* folder you will find new versions of *Main.pd* and *ExsSet.pd*. Let us go through these. The changes to *Main.pd* are slight. In the middle of the big `if` statement, there are two new cases, to call what look like two new members of the `ExsSet` class, `clean` and `deliverNat`. The former doesn't appear to return any result, while the latter apparently returns a `string`, since the result is concatenated with a newline and sent to the `print` schema of `context`. That's all for *Main.pd*.

We look at *ExsSet.pd*. It starts as before. However, in the `abstract` section there are two new items, another (unary) version of our friend `undefRel`, and another constant translation map `alph2Num`. Given that we already mentioned that we will be using numbers, the presence of such a map should not surprise you.

Next is a new section, the `internal` section. In a normal class, the `internal` section contains the definitions of the data variables and other entities, in terms of which the `abstract` entities are to be implemented. Their presence signals that although *Perfect* will continue to **reason** about the class using abstract entities where possible, everything involving the state of the class will have to be re-implemented in terms of the new data.

The `internal` section starts with the declaration of the implementing variable. This declares `theNat` to be that subset of the integers whose members are all at least 1: `theNat : those x : int :- x >= 1`. Next comes the re-implementation of `theSet`. Before, *Perfect* knew how to deal with `theSet` by virtue of its abstract definition; now we must tell *Perfect* how to do everything in terms of `theNat`, the first of these things being how to calculate the abstract set from the internal variable. The calculation is the union (syntax `++`) of four sets. Each of these sets either contains a single element of `ExsAlph` (if `theNat` is exactly divisible by the number that represents that element) or is empty (if `theNat` is **not** exactly divisible by the number that represents that element). Since 2, 3, 5, 7 are all mutually coprime, these exact divisibilities are all independent. Note the syntax of the `theSet` implementation as the union of four conditional expressions, one for each of the contributing sets.[11]

The next item in the `internal` section is a re-implementation of equality (for elements of the `ExsSet` class) in terms of the new data. (This shows how much the abandonment of the abstract representation has cost us). The equality is defined as a *Perfect* **operator** that acts between the current object (`self`) and the other argument (`arg`). Since `arg` is assumed to belong to the same class, it is assumed to have a similar internal representation to `self`, which can be referred to using `arg.theNat` (whereas `self`'s internal representation is referred to using just `theNat`, with `self.` suppressed in the usual manner). The equality test (for equality as elements of `ExsSet`), can now be written as the conjunction of equivalences of exact divisibilities by 2, 3, 5, 7 of `theNat` and `arg.theNat`. Note the syntax for this. It uses a `via ... end` construction immediately following the abstract definition, which is the standard format for describing implementations. Furthermore it returns its result using a `value` *`expression`* clause. This is the standard format for returning the value of an expression when the value is being generated by executing a sequence of statements.[12]

Next is the `interface` section. Its first item exposes `theSet` as before. The next item is the re-implementation of `check`. Using a `via ... end` construction once more, the exact divisibility of `theNat` by the numerical representative of the parameter is returned. The next item is the re-implementation of `deliver`. What `toString` did automatically before, is now replicated in low level detail by concatenating `string` representatives of the elements of `ExsAlph` present in order in `theSet`, as revealed by the relevant exact divisibilities of `theNat`, and then adding the remaining parts of what `toString` output. The next item is the re-implementaiton of `theSeq`. It follows a similar pattern to the previous case.

11. Actually, it was slightly inaccurate above, to say that "everything involving the state of the class will have to be re-implemented". Only those parts of the state for which a re-implementation function is provided (such as for `theSet`, as we just described) are re-implemented. Remaining unimplemented abstract state stays as part of the class state. Equally, there can be additional internal state that is not the re-implementation of anything abstract, if that is useful.

12. We could have saved some work by writing `ghost operator = (arg) ;` in the interface section. But then we would not have been able to *use* equality in any non-abstract context, since a `ghost` member is left unimplemented by the source code (and thus the system).

Next is one of the new functions in this version of the `ExsSet` class, `deliverNat`. Its purpose is to allow you, the user, to inspect the changing values of `theNat` as the implemented class runs. As such it has no purpose at the abstract level. Thus, given that its concrete implementation will return a `string` representation of `theNat`, its abstract definition is just to return an unspecified `string`. Note the syntax. The `satisfy` *`condition`* format allows us to implicitly demand that a variable acquires a value that satisfies a given *`condition`* without having to describe an explicit calculation that yields such a value. The precise form of the *`condition`* in this case, namely `undefRel(result)`, uses our friend, the unary `undefRel`, and demands it of `result`, an internal keyword used by *Perfect* to refer to the result returned by a function.

Next is the re-implementation of `!add`. As you might expect, it multiplies `theNat`, the number representing the current value of `theSet`, by the numerical representative, `alph2Num[x]`, of the `ExsAlph` element `x` being added. (Note that the multiplication could also be written `theNat! = theNat * alph2Num[x]`.) After the multiplication, `theNat` will obviously be exactly divisible by `alph2Num[x]` regardless of whether it was exactly divisible or not previously, which is the effect that we want.

Next comes the re-implementation of `!del`. This is more complicated than the previous case since we have to ensure *non-exact divisibility* of `theNat` by `alph2Num[x]` after the completion of the schema, which requires that *all* `alph2Num[x]` factors of `theNat` are removed by the operation. We use a `loop` to ensure this, dividing repeatedly by `alph2Num[x]` until no more `alph2Num[x]` factors remain. Compared to the `loop` we saw earlier, this one works in a much more sensible manner. Let's look at it. The non-loop variables `changed` by the loop consist of just `theNat`. The loop invariant we want to maintain is the non-negativity of `theNat`, i.e. `theNat' >= 1`. The loop termination condition we want is just the non-exact divisibility of `theNat` by `alph2Num[x]` that we mentioned before, i.e. `theNat' % alph2Num[x] ~= 0`. The quantity that is bounded below, and that most naturally guarantees loop termination by decreasing at each iteration is `theNat'` itself. And then we finally get the loop body, which simply divides `theNat` by `alph2Num[x]`.

