CISC422 /
a2-PrinceLorianZ

Code Issues Pull requests Actions Projects Security Insights

cisc422-fall2024-classroom-assignment-2-a2-starter-model-F24 created by GitHub Classroom

0 stars 95 forks 0 watching Branches Activity Custom properties
Tags

Private repository · Forked from [CISC422/cisc422-fall2024-classroom-assignment-2-a2-starter-model-F24]()

master 1 Branch Tags Go to file t Go to file + Add file Code ...

This branch is up to date with `master` .

Contribute Sync fork

github-classroom[bot] Initial commit 8dbd483 · 3 weeks ago

alloy	Initial commit	3 weeks ago
doc	Initial commit	3 weeks ago
images	Initial commit	3 weeks ago
.gitattributes	Initial commit	3 weeks ago
README.md	Initial commit	3 weeks ago

CISC/CMPE 422: Formal Methods in Software Engineering (Fall 2024)

Assignment 2: Class Modelling with Alloy

Due date: Oct 22, 5pm (GitHub classroom and OnQ submission)

Software

This assignment uses Alloy, an analysis tool for class models developed by the [Software Design Group]() at MIT. Alloy is publicly available for

README

Overview ... You can find pointers to all of this on the [Alloy webpage](). There also is an [online web interface and tutorial](). **Note that some of this material describes Alloy 6, while we will be using Alloy 5.1.0.** The central difference is that Alloy 6 supports mutable signatures and the analysis of traces with respect to temporal operators. Nonetheless, the syntax of the Alloy language is very similar and, overall, the material should still be useful. Alloy 5.1.0 can be downloaded [here](). Installation and invocation are straightforward: Just store the .jar file somewhere and type `java -jar <name>.jar` in a terminal window where `<name>` is the name of the Alloy jar file. Otherwise, Alloy also is installed in [CasLab]().

Learning Outcomes

The purpose of this assignment is to give you

- practical experience with
  - expressing objects and their relationships formally and declaratively using constraints expressed in first-order logic and a relational calculus, and
  - reasoning about objects, their relationships and operations on them using constraint solving, and
- an increased understanding package and repository management.

Context

Many software systems (such as programming language or operating systems) support the notion of a package. A *package* is a piece of software that can be installed into the system to enhance its functionality. Packages may depend on or conflict with other packages. Conflicts may be due to technical (e.g., version incompatibilities) or non-technical (e.g., licence restrictions) reasons. *Package managers* facilitate installation or deinstallation of packages from some repository. A *repository* (sometimes also called *distribution* or *registry*) is typically a curated, internet-accessible collection of packages. The table below shows the most popular package managers and repositories for the Ubuntu Linux distribution and some common programming languages:

Software system	Package manager	Repository	Number of packages
Ubuntu	`apt-get`	`packages.ubuntu.com`	60,000
Python	`pip`	`pypi.org`	500,000
Java	`maven` , `gradle`	`maven.apache.org`	14 million
JavaScript	`npm`	`npmjs.com`	2.1 million
C++	`vcpkg`	`vcpkg.io`	1,500

The management of repositories is challenging. Reasons include:

- The number of packages contained is often very large.
- Many packages are updated frequently updates due to bugs, vulnerabilities, and new features. Each such updates can create a different versions, possibly with 'breaking changes', i.e., changes that may require changes to any code that uses the updated package.
- Packages can be subject to intricate, complex dependencies, conflicts, incompatibilities or restrictions.
- Repo metadata (i.e., relevant information stored in the repository about packages and their relationships) can be incorrect and make parts of the repository inaccessible. E.g., according to [GXY+24] repo info of almost 30% of PyPI releases cannot be retrieved due to invalid metadata.

This assignment is an introduction to the challenges of package management in general and repo management in particular. It uses class modeling in Alloy to explore some of the problems managers of repositories face and to provide answers to questions such as: Is it possible that a repo contains packages that cannot be installed? If so, under what circumstances and how could this be fixed? Is it possible that a repo becomes useless because none of its packages are installable? What exactly does it mean for two packages to be in conflict and what are the properties of this conflict relationship? How can the addition and removal of packages impact conflicts?

