A Formal Semantics for the SmartFrog Configuration Language

A Formal Semantics for the SmartFrog Configuration Language

Paul Anderson · Herry Herry

Abstract *System Configuration Languages* are now widely used to drive the deployment and evolution of large computing infrastructures. Most such languages are highly informal, making it difficult to reason about configurations, and introducing an important source of failure. We claim that a more rigorous approach to the development and specification of these languages will help to avoid these difficulties and bring a number of additional benefits. In order to test this claim, we present a formal semantics for the core of the *SmartFrog* configuration language. We demonstrate how this can be used to prove important properties such as termination of the compilation process. To show that this also contributes to the practical development of clear and correct compilers, we present three independent implementations, and verify their equivalence with each other, and with the semantics. Supported by an extended example from a real configuration scenario, we demonstrate how the process of developing the semantics has improved understanding of the language, highlighted problem areas, and suggested alternative interpretations. This leads us to advocate this approach for the future development of practical configuration languages.

Keywords Configuration management · configuration languages · semantics · SmartFrog · system administration · formal specification

Acknowledgements This work has been partially supported by an HP Labs Innovation Research Program Award, and we would like to thank Patrick Goldsack for his help in clarifying the SmartFrog semantics and implementation. It is also partially sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3006. The U.S. Government and University of Edinburgh are authorized to reproduce and distribute reprints for their purposes notwithstanding any copyright notation thereon.

P. Anderson
School of Informatics, University of Edinburgh, Edinburgh EH8 9AB
Tel.: +44-131-6513241
Fax.: +44-131-6502957
E-mail: dcspaul@ed.ac.uk

H. Herry*
Hewlett-Packard Labs, Bristol
Tel.: +44-560-1090011
E-mail: herry.herry@hp.com
* *The author did part of the work for this paper while being a PhD student at School of Informatics, the University of Edinburgh.*

1 Introduction

System configuration languages [1–3,7,9] are now ubiquitous. Almost all major installations and services depend on specifications written in these languages to deploy and manage the software and configuration files which define the very purpose of the system. *Puppet* [9] for example, is used by Google - to manage "many many thousands" of developer workstations; The international securities exchange (ISE); Los Alamos National Laboratory; GitHub; WikiMedia; and the Facebook gaming company Zynga.

This clearly represents an advance on the former ad-hoc approaches. However, most popular configuration languages are extremely informal. The behaviour is defined implicitly by a single implementation, and the languages are often used in ways that interact with the underlying implementation language to produce systems which are very difficult to understand or reason about.

> *"In his case study about Linux system engineering in air traffic control, Stefan Schimanski showed how scalable Puppet really is and how it can guarantee reliable mass deployment of the Linux-based, mission critical applications needed in air traffic control centers."* [32]

In critical applications such as air traffic control, this creates a huge gulf between the rigorous development of the application software itself, and the configuration of the infrastructure on which it runs. It is likely that configuration errors will continue to be a major source of system failures [28], and that the increasing scale of the automation will lead to larger and more dramatic failures.

One way to ease this problem is to provide a clearer, more formal semantics for the specification language itself. In cases where the language is intimately connected to the underlying implementation language, this is very difficult. However, for a well-designed declarative specification language which is decoupled from the implementation language, this would yield the kind of practical benefits which we expect from a well-designed programming language:

- The formal semantics acts as a precise, independent reference for implementors, so that we can confidently create different implementations which behave in the same way.
- It is possible to create tools which analyse the configurations in useful ways; for example, automatically analysing the *provenance* of configurations for fault analysis and security [15].
- It is easier to create interoperable tools, analogous to a software development environment, which might include tools for generating, analysing, refactoring and visualising configurations.
- A formal definition of the language supports the ability to prove certain properties of the configuration, increasing the reliability and security of the configurations.
- The formal semantics can be used to guide the implementation and structure of the compiler.

A less obvious, but very significant benefit is that the process of formalising the semantics often exposes ambiguities and other issues with the language itself. This ultimately leads to a cleaner language design which is less prone to misunderstanding and errors.

We are not aware of any other work which formalises the complete evaluation of an existing production language: Our contribution in this paper is to present a complete formal semantics for the core of the SmartFrog [11] configuration language, and to demonstrate that this approach can be used to produce reliable, compatible tools.

We start by giving sufficient background to establish a common understanding of "configuration languages" in general (§2), and the SmartFrog language in particular (§3). We include some background on the semantics to make the formal processes more accessible to configuration practitioners (§4), and we discuss some of the particular characteristics of system configuration languages which affect the choice of semantics (§5). We then present the detailed semantics for SmartFrog (§6), and demonstrate how this can be used to prove some formal properties (§7). We finish by describing the practical implementation and evaluation of several compilers (§8), and discussing some of the insights which the process has provided into the language (§9) and its application to real configuration scenarios (§10).

2 System configuration languages

System configuration tools are used to automatically deploy a new system, or reconfigure an existing one into some desired state. These tools are driven by specifications in a *system configuration language*. There are a range of approaches to the tools and corresponding languages. At one extreme, the language may be an extension of a conventional imperative scripting language. At the other extreme, it may be a domain-specific language which provides a declarative description of the desired state. Most tools tend towards one of these extremes, but incorporate elements of the other.

Tools which advocate a more imperative approach are easier for administrators to adopt, since they are usually a natural extension of an existing manual, or scripted process; there is little, or no explicit specification of the desired configuration state, and the administrator uses the language to manually define a workflow to achieve the desired result – for example, *Ansible* [1]. These workflows need careful testing to make sure that they achieve the (implicit) desired state, and they can be brittle if the initial state of the system is one which has not been anticipated.

More declarative tools usually provide a custom language to define the desired state of the system, which is independent of the deployment process – for example *Puppet* [9]. This makes it easier to specify and reason about the desired configuration, but harder to control the sequence in which the reconfiguration takes place. However, automated planning techniques can now be used to generate workflows [24] which are guaranteed to meet any required constraints, and to achieve the goal state, from any viable initial state.

There are often valid pragmatic reasons for including some imperative aspects in a configuration language. But for many imperative tools, the configuration "specification" is tied to the underlying implementation language, and the deployment process – for example, specifications in *Chef* [3] contain arbitrary Ruby code. The semantics for such a specification is (a) likely to require a complete semantics for the embedded imperative language, and (b) unlikely to capture the "higher level" intent of the configuration specification.

The use of a separate language for declarative configuration allows us to concentrate on a clear statement of the requirements; language features can be provided to support domain-specific concerns such as effective sharing of common configuration elements, composing requirements from many different sources, and supporting "loose" specifications [26]. Supporting these "high level" specifications typically requires features such as instance inheritance and other structuring facilities in the source language. These must be compiled down into explicit configuration parameters for each subsystem of each machine. The (separate) deployment process can then be orchestrated independently using whatever languages and tools are most appropriate. The semantics for a good declarative configuration language should therefore be simpler, and able to capture the essence of the configuration requirements. As well as being clearer and less error-prone, specifications in this form can be used for a wider range of purposes than simply driving the deployment process.

3 SmartFrog

SmartFrog [11] (Smart Framework for Object Groups) is "a framework for configuring and automatically activating distributed applications". The SmartFrog language (SF) is a declarative language for describing system configurations. This is independent of the runtime deployment system, self-contained, and has a well-defined (informal) syntax and semantics, with an open-source implementation. The language is sufficiently rich to include many of the significant features supported by other declarative languages, and a general object-model which can represent relationships between arbitrary objects. The style of the language is typical of declarative configuration languages, being broadly based on untyped attribute-value pairs with prototype-based instance inheritance. This makes it a good candidate for development of an exploratory formal semantics.

3.1 A SmartFrog example

We can illustrate the main features of the SmartFrog language using the configuration shown in figure 1. This consists of four machines: servers s_1 and s_2, and desktop clients pc_1 and pc_2. Each machine has an attribute *dns* that holds the address of the DNS server. Each server has a *web* service which runs on port 80 and can be in a *running* or *stopped* state. Both pc_1 and pc_2 are using the *web* service from s_1. Figure 2 shows the specification of this example system in SF.

SF Objects are collections of (untyped) attribute-value pairs, and the language provides an *instance-based* inheritance mechanism which allows objects to inherit attribute *values* from other *prototype* objects. This is common in configuration languages, but it is different from the class-based type inheritance which is typical in programming languages such as Java or C++ (although Javascript uses a similar inheritance mechanism, as does Self [10]). The values of inherited attributes can be overridden with more specific values, and this enables code reuse through inheritance and composition.