The next schema is a new one, `!clean`. Now, whereas at the abstract level, each possible value of `theSet` was unique (since `theSet` was **defined** to be so), at the concrete level, the same value of `theSet` (take `{a}` for example) is represented by many different possible `theNat` representatives (i.e. `2,4,8,16,32,`... etc.). The purpose of `!clean` is to discard additional factors of the representing elements from `theNat`, so that each representing element `alph2Num[x]` is present exactly once as a factor of `theNat` (if `x` is in `theSet`), or is not present at all (if `x` is not in `theSet`). This is most easily done by testing `theNat` for exact divisibility by the representing elements, and including the element just once as a factor in the final value of `theNat`, as needed. Note that because of the uniqueness at the abstract level, there is no sensible abstract counterpart of the concrete version of the operation, so the abstract postcondition just says `pass` (which means 'do nothing' in *Perfect*-speak).

The last thing is the `build{}` constructor. Since the numerical representative of the empty set can be any number relatively coprime to 2, 3, 5, 7, (since any such number will be non-exactly divisible by all of 2, 3, 5, 7), we set `theNat` to `1`, since that is the most natural and straightforward option.

Well, that covers what is happening in the refinement. Now build the project again, and run the resulting code. Play with the additional functionality given by `deliverNat` and `!clean` to inspect what is going on underneath as you call various schemas.

Exercise 4

With your setup from Exercise 3, verify the project. (Depending on the speed of the machine you are using, allow around fifteen minutes or more for this — verifying harder things is not such a fast business!) The result is less than completely happy, right? One reason for this is the unnaturalness of the refinement that was implemented. The reasoning processes needed for verifying the various aspects of a language as rich as *Perfect* are not guaranteed to succeed, so their automation has to be curtailed in various ways — the underlying prover focuses its attention on things that are used most frequently and yield success most often. So, although we were able to informally indicate the appropriateness of the refinement as we were discussing the implementation details above —and such a style of argument could be made properly watertight mathematically without too much effort if we really wanted to achieve this— the *Perfect* prover (and many similar ones) are simply not designed to go looking for **those kinds of proof**, since such things arise so rarely in practice. That is the explanation for the various failures to prove particular verification conditions that you saw. Let's leave verification to one side for now.

As an exercise, create a different refinement of the `ExsSet` class, based on concrete data consisting of four boolean variables, each representing the presence or absence of one of the elements of `ExsAlph` in `theSet`.

You should have no problems writing the code, building the project, or running the resulting executable. Finally, verify the project. This should go much more smoothly than before, since representing abstract data by boolean flags is a much more common thing to do in software construction than what we did earlier.

Exercise 5

Time for you to build something from scratch. This is going to be a very simple aeroplane system. An aeroplane can be `on_ground` or `in_air`. Its engine can be `eng_off` or `eng_on`. Its fuel load can be `fuel_low` or `fuel_ok`. This gives you an eight state system. You can `turn_on` the engine provided it is off, the plane is on the ground, and has plenty of fuel. The plane can `take_off` provided its engine is running and it has plenty of fuel. It can then `fly`, which reduces its fuel to `fuel_low`. It can then `land` which puts it back down on the ground. Once it has landed, you can `turn_off` the engine, after which the plane can `refuel`, which replenishes its fuel supply.

Write a simple *Perfect* system that models all the above, and anything else that you need for it to make sense. Make sure that all the schemas have appropriate preconditions, to document the fact that they must only be called when the circumstances are appropriate (eg. you don't want to allow the engine to be switched off when the plane is in the air in our very simple system).

Write a suitable *Main.pd* to control the system from the command line. (You can write a graphical interface too if you wish, but that's not necessary.) Build and run your system.

You can obviously enhance the preceding as much as you want to get as sophisticated a model of an air traffic control system as you like, but bear in mind two things: (1) you have the rest of the course to deal with, so don't get too sidetracked; (2) you will need a copy of the simple system for Exercise 11 (which is assessed), so save it before getting carried away.

Exercise 6

In this exercise we'll revisit the verification of the *MySetEx* project. Here are a few cautionary remarks before we get going.

Firstly, the automated proving behaviour of *Perfect Developer* in the area of verification is a combination of two things, one prosaic, one more technical. On the prosaic side, *Perfect Developer* sets itself a (real) time limit when attempting a proof task. Therefore, a faster machine will be able to get through more work than a slower one in the same amount of time, potentially leading to differences in observed behaviour. On the technical side, the proving subsystems of tools such as *Perfect Developer* are the most changeable aspects of these tools. It is entirely possible that by the time you are doing this exercise, the software might have been updated, and the prover technology improved a bit, and this could cause the behaviour to differ from what is described here. The reason for this is that automated proof is a search process. It happens (assuming that we are trying to prove something that is actually true), that there are (in general) an enormous number of true facts that are derivable about the situation at hand — too many to search through **naively** in the hope of coming upon what we need. Also, most of the time, the theories needed for reasoning about realistic applications are not decidable — that is to say that there is **no** mechanical procedure that is **guaranteed** to determine if a conjecture is true or false (disregarding completely how much time or memory might be expended in the process). So automated provers work by searching through the possibilities, and to have any hope of doing a reasonable job, they have to **search smart**.