For additional information on package and repository management (optional), please see:

- [Spi12] Spinellis. Package Management Systems. IEEE Software 29(2):84-86. 2012. https://ieeexplore.ieee.org/document/6155145
- [GXY+24] Gao, Xu, Yang, Zhou. PyRadar: Towards Automatically Retrieving and Validating Source Code Repository Information for PyPI Packages. FSE'24. 2024. https://arxiv.org/abs/2404.16565
- [XHG+23] Xu, He, Gao, Zhou. Understanding and Remediating Open-Source License Incompatibilities in the PyPI Ecosystem. ASE'23. 2023. https://arxiv.org/abs/2308.05942
- [MBD+06] Mancinelli, Boender, Di Cosmo, Vouillon, Durak, Leroy, Treinen. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. ASE'06. 2006. https://ieeexplore.ieee.org/document/4019575
- [JYZ+23] Jing, Yu, Zhang, Meng, Liu, Xue. Classifying Packages for Building Linux Distributions. COMPSAC'23. 2023. https://ieeexplore.ieee.org/document/10197038
- [LDM20] Legay, Decan, Mens. On Package Freshness in Linux Distributions. ICSME'20. 2020. https://arxiv.org/abs/2007.16123

Questions

File to Edit

Please enter all your answers into the given file alloy/repoMgmnt_starterModel.als in the indicated places.

Part I: Structure [64/128 points]

Preparation

We start with the class model below:

```
module RepoMgmnt
sig Pkg {}
sig Repo {
   pkgs: set Pkg,            // packages contained in the repo
   imp: Pkg -> Pkg,          // repo metadata recording which package imports which other packages
   confl: Pkg -> Pkg         // repo metadata recording which package conflicts with which other packages
```

```
}
fact MetaDataValidity {
   all r:Repo | all p:Pkg | p !in p.(r.confl)                      // "no package conflicts with itself" (MDV1)
   // no r:Repo | no p:Pkg | p in p.(r.confl)                          // equivalent to above using De Morgan
   // all r:Repo | all p:Pkg | p->p !in r.confl                        // equivalent to above using tuple notation
   // all r:Repo | all p:Pkg | p !in conflicts[p,r]                    // equivalent to above using helper function
   all r:Repo | all p:Pkg | p !in p.(r.imp)                        // "no package imports itself" (MDV2)
   // all r:Repo | all p:Pkg | p !in imports[p,r]                      // equivalent to above using helper function
   all r:Repo | all p,q:Pkg | q in p.(r.confl) => p in q.(r.confl)       // "conflicts are symmetric" (MDV3)
   // all r:Repo | all p,q:Pkg | q in conflicts[p,r] => p in conflicts[q,r]  // equivalent using helper function
   // all r:Repo | r.confl = ~(r.confl)                                // equivalent to above using relational inverse
}
fun conflicts[p:Pkg,r:Repo] : set Pkg {                        // "the set of packages that 'p' conflicts with in 'r'"
   p.(r.confl)
}
fun imports[p:Pkg,r:Repo] : set Pkg {                          // "the set of packages that 'p' imports in 'r'"
   p.(r.imp)
}
pred empty[r:Repo] {              // "a repo is empty iff it contains no packages"
   no r.pkgs
}
pred equal[r1,r2:Repo] {          // "two repos are equal iff they have the same packages and metadata"
   r1.pkgs = r2.pkgs
   r1.imp = r2.imp
   r1.confl = r2.confl
}
```

Elements of signatures `Pkg` and `Repo` will be called *packages* and *repositories* (or just *repos*), respectively. The model expresses that a repo `r` has three attributes `pkgs`, `imp`, and `confl` where `r.pkgs` is a possibly empty set of packages and `r.imp` and `r.confl` both are binary relations over packages. Given a repo `r`, `r.pkgs` represents the set of packages contained in `r`. I.e., we say that package `p` *is in* `r` if and only if (iff) atomic formula `p in r.pkgs` holds. The relations `r.imp` and `r.confl` represent metadata in `r`, keeping track of imports and conflicts, respectively. I.e., given two packages `p` and `q`, if the pair `p->q` is contained `r.imp` (i.e., formula `p->q in r.imp` holds or, equivalently, `q in p.(r.imp)` holds), then according to `r`'s metadata, package `p` imports package `q`. Similarly, if the pair `p->q` is contained `r.confl` (i.e., `p->q in r.confl` or, equivalently, `q in p.(r.confl)` holds), then according to `r`'s metadata, package `p` conflicts with package `q`.