Lines 1-3 define a prototype component `Machine` with a single attribute `dns` which holds the address of the DNS server. Lines 4-7 define another prototype

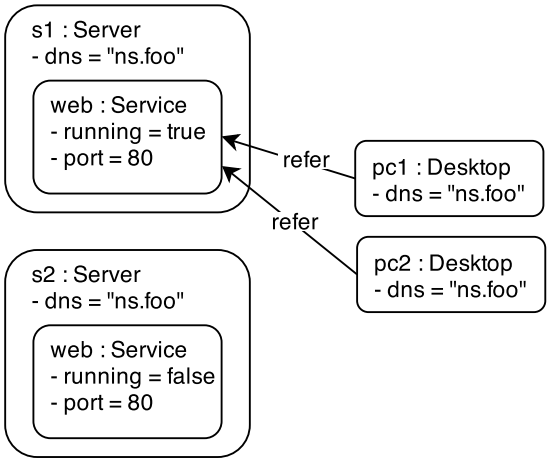

Fig. 1: An example system that consists of two web servers and a client.

```
1  Machine extends {
2    dns "ns.foo";
3  }
4  Service extends {
5    running true;
6    port 80;
7  }
8  sfConfig extends {
9    s1 extends Machine, {
10     web extends Service
11   }
12   s2 extends s1, {
13     web:running false;
14   }
15   pc1 extends Machine, {
16     refer DATA s1:web;
17   }
18   pc2 pc1;
19 }
```

Fig. 2: The SF specification for the configuration in figure 1.

component `Service` with two attributes: `running` and `port` which hold the service state and port number respectively. Lines 8-19 specify the main component `sfConfig`, which defines the specification of the overall system.

Lines 9-11 describe a server instance `s1` which uses `Machine` as prototype. In addition to the `dns` attribute (inherited from `Machine`), it has a `web` attribute which represents the service. Lines 12-14 describe a second server `s2` which again uses `s1`

as prototype where the inherited value of `web.running` (`true`) is overridden with `false`. Lines 15-17 define desktop client `pc1` which also uses `Machine` as a prototype.

SF also supports several types of *reference* between objects: the default reference type (*link reference*) is evaluated at compile time so that the value of the defined attribute is the same as the value of the referenced attribute. For example, line 18 defines the desktop client `pc2` to have the same attributes and values as `pc1`. *Data references* are prefixed with the `DATA` keyword and are not evaluated by the compiler (i.e. the reference simply evaluates to the identifier which is then interpreted by the runtime system); for example, line 16. The ":" operator is used to refer to sub-elements (for example, `s1:web`).

Figure 3 shows the output from the compiler (presented here in YAML [12]), which is an evaluation of the main `sfConfig` object.

```
1  s1:
2    dns: "ns.foo"
3    web:
4      running: true
5      port: 80
6  s2:
7    dns: "ns.foo"
8    web:
9      running: false
10     port: 80
11 pc1:
12   dns: "ns.foo"
13   refer: s1:web
14 pc2:
15   dns: "ns.foo"
16   refer: s1:web
```

Fig. 3: The evaluation of the specification in figure 2.

The above example covers the SF core features: prototyping, component (object), primitive value, data reference, link reference, and main component (`sfConfig`). SF also supports vectors (not shown in the example) which can be used to represent lists of arbitrary types. For example:

```
1 sfConfig extends {
2   myvector [ true, 1, [ false, 0 ] ];
3 }
```

Although evaluation appears to be a fairly simple process, it is not explicit in this informal description exactly how these basic features interact. For example, there are several possible interpretations of a specification in which an object overrides an attribute with one which has a different structure. It is exactly this kind of ambiguity which a formal semantics is able to clarify (see §9.3, for example).

4 Syntax & semantics

This section provides a brief overview of the notation and style of semantics used in the remainder of the paper.

4.1 Syntax

The *syntax* of a language defines the strings of characters which make meaningful statements in the language. This is usually expressed in BNF (Backus-Naur Form) notation. For example:

$$\begin{array}{lcl} \textit{digit} & ::= & \texttt{0}|\texttt{1}|\texttt{2}|\texttt{3}|\texttt{4}|\texttt{5}|\texttt{6}|\texttt{7}|\texttt{8}|\texttt{9} \\ \textit{num} & ::= & \textit{digit} \mid \textit{num}\ \textit{digit} \end{array}$$

If there are a finite number of a certain element (such as the decimal digits) then we can list them explicitly. Otherwise we can specify them using recursion (the *num* above), or we can use the Kleene $(\dots)^*$ and $(\dots)^+$ with the same meaning as in a regular expression.

Note that we use `typewriter` font to indicate literal strings that actually appear in the language itself, and *italics* to indicate *non-terminals* which represent a whole class of elements.

4.2 Semantics

The syntax of a language tells us which expressions are legal in the language, but it says nothing about the *meaning* of those expressions. A syntactically correct configuration specification (in Puppet, for example) will compile, but the language syntax on its own is insufficient to understand the configuration that will actually appear when this specification is deployed.

We need a *semantics* [29] to add this definition of the meaning. For example, the following strings intuitively have the same meaning (semantics), although they are written using different notations (syntax):

`forty two` , `42` , `0x2A` , `XLII`

4.2.1 Valuation functions

A *valuation function* specifies how to map the elements of the syntax into something which represents their "meaning" (a *denotation*). For example[1]:

$$\begin{array}{rcl} \mathbf{F}\ [\![\ \texttt{forty two}\ ]\!] & \triangleq & 42 \\ \mathbf{F}\ [\![\ \texttt{42}\ ]\!] & \triangleq & 42 \\ \mathbf{F}\ [\![\ \texttt{0x2A}\ ]\!] & \triangleq & 42 \\ \mathbf{F}\ [\![\ \texttt{XLII}\ ]\!] & \triangleq & 42 \end{array}$$

The elements on the left hand side (typewriter font) are literal strings which appear in the source language, while the elements on the right hand side (roman font) are abstract numbers. The symbol $\triangleq$ is read as "is defined as". The second line, for example, says that the "meaning" of the ASCII character `4` followed by the ASCII character `2` is the number 42.

In the same way that we can use recursion in the syntax to describe source strings of arbitrary length, we can use recursion in the semantics to define the meaning of these strings. We can do this easily with the decimal representation above, but it is shorter to illustrate using binary numbers (see [30], p. 27):

$$\begin{array}{rcl} \mathbf{F}\ [\![\ \texttt{0}\ ]\!] & \triangleq & 0 \\ \mathbf{F}\ [\![\ \texttt{1}\ ]\!] & \triangleq & 1 \\ \mathbf{F}\ [\![\ N\,\texttt{0}\ ]\!] & \triangleq & 2\times\ \mathbf{F}\ [\![N]\!] \\ \mathbf{F}\ [\![\ N\,\texttt{1}\ ]\!] & \triangleq & 2\times\ \mathbf{F}\ [\![N]\!] + 1 \end{array}$$

Notice that N is a variable in the domain of the language (i.e. a string of binary digits), and $\mathbf{F}\ [\![N]\!]$ is the denotation of N (i.e. the number represented by the string N).

With a bit more work, we could create similar expressions for the other representations. And the semantics can be expanded to incorporate further constructs of the language in the same way. For example, two different versions of multiplication syntax:

$$\begin{array}{lcl} \mathit{term} & ::= & \mathit{num} \mid \texttt{(mul}\ \mathit{term}\ \mathit{term}\texttt{)} \\ \mathit{term} & ::= & \mathit{num} \mid \mathit{term}\ \texttt{*}\ \mathit{num} \end{array}$$

may have the same semantics:

$$\begin{array}{rcl} \mathbf{F}\ [\![\texttt{(mul}\ \mathit{T1}\ \mathit{T2}\texttt{)}]\!] & \triangleq & \mathbf{F}\ [\![\mathit{T1}]\!] \times \mathbf{F}\ [\![\mathit{T2}]\!] \\ \mathbf{F}\ [\![\ \mathit{T1}\texttt{*}\mathit{T2}\ ]\!] & \triangleq & \mathbf{F}\ [\![\mathit{T1}]\!] \times \mathbf{F}\ [\![\mathit{T2}]\!] \end{array}$$

Notice that the `*` on the lefthand side is the ASCII character "`*`", but the $\times$ on the righthand side is the abstract mathematical multiplication operator.