The way that they work is by having a bunch of rules, which are applied to some among the collection of facts being worked with, in order to derive even more facts. (There are many specific ways in which this activity might be organised, which we needn't go into.) If you are lucky, the facts you derive amount to the conjecture you are trying to prove and the process can stop. However the internal details of how you organise the application of which rules to which facts at which times makes an enormous difference to the chances of success or failure, and this is the embodiment of the **smartness** of the search mentioned above. The developers of automated proving systems expend a lot of effort to understand the consequences of the policy decisions that they encode into their systems regarding such strategic and tactical questions, and the ongoing development of a proving system often amounts to the adjustment and enhancement of these low level details. Small changes in the internal details can redirect the way that the proof search goes at crucial points, and lead to dramatic changes in the success or failure rate for specific kinds of problem.

Now let's get back to the *MySetEx* project. Go back to the setup from Exercise 3, and verify the project again to get back to the starting point. (Recall that verification typically takes ten or fifteen minutes or more — so since you will be doing lots of verifications in this exercise, it's best to have something else to do while the verifica-

tions complete.) Look at the output of the verification in the *Perfect Developer* Results window. Among all the verification conditions successfully discharged by the prover (highlighted in green), there should be four lines highlighted in yellow, indicating failed verifications. Double click on these in turn, and *Perfect Developer* will open *Crimson Editor* for you if needed, and take you to the source line in the *ExsSet.pd* file associated with the failed verification condition. You see from the text of the error messages that these failed verification conditions are the simulation relationships for the refinements of `deliver`, `theSeq`, `!add`, `!del`. Well, if these four failed, then a number of others must have succeeded. In particular the simulation conditions for functions, schemas and constructors `theSet`, `check`, `deliverNat`, `!clean`, `build{}`, must all have succeeded. That's not too bad.

We will now illustrate some of what can be done by the user to help overcome prover failure in cases like these. Before doing so, we recall the cautionary remarks above. The fact that so much of the observable success or failure of the proving activity depends on fine detail buried in the guts of the proving system, means that there are no (or almost no) universally applicable broad principles that tell you how to approach a failed proof situation and show you how to **guarantee** to improve matters. A lot of it comes down to experience, some of which is generic, but a lot of which turns out to be very specific to the system at hand, and even to the specific version of the software that you are working with. Moreover, for commercial products, the precise details of how the internal mechanics work are valuable intellectual property (because they affect the outcome so much), and thus are held in confidence. The same applies *de facto* to most non-commercial provers too, since prover internal complexity is such that there is essentially no better description of how the prover works than the code of the prover itself. It remains sad, but true, that the only people who can **really** get the best out of a proving system (no matter which system) are the people who wrote it. Long experience in using a given system (along with the endless hours of frustration as you wonder how the hell to get this black box to do what you consider to be relatively obvious) is a great help in actually achieving useful results, but there is no substitute for actually having built the system yourself.

In the light of all that, let's look at the failed verification conditions. The first concerns `deliver` which produces a `string` description of `theSet`. The abstract version uses `toString`, and in *Perfect Developer*, the proving system does not attempt to reason about the `toString` builtin, so there's no chance at all of discharging the verification condition. This kind of thing it not untypical in this kind of system. The `toString` builtin primarily deals with output to the extenal environment, and the external environment is hard to pin down precisely within the constraints of a *Perfect*-like formalism (especially when you take into account what might be involved in implementing *Perfect Developer* on different operating systems — entailing interfacing with their different ways of handling the extenal environment), so *Perfect*-like formalisms in general tend not to try to reason about these aspects.

The next easiest failed verification condition is the one for `!add`. We conclude that it might be the next easiest by thinking about what might be involved in proving the simulation relationship. The schema merely multiplies `theNat` by `alph2Num[x]`. Regarding the relationship between concrete and abstract data, which concerns exact divisibility, the after-value of the state variable, `theNat'`, is certainly exactly divisible by `alph2Num[x]` no matter what. Moreover, if the before-value, `theNat`, was exactly divisible by any other number, `y` say, i.e. `theNat % y = 0`, then the after-value, `theNat'`, certainly remains exactly divisible by `y` unproblematically. The only snag that might arise is that if the before-value, `theNat`, was **inexactly** divisible by some other number, `z` say, i.e. `theNat % z = rem ~= 0`. Then it might arise that exact divisibility was achieved accidentally, `(theNat * alph2Num[x]) % z = (rem * alph2Num[x]) % z = 0`, because one of `rem` or `alph2Num[x]` had a common factor with `z` and the other was just the right size to achieve exact divisibility. We made sure that this wouldn't happen by using prime numbers to represent the various elements of `ExsAlph` — so that multiplying by `alph2Num[x]` would never cause this. But *Perfect Developer* doesn't understand about prime numbers. Given that it is a finite program that inevitably has limited capabilities, why should it? There are certainly many more important things for it to be concerned with. We can encourage *Perfect Developer* by giving it some useful information that might help it.

In the *Ex6* folder you will find file *ExsSet-1.pd*. Look inside it. It is the same as the *ExsSet.pd* you have so far aside from the presence of some new stuff in the `abstract` section. The new stuff comprises a Boolean function called `coPr`; this merely lists some pairs of numbers that are relatively coprime to one another (i.e. their greatest common divisor is 1).[13] There is also an `axiom` called `myAxExact` which says that **if** (syntax: `pre`,

13. You may wonder about why the four cases at the end of `coPr` are placed there. You may be suspicious that they were added as an afterthought, since they would seem to more naturally belong at the beginning of the definition ... and you'd be right. None of this goes smoothly. I'll spare the details.

as for preconditions) `y` is exactly divisible by `z`, **then** (syntax: `assert`) an integer multiple of `y`, `(x*y)` say, will also be exactly divisibly be `z`. And there is another `axiom` called `myAxInExact` which says that if `y` is **not** exactly divisible by `z`, then an integer multiple of `y`, `(x*y)` say, will also **not** be exactly divisibly be `z`, **provided** `x` is coprime to `z`. These certainly look like the kind of facts that might help the verification along. Note that coprimeness is defined, and the axioms introduced, out of thin air here. None of the underlying theory of divisibility has been entered into, even though we could have done so had we wanted to. **However, beware of the Garbage-In/Garbage-Out principle.** *Perfect Developer* assumes that you know what you are doing. If you introduce an axiom, *Perfect Developer* does not question it, or insist that you prove that it is correct before *Perfect Developer* is willing to make use of it.[14] If your axiom is wrong, you may perfectly well derive perfectly ridiculous results. So take care. Now incorporate the new stuff into your *ExsSet.pd*. Rebuild the project to check that you did it correctly, and then verify once more.

