A software micromodel

The intended benefits provided by a system model are

1. it captures formally static and dynamic system structure and behaviour;

2. it can verify consistency of the constrained design space;

3. it is executable, so it allows guided simulations through a potentially very complex design space; and

4. it can boost our confidence into the correctness of claims about static and dynamic aspects of all its compliant implementations.

Moreover, formal models attached to software products can be seen as a reliability contract; a promise that the software implements the structure and behaviour of the model and is expected to meet all of the assertions certified therein. (However, this may not be very useful for extremely under-specified models.)

We will model a software package dependency system. This system is used when software packages are installed or upgraded. The system checks to see if prerequisites in the form of libraries or other packages are present. The requirements on a software package dependency system are not straightforward. As most computer users know, the upgrading process can go wrong in various ways. For example, upgrading a package can involve replacing shared libraries with newer versions. But other packages which rely on the older versions of the shared libraries may then cease to work.

Software package dependency systems are used in several computer systems, such as Red Hat Linux, .NET's Global Assembly Cache and others. Users often have to guess how technical questions get resolved within the dependency system. To the best of our knowledge, there is no publicly available formal and executable model of any particular dependency system to which application programmers could turn if they had such non-trivial technical questions about its inner workings.

In our model, applications are built out of components. Components offer services to other components. A service can be a number of things. Typically, a service is a method (a modular piece of program code), a field entry, or a type – e.g. the type of a class in an object-oriented programming language. Components typically require the import of services from other components. Technically speaking, such import services resolve all un-resolved references within that component, making the component linkable. A component also has a name and may have a special service, called 'main.'

We model components as a signature in Alloy:

```
sig Component {
name: Name, -- name of the component
main: option Service, -- component may have a 'main' service
export: set Service, -- services the component exports
import: set Service, -- services the component imports
version: Number -- version number of the component
}{ no import & export }
```

The signatures Service and Name won’t require any composite structure for our modelling purposes. The signature Number will get an ordering later on. A component is an instance of Component and therefore has a name, a set of services export it offers to other components, and a set import of services it needs to import from other components. Last but not least, a component has a version number. Observe the role of the modifiers set and option above.
A declaration i : set S means that i is a subset of set S; but a declaration i : option S means that i is a subset of S with at most one element. Thus, option enables us to model an element that may (non-empty, singleton set) or may not (empty set) be present; a very useful ability indeed.
Finally, a declaration i:S states that i is a subset of S containing exactly one element; this really specifies a scalar/element of type S since Alloy identifies elements a with sets {a}.
We can constrain all instances of a signature with C by adding { C } to its signature declaration. We did this for the signature Component, where C is the constraint no import & export, stating that, in all components, the intersection (&) of import and export is empty (no).