4.2.2 Stores

In the above examples, evaluation depends only on the argument to the function. Most real languages involve some kind of *store* to hold, for example, the values of the variables in an imperative programming language, or a configuration language. Many operations then depend on the values in this store, as well as their arguments.

As a simple example, a calculator may have a memory (store), and an `M+` operation which adds a number to the memory. It is usual to represent such operations as functions which take the old contents of the store and return the new contents. For example:

$$\begin{array}{rcl} \mathbf{M+}: & \mathit{Integer} \rightarrow \mathit{Memory} \rightarrow \mathit{Memory} & \\ \mathbf{M+}\ [\![\,\mathrm{N}\,]\!] & \triangleq & \lambda\, s\,.\,(s+N) \end{array}$$

The above λ (*lambda*) notation simply means "a function which adds N to its argument". So the meaning of the operation $\mathbf{M+}\ [\![\,5\,]\!]$ is "a function which adds 5 to its argument". Repeated operations then translate naturally into nested applications of the function. For example:

$$\mathbf{M+}\ [\![\,2\,]\!](\mathbf{M+}\ [\![\,4\,]\!](\mathbf{M+}\ [\![\,3\,]\!](0)))$$

4.2.3 Semantic algebras

The combination of the *semantic operators*, together with the *semantic domain* of objects that they operate on, is known as the *semantic algebra* – in this case, we are dealing with arithmetic operators over the natural numbers.

The examples above are a lot simpler than any real language, but it should be possible to see how this approach can be used to address some of questions raised in the introduction: for example, we could *prove* that "one hundred and seven" (in one language) had the same meaning as "CVII" (in some other language). And, if we were writing a new compiler, we would have an unambiguous description of the intended result that we could use during implementation and testing.

5 Semantics of configuration languages

There are several different approaches to specifying the semantics of programming languages. However, declarative configuration languages are quite different from most programming, or scripting languages – they define the characteristics of the intended configuration rather than the *process* of deploying that configuration. In many ways, this makes the semantics much simpler. However, they often emphasise additional features which may not be prominent in a mainstream "programming" language (such as prototypes, value-inheritance and composition). This motivates our choice of the *denotational* approach to the semantics, as described in the previous section, rather than an *operational* or *axiomatic* approach. This is consistent with the approach normally used in other domains, such as database query languages.

5.1 The semantic domain

The values of the denotations belong to some semantic domain. For a programming language, this is usually some form of abstract "store" representing a computer memory and the values that it holds. The result of evaluating a program is taken to be the contents of the memory when the program terminates. However, the meaning of "evaluation" may be different for different types of language. For example with CSS/HTML we might be interested in the resulting attributes on the various elements.

One of the challenges for configuration languages is to choose a meaningful abstract representation for the results of the evaluation. At one extreme, we might consider the entire contents of the target machine disk to be the result of a configuration process. However, most declarative tools provide a more useful description of the configuration at the level where the compiler interfaces with the deployment engine. This is usually equivalent to a list of attribute-values pairs which represent meaningful abstractions of configuration parameters such as "the IP address of the DNS server". The Puppet *catalog* is a typical example. This is discussed further in §9.4.

5.2 The abstract syntax

There is also some choice about the level of abstraction of the "input" to the semantic functions; in the previous section, we used the concrete syntax of the source language. For a realistic language, the semantics would usually deal with slightly higher-level concepts represented by some *abstract syntax*; this would typically include *terminals* such as *integer*, *identifier* or *string*, without describing their detailed representation as input strings.

5.3 Types

Most configuration languages are not strongly-typed. Attribute values are usually arbitrary strings, and attributes can often be dynamically added with arbitrary names. Although it is sometimes possible to specify validation expressions or "schema" which restrict the allowable values for some attributes, type checking typically only occurs at runtime. Stronger typing is useful in some applications – for example when generating configurations from constraints [25], or when planning workflows [24]. However, the core SmartFrog language is dynamically-typed, and there is no type-checking during the compilation process, so we do not discuss typing issues further in this paper.

5.4 Previous work

We are aware of no significant related work specifically on the semantics of "system" configuration languages such as those discussed above. Couch [19] describes the "meaning crisis that system administrators face", arguing for a clearer semantics, and in [20], he describes an operational approach to the formalisation of the deployment process which defines the transformation of one configuration into another. Hewson [25] provides a semantics for the ConfSolve language, although the domain in this case is the MiniZinc input to a constraint solver, rather than a more explicit representation of the configuration itself. Bekezhanova [16] discusses a semantics for Puppet, but this is not developed in detail.

A number of languages have been designed for alternative approaches to the configuration problem - for example *policy languages* which usually describe *actions* triggered by *events*. Such languages are concerned with the sequence of actions required to deploy a configuration. As noted in section 2, this is deliberately outside the scope of a declarative configuration language such as SmartFrog which is focussed on describing the collaborative construction of the desired state of the system, rather than the process of deploying that state. Formal semantics have been developed for several such languages [27, 13], including Ponder [21]. These tend to use an operational semantics (rather than a denotational approach) which is more suited to the imperative nature of the actions.

The term "configuration" is also used in a number of different disciplines, and the value of a formal semantics has been demonstrated in many related areas (for example, feature composition for product configuration [18]), but none of these are specifically relevant to the denotational semantics of system configuration languages as discussed in this paper.

Some aspects of programming language semantics are relevant however, particularly for illustrating the difficulty of trying to clarify the semantics of languages which have been developed in an informal way – for example, the the C preprocessor [22].

6 Semantics of SmartFrog

This section presents the formal semantics of the core SF language: the abstract syntax (§6.1), the domains (§6.2), the basic operations on the domains (§6.3) and the valuation functions (§6.4).

The production language supports a few additional features such as the ability to "include" files, and to evaluate simple builtin functions – for example, basic arithmetic and string manipulation. These do not present any special semantic difficulties, but they are not included in the core semantics for simplicity. The production language also supports an extended semantics for references which is not included here, but is discussed in section §9.2.

6.1 Abstract syntax

Definition 1 (Terminal Symbols) These are the basic symbols of the language which appear in the source code:

$$\begin{array}{rcl} \text{Bool} & \in & \mathit{Boolean} \\ \text{Num} & \in & \mathit{Number} \\ \text{Str} & \in & \mathit{String} \\ \text{I} & \in & \mathit{Identifier} \\ \text{Null} & \in & \mathit{NullValue} \end{array}$$

Definition 2 (Non-Terminals) These are the non-terminal elements of the syntax:

$$\begin{array}{rcl} \text{SF} & \in & \mathit{SFSpecification} \\ \text{B} & \in & \mathit{Block} \\ \text{A} & \in & \mathit{Assignment} \\ \text{P} & \in & \mathit{Prototype} \\ \text{V} & \in & \mathit{Value} \\ \text{R} & \in & \mathit{Reference} \\ \text{DR} & \in & \mathit{DataReference} \\ \text{LR} & \in & \mathit{LinkReference} \\ \text{Vec} & \in & \mathit{Vector} \\ \text{BV} & \in & \mathit{BasicValue} \end{array}$$

Definition 3 (Syntax) The non-terminals are defined by the following *abstract* syntax. Note that we do not specify the details of the concrete syntax; for example, an *assignment* includes a *reference* and a *value* – but we do not care about the punctuation or the keywords which are used to represent this in the source code (the concrete syntax is given in appendix A).

SF	::=	B	// *SF specification*
B	::=	A B | ϵ	// *block*
A	::=	R V	// *assignment*
P	::=	R P | B P | ϵ	// *prototype*
V	::=	BV | LR | P	// *value*
R	::=	(I)$^+$	// *reference*
DR	::=	R	// *data reference*
LR	::=	R	// *link reference*
Vec	::=	(BV)*	// *vector*
BV	::=	Bool | Num | Str | DR | Vec | Null	

The symbol ϵ represents the empty string, and (L)* and (L)$^+$ represent possibly-empty, and non-empty lists of elements of type L respectively.