Arghhh! There are now **more** errors than previously! Now, !clean, which was fine before, fails. In the *Ex6* folder you will find file *ExsSet-2.pd*. Look inside it. It is the same as the *ExsSet.pd* you have so far, aside from some new stuff in the `internal` section. At first sight, the new stuff seems to be senseless mathematically. All that has happened is that there is a new function `rrr`, identical in functionality to the retrieve function `theSet` that implemented the abstract `theSet` data in terms of `theNat`, and the retrieve function `theSet` has been re-expressed in terms of `rrr`.[15] Why bother? Be patient! Incorporate the new stuff into your *ExsSet.pd*. Rebuild the project to check that you did it correctly, and then verify once more.

Cripes! There are now **fewer** errors than last time! Well, not fewer errors than **ever**, but at least we are not worse off than previously. We have recovered the verification of !clean, and the others are the same as before. Why? Essentially, the introduction of the indirection, although pointless mathematically, resulted in an internal reorganisation of the search process (since the proof search is organised on the basis of the material presented to *Perfect Developer*), and it so happened that the new search process found a proof that it failed to locate last time.

There is another benefit to the indirection. Recall that we started out trying to improve things regarding the failed verification condition for `!add`. If you look at the text of the error message, it is clear that it is the proof of the retrieve relation for the `!add` schema in the after-state that caused the problem (i.e "proving: Specification satisfied at end of implementation"). We would like to be able to give hints to *Perfect Developer* regarding how it might prove this verification condition. Unfortunately, all the overloading of names between abstract data and its implementing retrieve function prevents us (in practice) from interacting directly with the proof process for these functions. However, an arbitrary non-retrieve function is not so constrained. If we can interact with the proof process for `rrr`, to help it to prove for a value equivalent to the after-state for `!add`, then we hope that *Perfect Developer* will be able to make the connection with the `theSet` retrieve function and thus discharge the `!add` schema verification condition.

In the *Ex6* folder you will find file *ExsSet-3.pd*. Look inside it. It is the same as the *ExsSet.pd* you have so far aside from the presence of some new stuff in the `!add` schema. This amounts to a couple of lines in the implementation, i.e. in the `via` clause. Remember that this is a sequential program that implements `!add` using the multiplication by `alph2Num[x]`. The first line is now `let oldNat ^= theNat`. Since nothing else has happened yet, this evidently stores the before-value of `theNat` in the new variable `oldNat`. Then comes the multiplication. The multiplication completes the implementation, after which there is an assertion: `assert rrr(theNat) = rrr(oldNat).append(x)`. Since this occurs after the multiplication, the value of `theNat` is now its after-value, and the assertion says that `rrr` maps `theNat` and `oldNat` to sets that can be reconciled by `append`ing `x`. This is very similar to the verification condition for `!add`. Incorporate the new stuff into your *ExsSet.pd*. Rebuild the project, and then verify once more.

Well there are still as many errors as before. Look at the one for `!add`. It is different. It now refers to the assertion (i.e "proving: Assertion valid in context of class ExsSet"). In one sense we are no better off than before. However, in another sense, we are. Two lines down from the error is the earlier verification condition, which now has discharged (i.e "Confirmed: Specification satisfied at end of implementation"). So we have traded one failed verification for another, but the new one is open to invasive surgery in a way that the previous one wasn't.

---

14. There are systems around that are more fussy and insist that you prove newly introduced facts before they are used elsewhere.
15. All this overloading of names doesn't make explaining things any easier.

Exercise 7

We continue our attempts to verify the the *MySetEx* project. The end of the previous exercise showed how to expose the failed verification condition to the glare of publicity. One could then try to attach proof steps to the relevant assertion (in a way that will become clear below) and thus to discharge it. However, a lot of bitter experience shows that this is not in fact a very productive strategy (at least, not given the time limits that are set by default in *Perfect Developer*). Instead, we proceed as follows.

It turns out that *Perfect Developer* employs a very naive strategy to try and prove things about the *MySetEx* project. As noted earlier, it knows nothing about sophisticated generic concepts like coprimality, so it just does a brute force case analysis and tries its best on each case, typically running out of time before it gets to the end. So the next tack that we will try is to cut down the number of cases. (In fact, cutting down the size of the problem is a tried and tested strategy in verification when you struggle to discharge some verification conditions.)

Copy the *SetEx* project folder into a new folder called *SmallSetEx*. Rename the project file *MySmallSetEx.pdp*. Go through all the source files and remove all reference to `c` and `d`, thus reducing the alphabet to just `a` and `b` — you should know enough about *Perfect* syntax by now to do this easily, but in case you get stuck the relevant files are in the *Ex7* folder, except that the *ExsSet.pd* file is called *ExsSet-4.pd*. Rebuild and verify.