The purpose of the constraints in fact `MetaDataValidity` is to impose some 'common sense' restrictions on this metadata, i.e., ensure that this metadata has properties that we would expect it to have. For instance, the first constraint `MDV1` prevents packages to conflict with themselves. I.e., given any repo `r` and any package `p`, without `MDV1` there is nothing in the model that prevents the pair `p->p` from being an element in `r.confl`. Similarly, `MDV2` says that a package cannot import itself. Finally, `MDV3` expresses that the relation `r.confl` is *symmetric*, i.e., that if a (any) package `p` conflicts with package `q` in some (any) repo `r`, then `q` also conflicts `p` in `r`.

When using 3-place relations such as `confl` and `imp`, we must be careful about the order in which relational composition is performed. E.g., `p.(r.confl)` represents the set of packages that package `p` conflicts with in repo `r`, while `(p.r).confl` (which can be abbreviated to `p.r.confl` since the relational composition operator `.` associates to the left) is not type-correct, because `p` and `r` cannot be composed. We can make 3-place relations easier to work with using helper functions. E.g., function `conflicts[p:Pkg,r:Repo]:set Pkg` returns the set of packages that `p` conflicts with in `r`, allowing metadata constraint `MDV1` can be re-written as

```
   all r:Repo | all p:Pkg | p !in p.(r.confl)                  // "no package conflicts with itself" (MDV1)
//   all r:Repo | all p:Pkg | p !in conflicts[p,r]                // equivalent to above
```

Similarly for `MDV2` and helper function `imports[p:Pkg,r:Repo]:set Pkg`.

Predicate `empty[r:Repo]` holds for `r` if and only if (iff) `r` does not contain any packages. Predicate `equal[r1,r2:Repo]` captures what it means for two repos to be equal. They will be used in Question 2.

One of the challenges with Alloy (and most other languages) is that the same thing can often be expressed in many different ways. The examples of equivalent versions given for constraints `MDV1`, `MDV2` and `MDV3` illustrate that. For this assignment, as long as your formalization correctly captures the semantics of the property or constraint to be formalized you should be getting full marks, i.e., as long as your formalization is semantically equivalent to the formalization that we expect, the way the formalization is expressed does not matter.

Note that some of these constraints can also be captured in a UML class diagram.

Finding satisfying instances (scenarios): We can explore the instances satisfying (the constraints in) a model using Alloy's `run` command:

```
run P1a {} for 3                                                   // satisfiable
run P1b {#Pkg=2 && some r1,r2:Repo | equal[r1,r2]} for 3          // satisfiable
run P1c {some r:Repo | some p,q:Pkg | q in conflicts[p,r] && !(p in conflicts[q,r])} for 3   // not satisfiable
run P1d {#Repo>=4} for 3                                           // not satisfiable
run P1e {some r:Repo | r.pkgs = Pkg} for 3                         // satisfiable
```

Command `P1a` asks the analyzer to find instances that satisfy the constraints in the model in scope 3 (i.e., instances can contain at most 3 elements of each signature). Command `P1b` does the same, except that it additionally requires satisfying instances to also satisfy the constraint `#Pkg=2 && some r1,r2:Repo | equal[r1,r2]`, i.e., contain exactly 2 packages and two repos that are equal. Command `P1c` fails to find any instances because the additional constraint that is the argument of the command contradicts `MDV3`. Similarly for command `P1d`, where the scope constraint of `3` prevents the generation of instances with 4 or more signature elements. Command `P1e` asks for instances that satisfy all the constraints in the model, and also contain at least one repo `r` such that `r` contains all packages.

Inspecting the generated instances for `P1a`, `P1b`, and `P1e`, we can see that the model above is currently does not fully capture all expected metadata constraints, i.e., the import and conflict relationships don't exhibit all the expected properties. For instance, the import and conflict metadata for some repo can actually refer to packages that are not contained in that repo. We will return to this observation later in Question `Q1c`.

Checking assertions: We can check if all instances satisfying the constraints in the model have some property `P` by using the `check {P}` command. Consider the following examples:

```
check P1f {all r:Repo | no p:Pkg | p in imports[p,r]} for 3       // holds
check P1g {all r:Repo | some r.pkgs} for 3                        // does not hold
check P1h {Repo.pkgs in Pkg} for 3                                // holds
check P1i {all r:Repo | r.pkgs = Pkg} for 2 but exactly 1 Repo    // does not hold
```

Command `P1f` checks if the constraints in the model are strong enough to cause every satisfying instance to also satisfy the formula `all r:Repo | no p:Pkg | p in imports[p,r]` (which captures the statement that "*for all repos, there is no package that imports itself in that repo*"). Command `P1g` checks if all satisfying instances of the model contain at least one package. Command `P1h` checks if in all satisfying instances, the packages contained in all repos are all elements of the `Pkg` signature. Command `P1i` checks if in all satisfying instances, all repos contain all available packages. The first check (`P1f`) succeeds, i.e., Alloy does not find a counter example for the assertion in the given scope, suggesting that the constraints in the model indeed imply the assertion. The second check (`P1g`) fails, because the constraints in the model do not force every repo to contain at least one package.