6.2 Domains

The primary semantic domain is used to represent the output of the configuration process. For the SF compiler, this is a tree of attribute-value pairs such as that shown in figure 3. Figure 4 shows a simple example, together with the representation used in the semantics.

```
1  a extends {
2    c 3;
3  }
4  b extends {
5  }
```

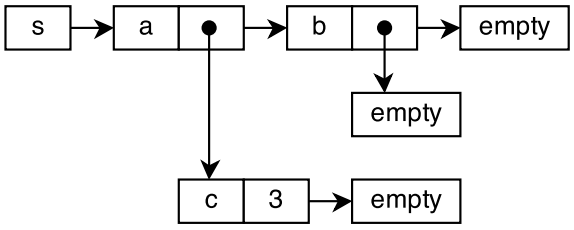

Fig. 4: An example of a compiled specification together with the corresponding store representation.

Definition 4 (List Data Structure) Elements of $(L)^*$ are lists of L-elements: $l_1 :: l_2 :: ... :: l_n :: \varnothing_L$, $n \geq 0$, and $\varnothing_L$ represents an empty list. If $x = l_1 :: l_2 :: ... :: l_n :: \varnothing_L$, then $|x| = n$ and $l' \in x$ iff $\exists i \,.\, l' = l_i$.

Definition 5 (The secondary domains) The values of the attributes belong to one of these secondary domains:

- $\mathbb{I}$, the identifier domain;
- $\mathbb{B} = \{\mathit{True}, \mathit{False}\}$, the boolean domain;
- $\mathcal{N} = \{\mathit{Null}\}$, the null domain;
- $\mathbb{R}$, the real number domain;
- $\mathbb{S}$, the string domain;
- $\mathcal{R} = \{r_i \,|\, r_i \in (\mathbb{I})^*\}$, the reference domain, where $\varnothing_{\mathbb{I}}$ is an empty reference;
- $\mathbb{V} = \mathbb{B} \cup \mathcal{N} \cup \mathbb{R} \cup \mathbb{S} \cup \mathcal{R} \cup \{v_i \,|\, v_i \in (\mathbb{V})^*\}$, the basic value domain.

Definition 6 (The primary domain) Formally, the *Store* domain $\mathcal{S} = (\mathbb{I} \times \mathcal{V})^*$ is a list of identifier-value pairs, where $\mathcal{V} = \mathbb{V} \cup \mathcal{S}$. i.e. the elements of the store are either basic values or (recursively) sub-stores. We use $\varnothing_{\mathcal{S}}$ to represent an empty store, and $\mathcal{V}_\perp = \mathcal{V} \cup \{\perp\}$ for the lifted domain which includes the undefined value $\perp$.

A *reference* is the sequence of identifiers specifying the "path" to a particular attribute in the tree structure. This is similar to a filesystem pathname - for example in figure 4, the reference a :: b :: $\varnothing_{\mathbb{I}}$[2] refers to the attribute with value `3`. An empty reference ($\varnothing_{\mathbb{I}}$) refers the root. Note that a reference can be written as a concatenation of an identifier with another reference, for example: $r = id :: r'$ where $id \in \mathbb{I}$ and $r, r' \in \mathcal{R}$.

Note that vectors may be nested, and a single-element vector is not the same as the element itself.

6.3 Semantic operations

This section defines the fundamental operations on the semantic domain. These are analogous to the basic arithmetic operations in the examples from section §4.2; in this case they operate on the trees of attribute-value pairs rather than the natural numbers.

For completeness, we include formal definitions for all of the operations. Most of these are straightforward manipulations involving references and the hierarchical store. However, some such as *bind* (definition 13) and *inherit* (definition 18) have significant implications for the semantics, and these are discussed further in section §9.

Definition 7 (operator $\oplus$) $\oplus$ is a binary operator that returns the concatenation of the operands. Either operand may be an identifier or a reference, but the result is always a reference.

$\oplus : (\mathbb{I} \cup \mathcal{R}) \times (\mathbb{I} \cup \mathcal{R}) \to \mathcal{R}$

[2] The concrete syntax of this formal representation is `a:c`.

$$
\begin{array}{rcl}
id_1 \oplus id_2 & \triangleq & id_1 :: id_2 :: \varnothing_{\mathbb{I}} \\
\varnothing_{\mathbb{I}} \oplus r & \triangleq & r \\
(id :: r) \oplus \varnothing_{\mathbb{I}} & \triangleq & id :: r \\
(id :: r) \oplus r_2 & \triangleq & id :: (r \oplus r_2) \\
id \oplus r & \triangleq & id :: r \\
\varnothing_{\mathbb{I}} \oplus id & \triangleq & id :: \varnothing_{\mathbb{I}} \\
(id_1 :: r) \oplus id_2 & \triangleq & id_1 :: (r \oplus id_2)
\end{array}
$$

Where :: is the *cons* operators which prepends a value to a list, r is a list of identifiers, and $\varnothing_{\mathbb{I}}$ is the empty list.

Definition 8 (operator $\ominus$) $\ominus$ is a binary operator that returns the first operand with any common prefix removed.

$\ominus : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$

$$
\begin{array}{rcl}
\varnothing_{\mathbb{I}} \ominus r & \triangleq & \varnothing_{\mathbb{I}} \\
r \ominus \varnothing_{\mathbb{I}} & \triangleq & r \\
(id :: r_1) \ominus (id :: r_2) & \triangleq & r_1 \ominus r_2 \\
(id_1 :: r_1) \ominus (id_2 :: r_2) & \triangleq & id_1 \ominus r_1
\end{array}
$$

Definition 9 (operator $\equiv$) $\equiv$ is a binary operator that return *True* if two references are equal, otherwise it returns *False*.

$\equiv : \mathcal{R} \times \mathcal{R} \to \mathbb{B}$

$$
\begin{array}{rcll}
\varnothing_{\mathbb{I}} & \equiv & \varnothing_{\mathbb{I}} & \triangleq \ \mathit{True} \\
(id :: r) & \equiv & \varnothing_{\mathbb{I}} & \triangleq \ \mathit{False} \\
\varnothing_{\mathbb{I}} & \equiv & (id :: r) & \triangleq \ \mathit{False} \\
(id_1 :: r_1) & \equiv & (id_2 :: r_2) & \triangleq \ (id_1 = id_2) \ \wedge (r_1 \equiv r_2)
\end{array}
$$

Definition 10 (operators $\subseteq_{\mathcal{R}}$ and $\subset_{\mathcal{R}}$) $\subseteq_{\mathcal{R}}$ and $\subset_{\mathcal{R}}$ are true if the left reference is a (strict/non-strict) prefix of the right one.

$\subseteq_{\mathcal{R}} : \mathcal{R} \times \mathcal{R} \to \mathbb{B}, \quad \subset_{\mathcal{R}} : \mathcal{R} \times \mathcal{R} \to \mathbb{B}$

$$
\begin{array}{rcl}
r_1 \subseteq_{\mathcal{R}} r_2 & \triangleq & ((r_1 \ominus r_2) = \varnothing_{\mathbb{I}}) \\
r_1 \subset r_2 & \triangleq & (r_1 \subseteq_{\mathcal{R}} r_2) \wedge \neg (r_1 \equiv r_2)
\end{array}
$$

Definition 11 (*prefix*) The ***prefix*** function returns the longest strict prefix of the given reference.

$\mathit{prefix} : \mathcal{R} \to \mathcal{R}$

$$
\begin{array}{rcl}
\mathit{prefix}(\varnothing_{\mathbb{I}}) & \triangleq & \varnothing_{\mathbb{I}} \\
\mathit{prefix}(id :: \varnothing_{\mathbb{I}}) & \triangleq & \varnothing_{\mathbb{I}} \\
\mathit{prefix}(r :: id_n :: \varnothing_{\mathbb{I}}) & \triangleq & r :: \varnothing_{\mathbb{I}}
\end{array}
$$

Definition 12 (*put*) The ***put*** function updates the value of an identifier in a store, or adds it if it does not already exist. Notice that this operates only on single identifiers – the following function (bind) extends this to support hierarchical references.

$\mathit{put} : \mathcal{S} \times \mathbb{I} \times \mathcal{V} \to \mathcal{S}$

$$
\begin{array}{rcl}
\mathit{put}(\varnothing_{\mathcal{S}}, id, v) & \triangleq & \langle id, v\rangle :: \varnothing_{\mathcal{S}} \\
\mathit{put}(\langle id, v_s\rangle :: s', id, v) & \triangleq & \langle id, v\rangle :: s' \\
\mathit{put}(\langle id_s, v_s\rangle :: s', id, v) & \triangleq & \langle id_s, v_s\rangle :: \mathit{put}(s', id, v)
\end{array}
$$

Definition 13 (*bind*) The ***bind*** function updates the value of a reference in a store. An error occurs if an attempt is made to update a reference whose parent does not exist (***err*$_2$**), or whose parent is not itself a store (***err*$_1$**). It is also illegal to replace the root store (***err*$_3$**). For simplicity, we leave the error handling rather informal.