Amazing! the `!add` verification has now succeeded. In fact, it is even better than that. If you remove the `let oldNat...` and `assert rrr...` lines that you added last time and re-verify, the `!add` verification **still** succeeds. So it was indeed true that merely reducing the number of cases to check, enabled *Perfect Developer* to complete them all before the deadline it had set itself had expired. (In fact subsequent *ExsSet-n.pd* files omit these lines.)

Emboldened by the success, we conside the next easiest remaining unproved verification condition. This concerns the function `theSeq`. With the reduction in the number of cases, it doesn't look too daunting to explicitly list all the cases that arise. This notwithstanding, the *Perfect* verifier still got lost somehow, so let's help it. In the file *ExsSet-5.pd*, the implementation of the function `theSeq` acquires some new text, namely an `assert` clause following the imperative statements and before the `value` clause that returns the result of the function. Incorporate this into your project. Rebuild and re-verify. It still fails. In the file *ExsSet-6.pd*, the `assert` clause is enhanced by some proof hints — this example shows you the syntax for doing this. As you can see, the proof hints consist of a case breakdown, splitting up the proof according to the four possible values that `theSet` can have, as reflected in the four possible exact divisibility properties that are relevant to `theNat`. For each case there is an assertion regarding what `theSeq` should be in that case. Incorporate this material into your project. Rebuild and re-verify. It now succeeds! We have our first instance of successfully helping the verifier to actually complete a proof that it formerly failed on.

Exercise 8

The final potentially verifiable thing is the `!del` schema. The files in question are in the *Ex8* folder. The first thing to do is to add an assertion after the last executable statement to say that the concrete code achieves a result compatible with the abstract update in the expected way. The file *ExsSet-7.pd* in the *Ex8* folder contains the required line: `assert rrr(theNat) = rrr(oldNat).remove(x)`. (Of course, you also have to define `oldNat` using `let oldNat ^= theNat` in the way we did before — *ExsSet-7.pd* does this too). Incorporate this in the usual way, rebuild and re-verify. Well, it's no surprise by now that the assertion fails, though, pleasingly, the specification overall has succeeded, and we have exposed the problem area to the light of day. The next step is to encourage *Perfect Developer* to do the right case analysis. The file *ExsSet-8.pd* contains the proof hints required. Incorporate these in the usual way, rebuild and re-verify. Well, it's no surprise that the assertions in the case analysis now fail to prove. Why might this be? Well, if you think about it, the assertions make claims about the relationship between `oldNat` and `theNat`. What might *Perfect Developer* know about this? Absolutely nothing, that's what. So it makes sense to try to help it out a bit here. What is required is to strengthen the loop invariant (the `keep` clause) so that *Perfect Developer* might figure out how `oldNat` and `theNat` could be related, from which it might, with a bit of luck, be able to figure out how discharge the failing verification conditions. The file *ExsSet-9.pd* contains some enhancements to the loop invariant. Incorporate these in the usual way, rebuild and re-verify. Well, it's no surprise that we aren't out of the woods. In fact things seem to have got worse rather than better. There are two new errors that say "Warning: Unable to prove: Loop body preserves loop invariant"). These presumably refer to the two new facts that we have introduced into the `keep` clause. Also there are two errors concerning the first and third of the proof hint assertions. This is strange. Why the first and third and not the other two? We look for some asymmetry in the system that might be related to this different

behaviour. After some thought, it occurs to us that maybe there is an asymmetry in the `coPr` function. It says that 2 and 3 are coprime but not that 3 and 2 are coprime — *Perfect Developer* has no capacity to suspect the latter on the basis of the former. The file *ExsSet-10.pd* contains an enhanced `coPr` function with the extra case included. Incorporate this in the usual way, rebuild and re-verify. Well, it's no surprise that the loop invariant errors are still there, but pleasingly, the four proof hint assertions have now all discharged — we were right about the asymmetry! What is to be done about the loop invariants? We think about it a bit. What the loop body is doing is **dividing** `theNat` by something. However, our axioms about remainders under coprime division only deal with multiplying the candidate number by something, and then stating some properties of remainders. Maybe we need more axioms, ones specifically designed to deal with the division process. The file *ExsSet-11.pd* contains an enhanced collection of axioms, containing two new axioms intended for this purpose. Incorporate these in the usual way, rebuild and re-verify. Well, it's no surprise that there are errors, is it? The first one is the usual `toString` one. Double click on the (only) other error. It's about the result of the `!add` schema. After all that labour, after all that we have done, we have managed to completely prove the `!del` schema, only for the `!add` schema to fall apart. Arghhh!

Exercise 9

After all that we have been through, it must surely be within our power to fix the `!add` schema. The files in question are in the *Ex9* folder. The needed routine ought to be getting predictable by now. The file *ExsSet-12.pd* contains the `let oldNat ^= theNat` and `assert rrr(theNat) = rrr(oldNat).append(x)` lines that we start with. Incorporate this in the usual way, rebuild and re-verify. Well, it's no surprise by now that the assertion fails, though the specification overall has succeeded in the way it did previously. The next step is to encourage *Perfect Developer* to do the right case anaylsis. Because the `!add` schema does not have a non-trivial precondition, there are, a little sadly, more cases to take care of, but not too many. The file *ExsSet-13.pd* contains the case analysis needed. You see that the structure is similar to what we had previously for `theSeq` and `!del`. Incorporate this in the usual way, rebuild and re-verify. It works. Phew! We're finally done.

The Verification Process in General

It is worth highlighting a number of points about what you have just worked through.