The fourth check (`P1i`) fails, because there are satisfying instances in which at least one repo does not contain all packages. One such instance is shown here together with the evaluator window showing what different expressions evaluate to in this instance. While the default visualization (which shows all attributes as arcs) is fine for small instances and simple models, it quickly becomes hard to read for large models and models attributes ranging over binary relations (such as `imp` and `confl`). To mitigate that, the customization of the visualization is highly recommended. Please go here for more information.

Question 1: Metadata validity [15 points]

- For each of the following two properties, complete the corresponding `run` commands to generate scenarios that satisfy the properties. Use the Alloy analyzer to enumerate the first few scenarios and check that they look as expected.
  - **Q1a [3 pts]:** "*There exist a repo that is empty, a repo that is non-empty and conflict-free, and a repo that is non-empty and not conflict-free*".
  - **Q1b [3 pts]:** "*There exists a repo that contains 3 packages and at least one of them imports every other package in the repo*".
- Using provided predicates `empty` and `equal`, complete assertion `Q1c` so that it captures the meaning of the statement below. Use the Alloy analyzer to see if the assertion holds.
  - **Q1c [3 pts]:** *Empty repos are equal*, i.e., "*For all repos* `r1` *and* `r2`*, if* `r1` *and* `r2` *are both empty, then they are equal*".
- The reason assertion `Q1c` fails is that the metadata for imports and conflicts can, as already observed above, mention packages that are not actually contained in the repository. For each of the following statements below, formalize them in Alloy and add them to the fact `MetaDataValidity`.
  - **Q1d [3 pts]:** "*For all repos* `r`*, the import metadata* `r.imp` *does not mention any packages that are not contained in* `r`", i.e., "*For all repos* `r` *and all packages* `p` *and* `q`*, whenever* `p` *imports* `q` *in* `r`*, then both* `p` *and* `q` *are contained in* `r`".
  - **Q1e [3 pts]:** "*For all repos* `r`*, the conflict metadata* `r.confl` *does not mention any packages that are not contained in* `r`", i.e., "*For all repos* `r` *and all packages* `p` *and* `q`*, whenever* `p` *conflicts with* `q` *in* `r`*, then both* `p` *and* `q` *are contained in* `r`".

  These additional metadata validity constraints rule out the counter examples that made assertion `Q1c` fail. Rerun the assertion check `Q1c` (for increasing scopes till 6) and convince yourself that the assertion now holds.

Question 2: Subrepos [12 points]

- Given two repositories `r1` and `r2`, we say that `r2` *is a sub-repository of* `r1` if and only if (iff)

  i. the packages contained in `r2` are also contained in `r1`,

  ii. package `p1` imports `p2` in `r2` iff (`p1` imports `p2` in `r1` and `p1` and `p2` are contained in `r2`), and

  iii. package `p1` conflicts with `p2` in `r2` iff (`p1` conflicts with `p2` in `r1` and `p1` and `p2` are contained in `r2`).

  - **Q2a [3 pts]:** Complete the predicate `subRepo[r2,r1]` such that it holds precisely when `r2` is a sub-repository of `r1` as defined above. Use the run command `Q2a` to generate some instances and convince yourself that they look as expected.

- For each of the following statements, complete the corresponding assertion such that it captures the meaning of the statement.
  - **Q2b [3 pts]:** "*A (i.e., any) repo is empty iff all its subrepos are empty*".
  - **Q2c [3 pts]:** "*For any two repos* `r1` *and* `r2` *, if* `r1` *is a subrepo of* `r2` *and* `r2` *is subrepo of* `r1` *, then* `r1` *and* `r2` *are equal*". This property is usually called *anti-symmetry*.
  - **Q2d [3 pts]:** "*For any three repos* `r1` *,* `r2` *, and* `r3` *, if* `r1` *is a subrepo of* `r2` *and* `r2` *is subrepo of* `r3` *, then* `r1` *is also a subrepo of* `r3` ". This property is usually called *transitivity*.