$\mathit{bind} : \mathcal{S} \times \mathcal{R} \times \mathcal{V} \to \mathcal{S}$

$$
\begin{array}{rcl}
\mathit{bind}(s, \varnothing_{\mathbb{I}}, v) & \triangleq & \boldsymbol{err_3} \\
\mathit{bind}(s, id :: \varnothing_{\mathbb{I}}, v) & \triangleq & \mathit{put}(s, id, v) \\
\mathit{bind}(\varnothing_{\mathcal{S}}, id :: r', v) & \triangleq & \boldsymbol{err_2} \\
\mathit{bind}(\langle id, v_s\rangle :: s', id :: r', v) & \triangleq & \text{if } v_s \in \mathcal{S} \text{ then } \langle id, \mathit{bind}(v_s, r', v)\rangle :: s' \\
 & & \text{else } \boldsymbol{err_1} \\
\mathit{bind}(\langle id_s, v_s\rangle :: s', id :: r', v) & \triangleq & \langle id_s, v_s\rangle :: \mathit{bind}(s', id :: r', v))
\end{array}
$$

Note that this behaviour is slightly different from the current version of SF – see the discussion in section §9.2.1.

Definition 14 (*find*) The ***find*** function looks up the value of a reference in a store.

$\mathit{find} : \mathcal{S} \times \mathcal{R} \to \mathcal{V}_{\perp}$

$$
\begin{array}{rcl}
\mathit{find}(s, \varnothing_{\mathbb{I}}) & \triangleq & s \\
\mathit{find}(\varnothing_{\mathcal{S}}, r) & \triangleq & \perp \\
\mathit{find}(\langle id, v_s\rangle :: s', id :: \varnothing_{\mathbb{I}}) & \triangleq & v_s \\
\mathit{find}(\langle id_s, v_s\rangle :: s', id :: \varnothing_{\mathbb{I}}) & \triangleq & \mathit{find}(s', id :: \varnothing_{\mathbb{I}}) \\
\mathit{find}(\langle id, v_s\rangle :: s', id :: r') & \triangleq & \text{if } v_s \in \mathcal{S} \text{ then } \mathit{find}(v_s, r') \text{ else } \perp \\
\mathit{find}(\langle id_s, v_s\rangle :: s', id :: r') & \triangleq & \mathit{find}(s', id :: r')
\end{array}
$$

Definition 15 (operators $\subset_{\mathcal{S}}$) $\subset_{\mathcal{S}}$ is true if the left store is a sub-store of the right one.

$\subset_{\mathcal{S}} : \mathcal{S} \times \mathcal{R} \to \mathbb{B}$

$$
s_1 \subset_{\mathcal{S}} s_2 \quad \triangleq \quad \exists r \in \mathcal{R} \text{ where } \mathit{find}(s_2, r) = s_1 \text{ and } r \neq \varnothing_{\mathbb{I}}
$$

Definition 16 (*resolve*) The ***resolve*** function looks up a reference in a store, by starting with a given *namespace* (reference of the sub-store) and searching up the hierarchy of parent stores until a value is found (or not). It returns a tuple $\langle ns, v\rangle$, where ns is the namespace in which the target element is found and v is the value. If the target is not found then $\langle ns, v\rangle = \langle \varnothing_{\mathbb{I}}, \perp\rangle$.

$\mathit{resolve} : \mathcal{S} \times \mathcal{R} \times \mathcal{R} \to \mathcal{R} \times \mathcal{V}$

$$
\begin{array}{rcl}
\mathit{resolve}(s, \varnothing_{\mathbb{I}}, r) & \triangleq & \langle \varnothing_{\mathbb{I}},\ \ \mathit{find}(s, r)\rangle \\
\mathit{resolve}(s, ns, r) & \triangleq & \text{if } v = \perp \text{ then } \mathit{resolve}(s, \mathit{prefix}(ns), r) \text{ else } \langle ns, v\rangle
\end{array}
$$

where $v = \mathit{find}(s, ns \oplus r)$

Definition 17 (*copy*) The ***copy*** function copies every attribute from the second store to the first store at the given prefix (*pfx*).

$copy : \mathcal{S} \times \mathcal{S} \times \mathcal{R} \to \mathcal{S}$

$$\begin{array}{rcl} copy(s_1, \varnothing_{\mathcal{S}}, pfx) & \triangleq & s_1 \\ copy(s_1, \langle id, v \rangle :: s_2, pfx) & \triangleq & copy\left(bind\left(s_1, pfx \oplus id, v\right), s_2, pfx\right) \end{array}$$

Definition 18 (*inherit*) The ***inherit*** function copies values from a given prototype (*proto*) to the target store (r). The prototype may be located in a higher-level namespace, hence the use of *resolve* to locate the corresponding store.

$inherit : \mathcal{S} \times \mathcal{R} \times \mathcal{R} \times \mathcal{R} \to \mathcal{S}$

$$inherit(s, ns, proto, r) \quad \triangleq \quad \text{if } v_p \in \mathcal{S} \text{ then } copy(s, v_p, r) \text{ else } \boldsymbol{err_4}$$

where $\langle ns_p, v_p \rangle = resolve(s, ns, proto)$

This function determines the behaviour of the prototype inheritance in SF – this is discussed further in section §9.3. $\boldsymbol{err_4}$ occurs if the value of the prototype is not a store.

6.4 Valuation functions

The valuation functions in this section show how each element of the abstract syntax of an SF specification is evaluated. Evaluation of a complete SF specification yields a store $s \in \mathcal{S}$:

Definition 19 (terminals) The terminal symbols are evaluated in the obvious way, as described in section §4.2, using functions with the following signatures:

$$\begin{array}{rl} \textbf{Bool}: & Boolean \to \mathbb{B} \\ \textbf{Num}: & Number \to \mathbb{N} \\ \textbf{Str}: & String \to \mathbb{S} \\ \textbf{I}: & Identifier \to \mathbb{I} \\ \textbf{Null}: & NullValue \to \mathcal{N} \end{array}$$

For example:

$$\begin{array}{rcl} \textbf{Num}\ [\![\ \texttt{42}\ ]\!] & \triangleq & 42 \\ \textbf{Bool}\ [\![\ \texttt{false}\ ]\!] & \triangleq & False \end{array}$$

Definition 20 (references) Both types of reference are evaluated to a list of identifiers:

$$\begin{array}{rl} \textbf{LR}: & LinkReference \to \mathcal{R} \\ \textbf{DR}: & DataReference \to \mathcal{R} \\ \textbf{R}: & Reference \to \mathcal{R} \end{array}$$

$$\begin{array}{rcl} \textbf{LR}\,[\![\,\mathrm{R}\,]\!] & \triangleq & \textbf{R}\,[\![\,\mathrm{R}\,]\!] \\ \textbf{DR}\,[\![\,\mathrm{R}\,]\!] & \triangleq & \textbf{R}\,[\![\,\mathrm{R}\,]\!] \\ \textbf{R}\,[\![\,\mathrm{I}_1, ..., \mathrm{I}_n\,]\!] & \triangleq & \mathrm{I}\,[\![\,\mathrm{I}_1\,]\!] :: ... :: \mathrm{I}\,[\![\,\mathrm{I}_n\,]\!] :: \varnothing_{\mathbb{I}} \end{array}$$

Definition 21 (vectors) Vectors are evaluated by evaluating each element:

$$\mathbf{Vec}: \quad \mathit{Vector} \to (\mathbb{V})^*$$

$$\begin{array}{rcl} \mathbf{Vec}\,[\![\,\mathrm{BV}_1, ..., \mathrm{BV}_n\,]\!] & \triangleq & \mathbf{BV}\,[\![\,\mathrm{BV}_1\,]\!] :: ... :: \mathbf{BV}\,[\![\,\mathrm{BV}_n\,]\!] :: \varnothing_{\mathbb{V}} \\ \mathbf{Vec}\,[\![\,\epsilon\,]\!] & \triangleq & \varnothing_{\mathbb{V}} \end{array}$$