- The behaviour you saw is specific to *Perfect Developer Version 4.10*. Earlier releases have slightly different behaviour, and later releases may well also have different behaviour.
- Adding more true facts doesn't necessarily help matters. While the set of **mathematical consequences** that can be derived from the true facts available increases when you add more true facts, the set of **consequences output by the tool** depend on the detailed working of the tool's search process, which is a different matter.
- Despite appearances perhaps, you have been led through a 'santitised' version of the verification process — by no means all of the pitfalls I encountered in putting this together have been replicated above.
- The process you have been through is not untypical of what happens with a **black-box prover**. A great deal of stuff just proves straight away — you've no idea why. A few things don't prove straight away — you've no idea why either.
- I couldn't have done it without help from Eschertech — the free version of *Perfect Developer* gives far too little feedback regarding how to make genuine progress with failed verifications. You can't see the proof tree (something to do with it being the free version), so you usually need the guys who wrote it on board.
- If you are lucky, you can do a certain amount yourself, especially if you develop some experience with the tool, and what you are doing is not too crazy, unlike the development that we have just been through. The *Verification Hints and Tips* file in the General Resources section of the COMP31111 web site offers some suggestions for *Perfect Developer*, which, if they can be applied, can often help things along.
- There are no hard and fast rules to speak of. A certain amount of experience can often help a bit, but each tool has its own quirks, and what works for one tool (or for a given release of a given tool) can't be relied on to work for another tool (or even for another release of the same tool — as noted above).

In genuine industrial situations:

- There is always close interaction between the system developers and the tool developers.
- Unlike us, the system developers can see the proof tree (invariably this costs money) and can better figure out what is going wrong.
- The system people have (or rapidly acquire) extensive experience with the tool and its quirks, and can figure out how to fix straightforward problems. For more difficult issues, the system people tell the tool people the nature of the problem, and the tool people adapt the tool to make it deal with that kind of situation better.

More generally:

- Most systems in this field behave roughly as above.
- In most systems, there is a big gulf between how the tool goes about proving things, and the human view of proving. Often, even when you can see the proof tree, and can see that applying a certain axiom or lemma at a specific point in the failed proof would help things along, there is no mechanism to allow you to tell the system to **apply this particular rule right here right now!**
- The only free system (that I know of) that **does** allow you to manipulate the proof by insisting that a specific rule is applied at a specific point in the proof tree, is the KIV system from Augsburg. But we wouldn't have time to cover KIV to any reasonable extent in a course of 20 hours or so.

Exercise 10

Make a copy of the *Ex1* folder called *SetExProps*. Use *Perfect Developer* to create a project file in it also called *SetExProps*, and add the three files in the directory to the project. In the General Resources section of the COMP31111 web site find the *Dictionary Example Slides (.pdf)* (or the 6up counterpart). It discusses the dictionary example treated at greater length in the *Getting Started with Perfect (.pdf)* which you should have been looking at at some stage. On slide 15, you find (highlighted in red) some `assert`ions and `property`s. The `assert`ions attached to `!add` and `!remove` and `build{}` are postassertions. The `property`s are properties of the system as a whole. There are three things to be said now.

Firstly, although both the postassertions and properties contain `assert` clauses, there is no possible way that they could be **proved**. The reason is simple. The bodies of the definitions of the functions and schemas are all `?`. This means that they have been postponed till later (*Perfect* allows this). So we don't know what all these functions and schemas **actually do**. If we don't know what they all do, how can we possibly prove what these postassertions and properties say? We can't. So at this stage, they define **requirements** of the system. All we can check is that they conform to the types of the various parts of the system (insofar as these are known at this stage, in the absence of the definition of the class's abstract data). If you tried to verify such a class, this limited typechecking would be the extent of what would be checked. However, when the bodies of the definitions of the functions and schemas are added, then we **do** know what they do. At that point, a verification would check that the function bodies and schema postconditions defined functions and schemas that **really did have** the properties asserted.

Secondly, let us focus on the postassertions (of `!add` and `!remove` and `build{}`). These are assertions attached to the definitions of functions and schemas. In this context, `self` refers to the before-state of an instance of the class, and `self'` refers to the after-state (relevant, obviously, only for schemas). As you can see from the bodies of the assertions, any class member that is externally visible (by dint of living in the `interface` section) can be used in constructing the body of such a postassertion. The restriction to visible members is because a **postassertion forms part of the contract** that the class offers to its external environment. This is unlike the postcondition, which defines the class behaviour internally, and **does not form part of the contract unless the class is final**. (If the class is not final, a class that inherits from it is allowed to redefine a schema in a way that contradicts the original postcondition, but **not** in a way that contradicts any original postassertion.)

Thirdly, let us focus on the system properties. These can be used to assert facts about system behaviours as a whole (within the limits of the syntax permitted by *Perfect*). The first property has a precondition, after which it asserts something about the result of two successive member calls applied to a class instance. The class instance is referred to generically as `it`. The first call is to schema `!add`, written `it!add`, and transforms the before-state of the class instance to an intermediate state. The second call is to schema `!remove`, similarly written `it!remove`, and transforms the intermediate state of the class instance to the after-state of the whole sequence. The assertion as a whole claims that the after-state of the whole sequence is equal to the before-state. The class

instance after-state is referred to using `self after ...` (the keyword `after` indicating that it is indeed the after-state). Likewise, the class instance before-state is referred to just as `self` (the lack of any `after` adornment indicating that it is indeed the before-state). Sequences of member calls of any length can have their properties asserted in this way.

The second property says that adding a word to the dictionary does not affect the result that `check` delivers on some different word. From the preceding discussion, the syntax should be clear enough.

Note however, that the member calls in system properties of this kind have to be referred to explicitly by name. *Perfect* syntax is not flexible enough to say things like "the after-state of any execution that starts with an `!add(w)` and does not later include a call to `!remove(w)` will satisfy `check(w)`", since *Perfect* syntax cannot refer to an **arbitrary** execution (which would require variables of execution type).

Thus far, all these statements are just requirements. A few slides later, the bodies are defined, and then the requirements can be verified against these definitions.