  Use the Alloy analyzer to check that all three assertions above hold (in increasing scopes up to 6).

Question 3: Installability and co-installability [31 points]

- The metadata information for a repo is not quite complete. We also need the notion of dependency. Given two packages `p1` and `p2` , we say that `p1` *depends on* `p2` iff

  i. `p1` imports `p2` , or
  ii. `p1` imports some package that imports `p2` , or
  iii. `p1` imports some package that imports some package that imports `p2` , or
  iv. ...

  I.e., there is a non-empty, finite sequence of imports relationships (i.e., edges) from `p1` to `p2` .

- **Q3a [3 pts]:** Remove the comment at the beginning of the line:

```
//    dep: Pkg -> Pkg          // uncomment for Question 3
```

  Then, add a constraint to the fact `MetaDataValidity` in the indicated place such that for all repos `r` , the binary relation `r.dep` captures the notion of dependency defined above. I.e., given two packages `p1` and `p2` , `p2 in p1.(r.dep)` should hold iff `p1 depends on p2` in the sense above and both are contained in `r` . Use the `run` command (with, e.g., `run {some r:Repo | some r.dep}`) to convince yourself that your constraint on `dep` works as expected.

- Given a package `p` and a repo `r` , we say that `p` *is (individually) installable from* `r` iff

  i. `p` is contained in `r` ,
  ii. none of the packages that `p` depends on (according to `r` 's metadata) conflict with `p` , and
  iii. none of the packages that `p` depends on (according to `r` 's metadata) conflict with each other.

  Q3b [4 pts]: Complete the predicate `installable[p:Pkg,r:Repo]` such that it holds precisely when `p` is installable from `r` as defined above. Use the `run` command to generate some relevant instances and convince yourself that your formalization of installability works as expected.

- For each of the following statements, complete the corresponding assertion with a formula that captures the meaning of the statement.
  - **Q3c [3 pts]:** "*For all repos* `r` *, if* `r` *has an uninstallable package, then* `r` *has conflicts (i.e.,* `r` *is not conflict-free)*".
  - **Q3d [3 pts]:** "*For all repos* `r` *, if* `r` *has conflicts, then there exists at least one package that is not installable from* `r` ".
  - **Q3e [2 pts]:** Use the Alloy analyzer to determine which of the two assertions above hold and which do not. Check the assertions in increasing scopes up to 8. Hint: Exactly one of the two assertions should hold.

- Given a set of packages `P` and a repo `r` , we say that `P` *is co-installable* (or *jointly installable*) *from* `r` iff

  i. all packages in `P` are contained in `r` ,
  ii. none of the packages in `P` conflict with each other,
  iii. none of the packages that some package in `P` depends on do not conflict with a package in `P` , and
  iv. none of the packages that any of the packages in `P` depend on conflict with each other.

  Q3f [4 pts]: Complete the predicate `coInstallable[P:set Pkg,r:Repo]` such that it holds precisely when `P` is co-installable from `r` as defined above. If you want, you can add a helper function `dependsOn[p:Pkg,r:Repo]:set Pkg` to make working with the relation `dep` easier (just like we did for `confl` and `imp` above). Use the `run` command to generate some relevant instances and convince yourself that your formalization of co-installability works as expected.

- For each of the following statements, complete the corresponding assertion with a formula that captures the meaning of the statement.
  - **Q3g [3 pts]:** "*For all repos* `r` *, if all packages contained in* `r` *are (individually) installable from* `r` *, then the set of packages contained in* `r` *is co-installable from* `r` ".
  - **Q3h [3 pts]:** "*For all repos* `r` *, if the set of packages contained in* `r` *are co-installable from* `r` *, then each of the packages in* `r` *is individually installable from* `r` ".
  - **Q3i [3 pts]:** "*For all repos* `r` *, the set of packages contained in* `r` *is co-installable from* `r` *iff* `r` *is conflict-free*".

- **Q3j [3 pts]:** Use the Alloy analyzer to determine which of the three assertions above hold and which do not. Check the assertions in increasing scopes up to 8. Hint: Exactly two of the three assertions should hold.