Definition 22 (basic values) A basic value (BV) is one of the basic element types:

$$\mathbf{BV}: \quad \mathit{BasicValue} \to \mathbb{V}$$

$$\begin{array}{rcl} \mathbf{BV}\,[\![\,\mathrm{Bool}\,]\!] & \triangleq & \mathbf{Bool}\,[\![\,\mathrm{Bool}\,]\!] \\ \mathbf{BV}\,[\![\,\mathrm{Num}\,]\!] & \triangleq & \mathbf{Num}\,[\![\,\mathrm{Num}\,]\!] \\ \mathbf{BV}\,[\![\,\mathrm{Str}\,]\!] & \triangleq & \mathbf{Str}\,[\![\,\mathrm{Str}\,]\!] \\ \mathbf{BV}\,[\![\,\mathrm{DR}\,]\!] & \triangleq & \mathbf{DR}\,[\![\,\mathrm{DR}\,]\!] \\ \mathbf{BV}\,[\![\,\mathrm{Vec}\,]\!] & \triangleq & \mathbf{Vec}\,[\![\,\mathrm{Vec}\,]\!] \\ \mathbf{BV}\,[\![\,\mathrm{Null}\,]\!] & \triangleq & \mathbf{Null}\,[\![\,\mathrm{Null}\,]\!] \end{array}$$

Definition 23 (prototype) A prototype is a sequence of bodies or references. Bodies are evaluated directly, while references are first resolved (in the current context) and then evaluated. Composition proceeds right-to-left (since defined values override any corresponding values in an extended prototype).

$$\mathbf{P}: \quad \mathit{Prototype} \to \mathcal{R} \times \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$

$$\begin{array}{rcl} \mathbf{P}\,[\![\,\mathrm{B}\ \ \mathrm{P}\,]\!] & \triangleq & \lambda\,(ns, r, s)\,.\ \mathbf{P}\,[\![\,\mathrm{P}\,]\!]\,(ns, r, \mathbf{B}\,[\![\,\mathrm{B}\,]\!](r, s)) \\ \mathbf{P}\,[\![\,\mathrm{R}\ \ \mathrm{P}\,]\!] & \triangleq & \lambda\,(ns, r, s)\,.\ \mathbf{P}\,[\![\,\mathrm{P}\,]\!]\,(ns, r, \mathit{inherit}(s, ns, \mathbf{R}\,[\![\,\mathrm{R}\,]\!], r)) \\ \mathbf{P}\,[\![\,\epsilon\,]\!] & \triangleq & \lambda\,(ns, r, s)\,.\ s \end{array}$$

Definition 24 (value) A value is either a basic value, a prototype, or a link reference. Basic values are entered directly in the store. Prototypes are first evaluated, and link references are first resolved.

$$\mathbf{V}: \quad \mathit{Value} \to \mathcal{R} \times \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$

$$\begin{array}{rcl} \mathbf{V}\,[\![\,\mathrm{BV}\,]\!] & \triangleq & \lambda\,(ns, r, s)\,.\ \mathit{bind}(s, r, \mathbf{BV}\,[\![\,\mathrm{BV}\,]\!]) \\ \mathbf{V}\,[\![\,\mathrm{P}\,]\!] & \triangleq & \lambda\,(ns, r, s)\,.\ \mathbf{P}\,[\![\,\mathrm{P}\,]\!]\,(ns, r, \mathit{bind}(s, r, \varnothing_{\mathcal{S}})) \\ \mathbf{V}\,[\![\,\mathrm{LR}\,]\!] & \triangleq & \lambda\,(ns, r, s)\,.\ \text{if } v' = \bot \text{ then } \boldsymbol{err_5} \text{ else } \mathit{bind}(s, r, v') \end{array}$$

where $\langle ns', v' \rangle = \mathit{resolve}(s, ns, \mathbf{LR}\,[\![\,\mathrm{LR}\,]\!])$

$\boldsymbol{err_5}$ occurs if the value referred to by the link-reference (LR) does not exist.

Definition 25 (assignment) To assign a value to a reference, the store entry for the reference is updated to contain the value.

$$\mathbf{A}: \quad \mathit{Assignment} \to \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$

$$\mathbf{A}\,[\![\,\mathrm{R}\ \mathrm{V}\,]\!] \quad \triangleq \quad \lambda\,(ns, s)\,.\ \mathbf{V}\,[\![\,\mathrm{V}\,]\!](ns,\ ns \oplus r,\ s)$$

Definition 26 (body) A body is a sequence of assignments. These are recursively evaluated left-to-right with the store resulting from one assignment being used as input to the next assignment.

$$\mathbf{B}: \quad \mathit{Body} \to \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$

$$\begin{aligned} \mathbf{B}[\![\,\mathrm{A}\;\mathrm{B}\,]\!] &\triangleq \lambda\,(ns,s)\,.\,\mathbf{B}[\![\,\mathrm{B}\,]\!]\,(ns,\mathbf{A}[\![\,\mathrm{A}\,]\!](ns,s)) \\ \mathbf{B}[\![\,\epsilon\,]\!] &\triangleq \lambda\,(ns,s)\,.\,s \end{aligned}$$

Definition 27 (specification) A complete *SFSpecification* is evaluated as a body, in the context of an empty store $\varnothing_{\mathcal{S}}$ and a reference $\varnothing_{\mathbb{I}}$ to the root namespace. The evaluation of the main `sfConfig` component is returned (other components are ignored - see figure 2).

$$\mathbf{SF}: \quad \mathit{SFSpecification} \rightarrow \mathcal{S}$$

$$\mathbf{SF}[\![\,\mathrm{B}\,]\!] \triangleq \text{if } v \in \mathcal{S} \text{ then } v \text{ else } \boldsymbol{err_7}$$

Where $r = \texttt{sfConfig} :: \varnothing_{\mathbb{I}}$, and $v = \mathit{find}(\mathbf{B}[\![\,\mathrm{B}\,]\!](\varnothing_{\mathbb{I}}, \varnothing_{\mathcal{S}}),\, r)$

It is an error ($\boldsymbol{err_7}$) if the main `sfConfig` element is not a store (for example, if it is a basic value).

7 Provable properties

The formal semantics provides a framework in which to prove properties about the language. In most cases, a fairly informal proof is sufficient to increase confidence in the correctness of a language feature, or to expose weaknesses (if the proof fails). We illustrate this with two examples: an informal proof that the evaluation process always terminates, and a summary of the more formal proofs which demonstrate that the store representation always contains unique identifiers.

7.1 Termination

One important property of a well-designed configuration language is that the evaluation is guaranteed to terminate, regardless of the input specification. This is not true for most non-declarative (configuration) languages since they usually support arbitrary computations. But it is not *trivially* true for declarative configuration languages either, since recursive references (for example) can potentially lead to no-termination (see the discussion in 9.2.1). The following theorem states that evaluation of a SF specification always terminates:

Theorem 1 (termination) *The evaluation SF*$[\![\,sf\,]\!]$ *of a (finite) specification sf* $\in$ *SFSpecification always terminates.*

Proof Since *sf* is finite and *data references* are not resolved by any of the valuation functions, then there are only two possible cases that can cause non-termination: a cyclic *link reference* and a cyclic *prototype*. Thus, it is sufficient to show that the evaluation will always terminate in these two cases.
Case 1: a cyclic *link reference* can occur whenever there is at least one unresolved *link reference* in the store. This is impossible because the function *Value* in definition 6.4 immediately resolves every *link reference* at the time the dereferenced value is bound to the target variable. Since cyclic *link reference* will be never exist, then the evaluation will always terminate.

Case 2: a cyclic *prototype* can happen whenever an object uses its parent as prototype. The proof that the evaluation will always terminate in this case, is given by example: consider the following:

```
1  sfConfig extends {
2    a extends sfConfig
3  }
```

The updates of the store during evaluation can be represented in the following series:

1. $\varnothing_{\mathcal{S}}$
2. $\langle sfConfig, \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}}$
3. $\langle sfConfig, \langle a, \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}}$
4. $\langle sfConfig, \langle a, \langle a, \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}}$
5. $\langle a, \langle a, \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}}$

(1) represents the initial store before evaluation, (2) is the store after evaluating line 1, (3) is the store when evaluating line 2 but before expanding the prototype, (4) is the store after expanding the prototype, and (5) the store after extracting the main object. This evaluation always terminates with the store in (5) being returned as the result.

Since the evaluation always terminates in both cases, then the statement holds.

