Declarative Languages

A declarative configuration language is used to describe the components of a system and their relationships. The resulting configuration can usually be represented as a set of attribute-value pairs, possibly with some hierarchical structure, and possibly some ordering. A data format such as JSON serves as a typical generic representation:

```json
fig: {
  head: {
    face: "male"
    hair: {
      style: "short"
      colour: "brown"
    }
    hat: none
  }
  clothing: { ... }
}
```

The purpose of the configuration language is to allow complex configurations to be described in higher-level terms. This typically involves features for specifying replication, specialisation and composition of data objects, and ways of grouping these into higher-level structures. A declarative configuration does not describe a computation and this makes these languages quite different from programming languages.

A declarative configuration tool usually consists of a compiler and a deployment engine. The compiler translates the higher-level language into a low-level, concrete representation such as the one above. The deployment engine takes the resulting description and performs the necessary operations in the physical world to make the target domain conform to the required specification.

Practical examples of broadly declarative tools include Puppet [18][11], SmartFrog [19][7], and LCFG [17][1]. Other types of configuration tools may use a completely different model – for example, specifying conditions and actions for changing configurations, without an explicit declarative representation (e.g. Ponder [6], or Chef [15][12]), or combining the specification and deployment in a way which makes it very hard to reason about the semantics of the language (e.g. cfengine [14][13]).

Declarative tools have a number of benefits, including the ability to formalise the language semantics in a way which relates directly to the resulting configuration (see [4, 3]). The following discussions assume a tool of this type, and are concerned only with the language, and not the deployment.

1.1 Replication & Specialisation

Replicas of a component can usually be created by cloning a template component. This makes it easy to configure large numbers of identical components, and to specify the common characteristics in one place:

```plaintext
bob: $fig, dave: $fig, frank: $fig
```

Of course, it is rare to require copies which are exactly the same – they will usually vary in some small way, and specialisation is a common operation. This allows a replica of some generic component to be modified to create a particular variant:

```plaintext
female: $fig +> {
  head: {
    face: "female"
    hair: {
      style: "long"
    }
  }
}
```

The notation used here for specialisation (+>) is not standard – it has been chosen to minimise confusion with similar operations in other languages. Likewise, configuration languages often use the term inheritance instead of specialisation – however, specialisation is a form of prototype inheritance [5] (sometimes called instance inheritance). This is used in some programming languages (Javascript, IO, Self) but it is different from the type inheritance common in programming languages such as Java. We use the different terminol-

---

1See [16] for some example configurations in YAML as used by Google’s cloud Deployment manager.

2Comparing this with page layout languages such as HTML/CSS or LATEX may be a useful analogy.
ogy to emphasise this distinction.

## 1.2 Composition

Specialisation allows different components to be created from the same template, without the need to specify the common attributes in each case:

```
fireperson:
$fig +> {
  head: {
    hat: "firehat"
  }
  clothing: {
    top: "firetop"
    bottom: "redbottom"
  }
}
```

Components often represent different aspects of a system and it very common to want to compose these:

```
carol: $fig +> $femalehead +> $fireuniform
```

The components themselves cannot usefully be composed using the specialisation operation, because the attributes are fully-specified and the attributes of the rightmost component will take precedence:

```
carol: $fig +> $fireuniform +> $femalehead
```

So, it is common to define disjoint components with a subset of the attributes:

```
carol: $fig +> $female +> $fireuniform
```

If these components are independent, then the composition will be commutative and the order\(^4\) will not be significant:

```
carol: $fig +> $fireuniform +> $femalehead
```

However, if the regulations for our firefighters specify that “Hair will not extend beyond the bottom of the earlobe”\(^5\), then we might incorporate this into the above model by adding `hair:"short"` explicitly to the definition of the fireuniform:

```
carol: $fig +> $female +> $fireuniform
```

But the components now contain conflicting values for one of the attributes and the simple override semantics for composition is no longer commutative:

```
carol: $fig +> $fireuniform +> $female
```

In practice, many configuration aspects overlap in this way, and this can lead to very brittle configurations. For example: if the components represent the configurations for a web server and a database server, and the user is attempting to configure a machine which acts as both. In this case, the user may not have written either of the components, and is not in a position to understand which parameters may conflict and how they should be composed. In many cases, there may be no

---

\(^3\)A practical example may involve configuring a single machine to act as both a web server, and a database server.

\(^4\)Note that “order” in this case refers to the evaluation semantics of the language – it has no connection with any order in which changes may be applied during deployment.

\(^5\)International Association of Women in Fire and Emergency Services: [http://bit.ly/1Jt0Mz5](http://bit.ly/1Jt0Mz5)
possible order which produces the desired result.

None of the common languages handle this problem well, although it manifests itself in slightly different ways: SmartFrog simply composes the attributes using an override semantics in the order in which the components appear in the \texttt{extends} clause. Puppet uses a similar instance inheritance mechanism, although the semantics are complex (see \cite{3}) and this is now deprecated, except for the special case where a single base class is used to specify default values\footnote{There is some evidence that system administrators often find such inheritance mechanisms confusing\cite{10}, and it is likely that this is contributing to a move away from declarative configuration tools in some cases.}. LCFG has a very general mechanism which allows the configuration to define \textit{mutation} functions for an attribute. These are used to compute the resulting value from the values being composed. However, it is not always clear how to define a meaningful commutative function, and most practical configurations simply use \texttt{mSET} (right projection), which is equivalent to a simple override semantics that produces order-dependent compositions (one notable exception is the composition functions used for list composition).