Question 4: Trimness [6 points]

- **Q4a [3 pts]:** We say that a repo `r` *is trim* iff the set of packages it contains is co-installable from `r`. Complete the predicate `trim[r]` such that it holds precisely when `r` is trim as defined above. Intuitively, a trim repo is maximally useful, because any subset of its packages is installable.
- **Q4b [3 pts]:** Complete the assertion `Q4b` with a formula that captures the meaning of the following statement: "*All subrepos of a trim repository are also trim*". Use the Alloy analyzer to check that the assertion holds in scopes up to 8.

Part II: Operations [39/128 points]

Question 5: Trimifying a repo [11 points]

- As we have seen, the non-trimness of some repo is due to conflicting dependencies. In the worst case, most or all of the packages in the repo cannot be accessed. This can be due to the dependencies and conflicts that packages really have, or it could be due to erroneous metadata (as observed in [GXY+24]). We now consider an operation `trimfy[r1,r2:Repo]` that, intuitively, will produce a trim repo (`r2`) from a possibly non-trim one (`r1`) by removing packages. To maximize the utility of the resulting repo, `trimify` should minimize the number of packages removed.

  - **Q5a [3 pts]:** Complete the predicate `trimify[r1,r2:Repo]` such that for all repos `r1` and `r2` `trimify[r1,r2]` holds iff
    a. `r2` is a subrepo of `r1` ,
    b. `r2` is trim, and
    c. `r2` is the largest trim subrepo of `r1` , i.e., there is no trim subrepo of `r1` that contains more packages than `r2` .

  Use the `run` command with different argument constraints (e.g., `some r1,r2:Repo | !trim[r1] && trimify[r1,r2]`) to convince yourself that `trimify` works correctly.

- What kind of operation is `trimify` ? Which properties do we expect it to have, apart from the three properties above that make up its definition? Below, we will consider two properties: idempotency and uniqueness. For each of the two statements below, complete the corresponding assertion such that it captures the meaning of that statement.

  - **Q5b [3 pts]:** Idempotency: *Trimifying a trim repo leaves that repo unchanged*, i.e., "*for all repos* `r1` *and* `r2` *, if* `r1` *is trim and* `r2` *is the result of trimifying* `r1` *, then* `r1` *and* `r2` *are equal*".
  - **Q5c [3 pts]:** Uniqueness: *The result of* `trimify` *is uniquely determined*, i.e., "*for all repos* `r1` *,* `r2` *, and* `r3` *, if* `r2` *and* `r3` *are both the result of trimifying* `r1` *, then* `r2` *and* `r3` *are equal*".
  - **Q5d [2 pts]:** Use the Alloy analyzer to determine which of the two assertions above hold and which do not. Check the assertions in increasing scopes up to 8. Hint: Exactly one of the two assertions should hold.

Question 6: Removing a package [13 points]

- The next operation we consider is the removal of a package from a repo. We want this removal to be sensitive to existing dependencies and metadata validity constraints.

- **Q6a [4 pts]:** Complete the predicate `remPkg[r1:Repo,p:Pkg,r2:Repo]` such that for all repos `r1` and `r2` and packages `p` , `remPkg[r1,p,r2]` holds iff
  i. `p` is contained in `r1` ,
  ii. no package in `r1` imports `p` ,
  iii. `r2` contains exactly all packages that `r1` contains minus `p` ,
  iv. the import metadata of `r2` coincides with that of `r1` except that all imports involving `p` have been removed, and
  v. the conflict metadata of `r2` coincides with that of `r1` except that all conflicts involving `p` have been removed.

  Use the `run` command with different argument constraints (e.g., `some r1,r2:Repo | some p:Pkg | remPkg[r1,p,r2]`) to convince yourself that `remPkg` works correctly.

- Which emerging properties does `remPkg` have? We will consider three. For each of the two statements below, complete the corresponding assertion such that it captures the meaning of that statement.

  - **Q6b[3 pts]:** *Removal creates a subrepo*, i.e., "*for all repos* `r1` *and* `r2` *and packages* `p` *, if* `r2` *is the result of removing* `p` *from* `r1` *, then* `r2` *is a subrepo of* `r1` ".
  - **Q6c [3 pts]:** *Removal preserves trimness*, i.e., "*for all repos* `r1` *and* `r2` *and packages* `p` *, if* `r1` *is trim and* `r2` *is the result of removing* `p` *from* `r1` *, then* `r2` *is trim*".
  - **Q6d [3 pts]:** "*If removing a package makes a repo trim, then the repo has conflicts and all of these conflicts involve the package*", i.e., *for all repos* `r1` *and* `r2` *and packages* `p` *, if* `r1` *is not trim and* `r2` *is the result of removing* `p` *from* `r1` *and* `r2` *is trim, then* `r1` *is not conflict-free and all its conflicts involve* `p` .