7.2 Reference uniqueness

The choice of a list structure to represent the store (see §9.1) means that the following proofs are required to show that the uniqueness property of identifiers is maintained by the semantic functions. We omit the details of these proofs which are fairly straightforward – full versions are available in [23].

Lemma 1 (*put* uniqueness) *Assume s is a store that has unique identifiers i.e. $\forall \langle id_i, v_i \rangle, \langle id_j, v_j \rangle \in s \;.\; i \neq j \Rightarrow id_i \neq id_j$. Then operator $s' = put(s, id, v)$ always returns s' that also has unique identifiers i.e. $\forall \langle id'_i, v'_i \rangle, \langle id'_j, v'_j \rangle \in s' \;.\; i \neq j \Rightarrow id'_i \neq id'_j$.*

Lemma 2 (*bind* uniqueness) *Assume $s \in \mathcal{S}$ where s has unique identifiers, and $\forall s_i \subset_{\mathcal{S}} s : s_i$ has unique identifiers. Then operation $s' = bind(s, id, v)$ always returns s' that has unique identifiers, and $\forall s_j \subset_{\mathcal{S}} s' : s_j$ has unique identifiers as well.*

Theorem 2 (store uniqueness) *Assume a specification $sf \in SFSpecification$, if $SF[\![sf]\!] = s$ then s has unique identifiers and $\forall s_i \subset_{\mathcal{S}} s : s_i$ has unique identifiers.*

8 From semantics to compiler

In developing any formal semantics, there are choices to be made in both the representation of the domain, and the semantic operations. These will depend to some extent on the purpose of the semantics. Often, the representations will be rather abstract and too inefficient to be translated directly into code. But they

may be better suited to proving properties of the language (for example), and be more independent of the implementation technique.

Our formalisation of the SF semantics was largely motivated by the need for a new implementation of the compiler to support additional features related to the planning of configuration changes [23]. We wanted this to be compatible with the current production compiler (in the common features). We therefore chose a rather concrete representation. We were able to translate this fairly directly into a reference implementation. This means that it is easier to have confidence that the compiler is correct (with respect to the semantics), and the process itself is more familiar to users without a formal background.

8.1 Compiler implementation

The semantics was initially developed in parallel with the implementation of a demonstration compiler (written in Scala). In order to evaluate the results, two further compilers were then implemented independently, in different languages (Haskell and OCaml), by different people. These implementations were created directly from the semantics, without the use of any additional SF knowledge. All of the code for these compilers is available from [4].

The use of functional programming languages allowed us to make a very direct translation from the formal semantics into the core of the compiler code. Figure 5 show a typical evaluation function and its translation into Haskell. The compiler requires less than one page of code for the core semantics, and a further page for the supporting functions.

$$\mathbf{V}[\![\,\mathrm{LR}\,]\!] \quad \triangleq \quad \lambda\,(ns, r, s)\,. \\ \text{if } v' = \bot \text{ then } \boldsymbol{err_5} \text{ else } \mathit{bind}(s, r, v')$$

where $\langle ns', v' \rangle = \mathit{resolve}(s, ns, \mathbf{LR}[\![\,\mathrm{LR}\,]\!])$

```
1  evalValue (LinkValue lr) =
2    \(ns,r,s) -> do
3      (ns',v') <-
4        case (sfResolv(s,ns,lr)) of
5          Nothing    -> Left ENOLR(lr)
6          Just (n,v) -> Right (n,v)
7      sfBind(s, r, v')
```

Fig. 5: Example translation of a semantic function into Haskell code.

8.2 Compiler evaluation

To validate the correctness and compatibility of the compiler implementations, we created a framework to compile the same source files with multiple compilers and

compare the output. We tested both manually created, and randomly-generated specifications.

The manually-created tests were mostly small, targeted examples, designed to exercise specific features of the language - both syntactic, and semantic. These included tests for all expressible errors, as well as correct specifications (see appendix B for some examples). All test files are available with the compiler source [4].

Haskell's QuickCheck [17] package was used to produce randomly-generated, syntactically correct specifications for testing. However, almost all of these specifications initially included semantic errors such as references with missing or inappropriate targets. While it is useful to test for compatible handling of these invalid specifications, we wanted to generate a significant number of valid configurations. We therefore created custom QuickCheck generators which used the semantic functions as a guide to produce mostly correct specifications. As well as providing a further illustration of the application of the semantics, this created a large number of very complex, correct configurations (see appendix C for an example). We tested our implementations using multiple QuickCheck runs of 100 such examples.

This comparison process identified a small number of implementation errors. All but one of these were related to non-semantic issues: for example, a mistake in the parsing which failed to allow empty blocks, and differing interpretations of the `#include` directive (which is not formally specified). The one semantic error involved the ordering of store items, which is discussed below (§9.1). We consider this to be a very promising result.

We also used the above process to compare our compilers to the production SF compiler. However, the current production compiler supports a wider range of features, and a slightly different semantics (which requires multiple passes). This means that many of the automatic tests, were not appropriate. Nevertheless, this process identified some issues with the production compiler (including a non-termination problem). It also highlighted a misunderstanding of the semantics which we corrected in our semantics and implementations.

9 Discussion

The development of the semantics, and its subsequent use in the implementation, highlighted several interesting issues with the process itself, the SmartFrog language, and the design of configuration languages in general. This section discusses some of these in more detail.

9.1 Store order

It was initially very unclear whether the order of the items in the store was significant or not, and if so, how that order should be determined. If the order were not significant, then it would be more appropriate to represent the store in terms of sets, or more abstractly as a function $\mathcal{S} : \mathbb{I} \to \mathcal{V}_\perp$[3]. However, it appears that

[3] This is equivalent to a map data structure.

practical use of the language often relies on the item order, as determined by the current production implementation. We faithfully replicated this order in the semantics by using the list representation for the store. However, this complicates the semantic functions, and it necessitates some additional proofs (§7.2) to guarantee the integrity of the store.

In theory, it may be preferable to insist that the user specify the ordering where this is significant. However, it would still be desirable for an implementation to produce a deterministic ordering, in which case users may be tempted to rely on this, and different orders from different implementations would very likely to lead to portability problems. This suggests that a fully-specified order is preferable. This is discussed further in the context of a real example in section §10.

Technically, the JSON standard explicitly declares the order of the key-value entries to be indeterminate. However, we will continue to use this as a convenient format for representing the store contents, assuming an implied resource order equivalent to the order in which the the JSON attributes appear. YAML supports mapping types which preserve the key order.

9.2 References

The treatment of references is probably the most complex part of the SF language. There are several possible interpretations, and the choice among these affects the language in subtle ways:

- SF resolves references relative to the current component. It is only possible to refer to items in a sub-component – not in a higher-level component.
- SF supports references both on the right-hand side of an assignment (a *link reference*), and on the left-hand side (a *placement*).
- The production SF compiler currently supports "forward references". These are not supported by the semantics presented in section §6. The prototypes shown in figure 7 and figure 9, for example, are both illegal under our semantics. This is discussed in more detail below.

The discussion of references is a good example of the ability a formal semantics to highlight difficult areas of the language. The corresponding functions are more intricate, and the forward reference extensions complicate the semantics considerably. This makes the specifications more difficult for humans to understand, and hence more error-prone. It also leads to potential errors in the implementation. The use of the semantics to pinpoint these problems can be very helpful in designing improvements to the language.

9.2.1 Forward references

Technically there is no loss of functionality in prohibiting forward references, since mutually recursive prototypes are not necessary, and the source code can always be re-ordered to remove them. However: an important feature of configuration languages is the ability for different people to independently author different parts of a specification which are then composed into a final configuration. Most current languages have no explicit support for this, so it is typical for a specification to

simply include the text of sub-specifications written by other people (using `#include`). In many cases there may be no possible inclusion order which satisfies the requirement to define all prototypes before they are referenced. This dependence on ordering to define composition semantics is a common problem for configuration languages.

```
// server.sf
sfConfig:server extends {
  port 1234;
  ...
}
```

```
// main.sf
#include "server.sf"
sfConfig extends {
  client extends { ... }
  client:port server:port;
  ...
}
```

Fig. 6: Included files.

For example: figure 6 shows the file `server.sf` which is intended to be included in a main configuration to add some parameters specifying a particular kind of server. The main configuration also includes a client which is configured to have the same port as the server. There is no position in the main file where `server.sf` can be included without a forward reference (the example shown is not valid because `server.sf` makes a forward reference to `sfConfig`). This is one motivation for the forward reference support in the current production compiler.