Now notice that when the dictionary class has its data defined, its state is of type `set of` *`something`* (the *`something`* being `word` in this case). In our little running example, the state is also of type `set of` *`something`* (the *`something`* in question now being `ExsAlph`). The similar structure means that similar postassertions and properties will hold. Work out what they are and add them to the *ExsSet.pd* file. Rebuild. Fix the bug that occurs by adding an extra pair of parentheses where needed. Rebuild and verify again. There should be no problems.

Just for fun, add an extra property that says that directly after `!add`ing an element `x` **twice in succession** to `self`, function `check` returns `true`. (Note that this fact is a direct consequence of the `!add` postassertion, so does not amount to anything new.) Rebuild and verify again. There should be no problems.

Exercise 11

This exercise has a number of parts, each of which is assessed and is worth the number of marks stated. Each mark is worth 1% of the overall COMP31111 course assessment (20% in total for Exercise 11). Upload your solutions to Blackboard as described below. The submission deadline is the end of the semester.

EX-11-A (4 marks)

Go back to the simple system built in Exercise 5. Assuming you built and ran it successfully, it should verify without problems. Now in a simple aeroplane world like the one you built, planes can refuel, take off, fly, land etc. indefinitely. In this context, "indefinitely" means that whenever one of the schemas that causes the plane model to change state is executed, in the resulting after-state of the model, **there should always be at least one schema that is enabled** (i.e. its precondition is true), and which can therefore be executed next. This exercise is concerned with verifying such a property. The standard way to do this is to construct an invariant (in the `abstract` section) that is the disjunction of the preconditions of all the schemas you wrote. Assuming you do not permit the engine to be switched off in mid-air, there will be some states of the model that are not accessible in any execution, and so the disjunction of the preconditions does not just say "the system is in one of the possible states", but something more precise. Consequently, the disjunction of the preconditions is not true unconditionally, but is, at best, an invariant of all system executions.

Include such an invariant in your system. Rebuild. Assuming all is well, verify. *Perfect Developer* should be able to prove your invariant unproblematically. **Upload your `.pd` file to Blackboard as Assessment EX-11-A.**

Marking Scheme:
- suitable data structure and schemas (2 marks)
- suitable invariant (2 marks)

EX-11-B (4 marks)

An airline sells seats on its flights according to a dynamic pricing policy, which we model in this exercise. Suppose there are two price bands: A and B. Seats are first sold from the band A allocation at a lower price, and when these have all been sold, the remaining seats are sold at the higher band B price. This policy is pursued as long as the number of days to go before the flight exceeds a threshold. When the threshold is breached, i.e. when the departure date is very close, all remaining seats are sold at a discounted price, regardless of their earlier price band, to try to fill the flight. Write a *Perfect* class to model all this. There should be variables to model the total

number of seats, the numbers of seats in each price band, the prices of bands A and B, the discount price, the threshold, the numbers of seats bought and remaining, and the income earned by the flight so far. There should be invariants linking the variables as appropriate. There should be functions to output the current values of the variables, and schemas to set them (bearing in mind the requirements expressed in the invariants). There should be a schema to sell a seat, and a function to report the maximum income that the flight can earn (taking into account how many days are left before departure). **Upload your `.pd` file to Blackboard as Assessment EX-11-B.**

Marking Scheme:
- suitable variables and invariants (1 mark)
- sell a seat schema (1 mark)
- report maximum income schema (1 mark)

EX-11-C (4 marks)

This exercise is about Peterson's Algorithm, a simple algorithm that ensures mutual exclusion. (Consult a suitable textbook if you find you need more background.) There are two processes `p` and `q`, each of which has a *critical region*. We need to ensure that it is never the case that both of `p` and `q` are executing their critical region at the same time. Whether a process is executing its critical region or not is modeled by the process state. If the state is `Sleeping` then the process is not interested in entering its critical regoin. When a process becomes interested in entering its critical region it enters the `Trying` state (it can't enter the critical region directly, since the other process might be in its critical region at that time). When it is safe for the process to enter its critical region, it enters the `Critical` state, and when it has finished its critical region it returns to the `Sleeping` state. Thus the complete behaviour of both processes is cyclic. You should model the algorithm using *Perfect Developer* as follows.

You need an enumerated set containing the `Sleeping`, `Trying`, `Critical` process states. The state of process `p` is held in variable `pcp` and the state of process `q` is held in variable `pcq`. There is a boolean `turn` variable; when it is `true`, process `p` has the right to enter its critical region; when it is `false`, process `q` has the right to enter its critical region. There is also a boolean `flag` variable. Each process negates `flag` when it is in its critical region (it's just something for the processes to do while they're in their critical region — `flag` models the critical resource that can only be manipulated by a single process at a time). The activities of the two processes are modelled using six schemas as follows. Schema `wake_p` is enabled when process `p` is `Sleeping`. It sets process `p`'s state to `Trying` and sets the `turn` variable to `false`. Schema `wake_q` does a similar job for process `q` except that it sets `turn` to `true`. Schema `grant_p` is enabled when `pcp` is `Trying` and *either* `turn` is `true` *or* `pcq` is `Sleeping`. It sets `pcp` to `Critical`. Schema `grant_q` is enabled when `pcq` is `Trying` and *either* `turn` is `false` *or* `pcp` is `Sleeping`. It sets `pcq` to `Critical`. Schema `critical_p` is enabled when `pcp` is `Critical`. It negates `flag`, sets `turn` to `false`, and sets `pcp` to `Sleeping`. Schema `critical_q` does a similar job for process `q` except that it sets `turn` to `true`. Your *Perfect* model should have a class that contains all the above schemas. Ensure it builds correctly.