- **Q6e [0 pts]:** Use the Alloy analyzer to determine which of the three assertions above hold and which do not. Check the assertions in increasing scopes up to 8. Hint: All assertions should hold.

Question 7: Adding a package [15 points]

- The next operation we consider is the addition of a package to a repo. If we use standard, implementation-level thinking, we would expect this add operation to carry 5 arguments: the repo `r1` to which the package is to be added, the package `p` to be added, a list of packages in `r1` that `p` imports, a list of packages in `r1` that `p` conflicts with, and the resulting repo `r2`. However, we will omit the third and fourth arguments, that is, our operation will only have three arguments: `addPkg[r1:Repo,p:Pkg,r2:Repo]`. How can this work? Which imports and conflicts will `p` have in `r2`? The point is that the result of the addition does not need to be uniquely determined, i.e., for specific `r1` and `p`, there can be *several* result repos `r2` such that `addPkg[r1,p,r2]` holds. In other words, we will leave the addition *underspecified* and as a consequence, the addition turns into a *non-deterministic* operation that can have different results. If `addPkg[r1,p,r2]` holds, we know that `r2` contains `p` but we don't know if `p` has any imports or conflicts in `r2`, and, if so, what exactly they are. Why do this? Because it is enough for our purposes and a predicate with 3 arguments is easier to work with than one with 5 arguments.

  - **Q7a [3 pts]:** Complete the predicate `addPkg[r1:Repo,p:Pkg,r2:Repo]` such that for all repos `r1` and `r2` and packages `p`, `addPkg[r1,p,r2]` holds iff
    a. `p` is not contained in `r1`,
    b. `p` is contained in `r2`, and
    c. the removal of `p` from `r2` results in `r1`.

  Use the `run` command with different argument constraints (e.g., `some r1,r2:Repo | some p:Pkg | addPkg[r1,p,r2]` or `some r1,r2:Repo | some p:Pkg | trim[r1] && addPkg[r1,p,r2] && !trim[r2])`) to convince yourself that `addPkg` works correctly.

- Which emerging properties does `addPkg` have? We will consider three. For each of the three statements below, complete the corresponding assertion such that it captures the meaning of that statement.

  - **Q7b[3 pts]:** *Add and remove are inverses*: Requirement 3 in `Q7a` means that `addPkg` and `remPkg` should be *inverses* of each other, i.e., *"for all repos `r1` and `r2` and packages `p`, `r2` is the result of adding `p` to `r1` iff `r1` is the result of removing `p` from `r2`"*.
  - **Q7c [3 pts]:** *Unique results*, i.e., *"for all repos `r1`, `r2`, and `r3` and packages `p`, if `r2` is the result of adding `p` to `r1` and `r3` is the result of adding `p` to `r1`, then `r2` and `r3` are equal"*.
  - **Q7d [3 pts]:** *"Addition preserves conflicts"*, i.e., *"for all repos `r1` and `r2` and packages `p`, if `r2` is the result of adding `p` to `r1`, then all conflicts in `r1` also are conflicts in `r2` and all conflicts in `r2` not contained in `r1` involve `p`"*.
  - **Q7e [3 pts]:** Use the Alloy analyzer to determine which of the three assertions above hold and which do not. Check the assertions in increasing scopes up to 8. Hint: Exactly two of the three assertions should hold.

Part III: Executions [25/128 points]

We will now explore how sequences of operations can be formalized in Alloy and how this formalization can be used to execute these sequences, check properties of executions, and generate test inputs.