Other mechanisms have been suggested, such as assigning priority numbers, or explicit dependencies (as used to define ordering in the Unix \texttt{initd} or \texttt{systemd} configuration files). But both of these require each component to be aware of all the other components with which they may potentially be composed.

The real difficulty is that most languages require one, and only one value for each attribute. Different authors will supply values for different reasons, and knowledge of this meta-information is essential to automatically resolve conflicts in an appropriate way:

\begin{itemize}
\item “The value really must be 42.”
\item “I don’t really care what the value is, but I can’t leave it empty, so I’ll give it the value 0.”
\item “36 would be a good value, but I don’t care if someone else would rather have something different.”
\item “I think it should be 46, but if Jane thinks it should be different, then believe her.”
\item “The value must be between 100-200, but I can’t specify a range, so I’ll say 150.”
\end{itemize}

An ideal configuration language would provide a composition mechanism which allows the authors of the components (rather than the users) to specify how conflicting values should be resolved.

ConfSolve \cite{9,8} allows values to be given as arbitrary constraints which are then solved by a general-purpose constraint engine. This is capable of support-

\section{Composition in L3}

We would like a composition operator to have the following properties:

\begin{itemize}
\item Commutative:
\[ a <+> b \equiv b <+> a \]
\item Associative:
\[ (a <+> b) <+> c \equiv a <+> (b <+> c) \]
\item Idempotent:
\[ a <+> a \equiv a \]
\item Identity (\texttt{undef}):
\[ a <+> \texttt{undef} \equiv \texttt{undef} <+> a \equiv a \]
\end{itemize}

These follow from the discussions in \ref{1.2} and the following observations:

\begin{itemize}
\item The commutativity and associativity imply that the order of composition is insignificant. This allows an author to add requirements by composing additional components, without considering the order in relation to the existing components.
\item The idempotence allows values to be specified without considering whether someone else has already specified the same value.
\item Notice that there is no requirement for an inverse – authors must be able to specify values which cannot be revoked or over-ridden by anyone else.
\end{itemize}

For blocks, it is sufficient to compose the values for corresponding labels (recursively). Conflicting primitive values require additional information to resolve. L3 allows this to be provided in the form of simple annotations consisting of \texttt{tags} and \texttt{constraints}. Two special tags appear to be sufficient for many practical situations:

\begin{itemize}
\item the \texttt{#default} tag specifies a value which is to be used (only) if no non-default values are specified. If there are multiple defaults, then an arbitrary (but deterministic) one of them is used\footnote{The appropriate action in this case probably requires a little more evaluation – such arbitrary choices are technically correct, but users may find them too confusing, and prefer them to be classed as errors (the current compiler is able to generate warnings).}.
\item the \texttt{#final} tag specifies a value to be used in preference to all other values. It is an error if there are multiple values with final tags.
\end{itemize}
Values which are not annotated are used if neither of the above cases holds. If there are multiple candidates, then an arbitrary (but deterministic) one of them is used. Tags attached to blocks are inherited (recursively) by all of the resources in the block.

This process satisfies the above properties, and meets the requirement that the conflict resolution is determined by the author, rather than the user of a component. The figure in section 2.1 shows how this might be used to specify the composition required in 1.2.

**Arbitrary tags:** The current implementation of the L3 compiler also supports arbitrary tags and simple precedence constraints on these tags:

```
fig: {
  head: {
    face: "male"
    hair: {
      style: "short"
      colour: "brown"
    }
  }
  clothing: {
    top: "bluetop"
    bottom: "bluebottom"
  }
} #default
```

```
fireperson: $fig <+> {
  head: {
    hair: {
      style: "short"
      colour: "red"
    }
  }
  hat: {
    style: "fireHat"
    colour: "red"
  }
  clothing: {
    top: "firetop"
    bottom: "redbottom"
  }
} #final
```

```
female: $fig <+> {
  head: {
    face: "female" #final
    hair: {
      style: "long"
    }
  }
}
```

```
alice: $female
bob: $fireperson
Carol: $female <+> $fireperson
Eve: $fireperson <+> $female
```

Note that the constraints would normally be inherited from an enclosing block (probably the top-level configuration), rather than being attached to individual resources. This makes it possible to design configurations which behave differently in different contexts – for example, we may have a configuration which can be prioritised for security or convenience depending on the context. However, we are wary of “over-clever” use of the language which hides the simplicity of the fundamental requirements.

Similarly, it would be fairly straightforward to expand the supported constraints to include more complex relationships, but we are keen to maintain a balance between clarity/usability and expressiveness, and we would like to evaluate the existing mechanism more thoroughly on real configurations before proposing any further extensions.

**Specialisation:** Specialisation is implemented simply by adding anonymous labels and a corresponding constraint to the arguments:

```
a +> b ≡
( b #_L1 <+> a #_L2 ) #_L2 > #_L1
```

The details of the tag (and constraint) propagation semantics require more evaluation to ensure that the results are reasonably intuitive. This has some similarities with alternative models for provenance in configuration languages[2]
References


Links