If your model works as it should, it should always be the case that at most one process is in its critical region at any moment. Write an invariant that says that `pcp = Critical & pcq = Critical` is never true. Attempt to verify it. It will fail. It's a bit too hard for the *Perfect* prover. Think where the most likely places could be that are preventing *Perfect* from succeeding in completing the proof. They are the two `grant` schemas, since those are the only places that set either of `pcp` or `pcq` to `Critical`. Think about the precondition of each of the two `grant` schemas and figure out what it ought to imply if `pcp = Critical & pcq = Critical` is to remain untrue (bearing in mind what the `grant` schema achieves in its postcondition). Add these two implications as two additional invariants. Verify again. If you've got it right, it should all go through now. **Upload your `.pd` file to Blackboard as Assessment EX-11-C.**

Marking Scheme:
- suitable data structures (1 mark)
- appropriate schemas (1 mark)
- correct 'only one process in its critical section' invariant (1 mark)
- appropriate augmented invariant (1 mark)

EX-11-D (4 marks)

A mobile phone network consists of phones, channels, and a notion of nearness of phones to one another. If two phones are near to one another, and there is a free channel, then the two phones can talk to one another, otherwise not. Write a *Perfect* class that models this situation, and includes functions/schemas to cope with the following situations: (a) two phones become near to one another; (b) two nearby phones talk to each other; (c) two talking phones disconnect from each other; (d) two nearby phones become far apart from one another; (e) assuming a phone can act as a relay, two distant phones talk to one another via a third (unspecified) phone which they are both near to.

A starter `.pd` file is provided for you in the *Ex11* folder as *ORIG-Mobile.pd*. This contains all the relevant pieces, but without taking the channels into account. Build the application — it builds successfully. Verify it — it verifies successfully.

Actually, the successful verification is surprising. Consider `getfar`. Although there is a precondition that ensures that the two phones being separated are not in `talk`, nothing prevents `getfar` from separating a phone that acts as a relay in a connection from one of the connected phones. Thus phone `rr` say, that is near to two others which are `talk`ing, `x1` and `x2` say, and where `rr` acts as a relay between them, could get separated from one or other of `x1` or `x2`. In the real world such a thing would have some detrimental effect on the connnection between `x1` and `x2` — yet nothing goes wrong in the *ORIG-Mobile.pd* world since the model only demands that the relay be near the two other phones at the moment the connnection is set up, and does not demand that the relay `rr` remains nearby throughout the connnection. Correct this by writing a new invariant that says that for every pair of phones `{x1,x2}` in `talk`, *either* they are in `near`, *or* there is another phone `rr` that `x1` and `x2` are both `near`. Rebuild. Verify. Now the verification fails. Obviously, the `getfar` schema cannot preserve the new invariant as it stands. Correct the `getfar` schema, by removing (in the postcondition of `getfar`), those connections from `talk` that rely on the `near`ness of the pair being removed which are one part of an indirect connection. Rebuild. Reverify. It proves too hard for the *Perfect* verifier. Given the kind of phenomena we have seen above in carrying a failing proof through, we do not stop to pursue that aspect. **Upload your `.pd` file to Blackboard as Assessment EX-11-D.**

Marking Scheme:
- suitable invariant (2 marks)
- suitable `getfar` schema (2 marks)

EX-11-E (3 marks)

Assuming everything has gone fine so far, it is time to tackle the fact that channels do not figure at any point in the system so far. Now, we remedy this by having each each distinct `talk`ing pair associated with a distinct channel, drawn from a set of available channels. In this exercise you shoud rewrite the system to introduce channels, so that schemas `gettalking`, `stoptalking`, `talkviarelay` all have an additional, `channel` parameter, which refers to the channel relevant to the `talk`ing pair of phones being operated on. Build. Verify — it fails. **Upload your `.pd` file to Blackboard as Assessment EX-11-E.**

Marking Scheme:
- suitable data structures (1 mark)
- appropriate rewriting of all schemas except `getfar` (1 mark)
- appropriate rewriting of `getfar` schema (1 mark)

EX-11-F (1 mark)

In the previous exercise, the channel used for each connection is made explicit in the signature of every schema relevant to the connection. However, in normal use, users are not aware of the channel they are using during a mobile phone conversation. In this exercise rewrite the previous version so that the channel used is determined by the underlying system. So schemas `gettalking`, `stoptalking`, `talkviarelay` no longer have a channel parameter. This means that you must use the various forms of quantification that *Perfect* makes available instead, such as: `forall`, `exists`, `those`, `that`, `any`, `for ... yield`, `for those ... yield`. Build. Now, even building runs into problems — *Perfect* does not generate code for some of these highly nondeternimistic constructs. Verify — it fails, but the verification is not impeded by the build problems. In other words, reasoning about constructs like this is different from generating code to implement them. In practical situations, you would refine the abstract model to a low enough level that code generation was straightforward. **Upload your `.pd` file to Blackboard as Assessment EX-11-F.**

Some Hints on Java and Code Generation

The following hints were generated after some trial and error in the labs.

1) The path to the java tools is a bit different from the document referred to above.because a different java version is installed. You can try "C:\ProgramFiles\Java\jdk1.7.0_03\bin" (or similar, depending on the version).

2) When you open your Perfect project, make sure you enter the path in the file browser, and that it begins with "P:\". This can be done by using the file browser as usual, then clicking the path area to examine the text version of the path.

3) When you run cmd.exe, you will need to do "pushd P:\" to navigate to your home directory.

4) The path to the java runtime binaries will typically not be in the PATH environment variable; you can add it by typing "set PATH=%PATH%;C:\Program Files (x86)\Java\jre7\bin". (Or use the path to your favourite JRE version — but make sure it matches the JDK version above.)