- **Preparation:** Remove the comment at the beginning of line

```
// open util/ordering[Repo]   // uncomment for Question 8
```

  to impose the constraints of the `ordering` module on elements of signature `Repo`, i.e., repositories. In every scenario found by Alloy, these constraints force the elements of `Repo` in the scenario to be *linearly (totally) ordered*. That is, for any two repos in the scenario, they will either be the same, or one will come before the other in the ordering. Also, a repo is either the last (first) in the ordering, or it has exactly one successor (predecessor). The elements of `Repo` in a scenario (if any) will be called `Repo0`, `Repo1`, `Repo2`, ... (or, `Repo$0`, `Repo$1`, ... in Alloy's evaluator) with `Repo0` always being the first element in the ordering and `Repo1` being its successor. Also, the `ordering` module contains several useful functions (e.g., `first`, `last`, and `next`) and predicates (`lt`, `lte`, `gt`, and `gte`). E.g., function `first` (from module `ordering`) will evaluate to the first element in the scenario (i.e., `Repo0`), if any. In contrast, function `last` denotes the last element, if any. `next` is the successor relation. E.g., assuming a scenario in which `Repo0` and `Repo1` exist, the expression `Repo0.next` will evaluate to `Repo1` and `next.Repo1` (and `Repo1.prev`) will evaluate to `Repo0`. Given a scenario, use the evaluator to experiment with these functions and predicates, and see models/util/ordering.als (from Alloy's .jar file, but included in the directory `alloy/` of this assignment's repository for convenient reference) for more information.

Question 8: Generating executions [16 points]

To model executions, we need to express that, in all instances, the successor of a repo `r` is the result of applying `remPkg` or `addPkg` on `r` (we will not use `trimify` in this part). To achieve this, we will use predicate `legalExec[]`.

- **Q8a [3 pts]:** Complete the predicate `legalExec[]` such that it holds iff *for all repos `r1`, unless `r1` is last in the ordering, there exists a package `p` such that the successor of `r1` in the ordering is the result of either removing `p` from `r1` or adding `p` to `r1`*. Use the `run` command with different argument constraints (e.g., `legalExec[]` or `empty[first] && legalExec[]`) to generate different executions and convince yourself that they look as expected.

- **Q8b [3 pts]:** Complete the predicate `complex[r:Repo,n:Int]` such that it holds iff *repo* `r` *contains at least one package that depends on* `n` *different packages*. Use the `run` command with different argument constraints to generate different repos that satisfy `complex` and convince yourself that they look as expected.
- For each of the three statements below, complete the corresponding `run` command such that it generates executions described in the statement.
  - **Q8c [3 pts]:** "*There is a legal execution along which addition and removal alternate*".
  - **Q8d [3 pts]:** "*There is a legal execution along there exists exactly one package that is added but never removed*".
  - **Q8e [3 pts]:** "*There is a legal execution along which there is a package that is removed at least 3 times*".

  Use these `run` commands to convince yourself that they work as expected.
- **Q8f [1 pts]:** What is the smallest scope in which your version of `run Q8e {...}` finds an instance? I.e., what is the smallest scope in which there is an instance that satisfies all constraints of the model and the additional constraint `Q8e` ?

Question 9: Checking properties of executions [9 points]

We now can check properties of executions. For each of the statements below, complete the corresponding assertion with a formula that captures that statement.

- **Q9a [3 pts]:** "*Along all legal executions starting from an empty repo, a (i.e., any) repo is either empty or it contains at least one individually installable package*".
- **Q9b [3 pts]:** "*Along all legal executions, whenever a (i.e., any) package is removed twice, it has been added in between*".
- **Q9c [3 pts]:** "*Every legal execution starting from an empty repo is such that the size of the last repo is the number of additions in the execution minus the number of removals*". Use the built-in function `minus[m,n:Int]:Int`.
- **Q9d [0 pts]:** Use the Alloy analyzer to determine which of the three assertions above hold and which do not. Check the assertions in increasing scopes up to 8. Hint: All assertions should hold.

Part IV: Discussion [0 points]

We see how constraint solving (with Alloy) can be used to describe relationships between objects and the possible impact of operations on them. Note how this requires a description of the relevant properties of the operations, rather than a description of their implementation. I.e., the executions were described declaratively, rather than operationally. Finding executions (i.e., executing) was achieved through constraint (satisfiability) solving. Note that descriptions of executions need not specify the initial state or which operation was applied when. As result, our formalization of executions can be used to, e.g.,

1. find test inputs that cause executions with certain properties (e.g., "*find initial repos such that the use of exactly three removals results in a trim repo*"),
2. check properties (assertions) that all executions are expected to have, and
3. find counter examples, i.e., executions that cause some property to fail.

Some test generation tools use declarative modeling and constraint solving.

Instructions

Important: Please follow the instructions below carefully. Points may be taken off, if you don't.

- Only edit the file alloy/repoMgmnt_starterModel.als in folder `alloy` . For each of the three parts, answers to questions should only go into this file and into the indicated location.