9.2.2 Forward link references

```
sfConfig extends {
  a b;
  b 1;
  c a;
}
```

Fig. 7: A forward link reference.

Supporting forward link references requires an additional pass of the compiler and a significant increase in the complexity of the semantics. An extended version of the semantics which supports this is available in [23]. It is interesting that the

development of this extension identified an issue with the production compiler which failed to terminate on specifications of the form shown in figure 8. This has been corrected in the extended semantics, together with a corresponding extension to the termination proof given in theorem 1.

```
1 sfConfig extends {
2   comp1 extends {
3     comp2 comp1;
4   }
5 }
```

Fig. 8: Non-termination.

9.2.3 Forward placement

```
1 sfConfig extends {
2   a:b extends {
3     c 2;
4   }
5   a extends {
6     b 1;
7   }
8 }
```

Fig. 9: A forward placement.

Supporting forward placement is more difficult. The production SF compiler uses three passes to perform this and we have not attempted to provide a corresponding semantics. This also requires invalidating certain expressions such as figure 10 which are legal in the basic semantics provided above – since line 4 is now deferred to a later pass, "`a`" will have the value 1 rather than being a component by the time this line is evaluated.

The apparent difficulty of the semantics in this case is not a limitation of the technique, but rather a helpful indication that the process we are attempting to describe is inherently complex. This feature was added to the SmartFrog language to address some pragmatic issues, but it introduces an awkward, non-intuitive behaviour and highlighting this helps to motivate the search for a better solution to the original problem.

```
1  sfConfig extends {
2    a extends {
3      b extends { c 1; }
4    }
5    a:b:c 3;
6    a 1;
7  }
```

Fig. 10: Ambiguous forward-placement.

9.3 Inheritance

As noted at the end of section §3, there are several possible interpretations of prototype inheritance – see figure 11 and the resulting configuration in figure 12, for example.

```
1  p1 extends {
2    q1 1;
3    q2 2;
4    q4 extends { a 1; b 2; }
5  }
6  sfConfig extends {
7    p2 extends p1, {
8      q1 2;
9      q3 3;
10     q4 extends { b 3; c 4; }
11   }
12 }
```

Fig. 11: Prototype inheritance.

```
1  p2:
2    q1: 2
3    q2: 2
4    q4:
5      b: 3
6      c: 4
7    q3: 3
```

Fig. 12: Prototype inheritance evaluated.

The values of `q1`, `q2` and `q3` seem to be fairly natural, but `q4` demonstrates the shallow composition semantics defined by the *inherit* function (definition 18). Recursive composition of the sub-prototypes would be a plausible alternative semantics (i.e. `q4 { a: 1, b: 3, c: 4 }`), and there has been some discussion of adding a *merge* operator to the language to support this.

In the case of multiple inheritance, the order in which the prototypes are listed also determines the order in which they are composed (definition 23). This can lead to similar ordering problems to those noted in the previous section (§9.2.1).

9.4 Applicability to other languages

As we have noted, development of a formal semantics is easiest when the configuration language is small, well-defined and largely declarative. It is certainly questionable whether it is possible to develop a *useful* semantics for any language which relies heavily on exposing the underlying programming language, or on imperative features.

However, it is also necessary for the language to have a reasonably clear and explicit semantic domain (§5.1). This appears be the case for SmartFrog, Puppet[9] (the *catalog*) and LCFG[7] (the *profile*). Microsoft's recent release of a *desired state configuration* framework[8] based on PowerShell should be a particularly good example; this compiles down to a description in DMTF MOF[5] which is well-documented and explicitly designed as a target for multiple different tools.

One notable exception to this is the popular configuration tool cfengine[2]. This has no explicit intermediate form, and the specifications are interpreted directly, resulting in immediate changes to the actual state of the running system. This involves imperative operations (including arbitrary system commands), and the ability to read and write the persistent state of the entire machine (arbitrary files), as well as the state of the running processes. This makes it much more difficult to provide a meaningful (denotational) semantics for the cfengine language, despite its apparently declarative nature.

Some languages are more focussed on deployment, so that the translation of the source language into the intermediate form is less important than the deployment of the resulting configuration (the semantics of the deployment process is a different problem which we do not address). However, as systems become more complex, we expect the demand for richer descriptions and associated source languages to increase - this may involve extensions to existing languages or additional preprocessing (for example [31]) which should both be suitable candidates for a formal semantics.

We believe that the difficulty of developing a semantics correlates with the difficulty in understanding and predicting the system behaviour. So this is a useful indicator of the clarity and usability of the language, rather than a limitation of the technique.

10 A real-world example

The examples presented in the previous sections have been rather abstract, with the intention of isolating specific features of the language and not obscuring these

with unnecessary detail. In this section, we present a more realistic example which shows how the language might be used in practice, and how uncertainties about the semantics can lead to genuine confusion in the context of a large production system.

We have based this example on a real use case, but created a composite to illustrate a number of points simultaneously:

- The example is based on real configurations for a perimeter firewall, clearly illustrating the security implications of configuration errors.
- We have chosen to demonstrate the potential confusion caused by uncertainty over the store order semantics, since we initially misunderstood this aspect of the SmartFrog language ourselves, and we would therefore expect others to find this confusing.
- This example is based on real configurations which have been implemented using LCFG, rather than SmartFrog. However, this allows us to compare the handling of order semantics in the two languages and to suggest potential ways in which the SmartFrog semantics could be modified to reduce the potential for confusion.

10.1 The DICE iptables configuration

The School of Informatics at Edinburgh University uses LCFG to configure several hundred machines in a research environment (DICE) with very diverse configurations. Configurations for individual machines are constructed by composing many configuration classes which determine (among other things), the "holes" (protocol and port numbers) which each machine requires in the perimeter firewall. These are collated into a single specification of about 4500 rules which form part of the configuration of the firewall system itself.

The firewall system uses the `iptables` software[6] which is configured with this list of rules to define which connections from the outside world (by protocol and port number) are permitted to each of the machines on the internal network. The `iptables` rules support many options, but the basic principle is to *accept* or *deny* connections between specified sources and specified destinations. For example, the following rules would accept all (tcp) connections on port 22 to any host on the "production network" (defined by variable `PROD_NET`), and deny any attempt to connect to this port of any other machine:

```
1 -A INPUT -s ${PROD_NET} -p \
2    tcp --destination-port 22 -j ACCEPT
3 -A INPUT -p tcp --destination-port 22 -j DENY
```

Crucially, the order of the rules is important: the first matching rule in the chain is used, and the following rules are ignored. If a connection to the production network is permitted by the first rule in this example, then the second rule is ignored.

In the following examples, we will show only the significant rule options, in a simplified format. For example:

```
-s SOURCE-ADDRESS -p DESTINATION-PORT ACCEPT/DENY
```

10.2 Configuration classes

The configuration for a large system will typically establish a hierarchy of prototypes which can be inherited by individual machines depending on their function (class). For example, a particular service may be provided on two different ports, one for the "public" version of the service (defined by variable `PUB_PORT`), and a different one for a "private" (defined by variable `PRIV_PORT`), or test version. Figure 13 shows a prototype for a default configuration which blocks access to both of these from everywhere.

```
default extends {
  public "-p ${PUB_PORT} DENY";
  private "-p ${PRIV_PORT} DENY";
  ...
}
```

Fig. 13: A default prototype.

Different classes of machines may then override parts of this prototype, depending on their function. For example, figure 14 shows how a production server, may allow access to the production service (and inherit the default access to the private service):

```
prodServer extends default, {
  public "-p ${PUB_PORT} ALLOW";
  ...
}
```

```
prodServer:
  public: "-p ${PUB_PORT} ALLOW"
  private: "-p ${PRIV_PORT} DENY"
  ...
```

Fig. 14: A prototype for a production server with corresponding compilation output.

Figure 15 shows a prototype for a development server which may allow access to the private service, but only from machines on the "development network" (defined by variable `DEV_NET`), and inherit the default access to the public service:

```
devServer extends default, {
  private "-p ${PRIV_PORT} -s ${DEV_NET} ALLOW";
  ...
}
```

```
devServer:
  public: "-p ${PUB_PORT} DENY"
  private: "-p ${PRIV_PORT} -s $(DEV_NET) ALLOW"
  ...
```

Fig. 15: A prototype for a development server with corresponding compilation output.

Figure 16 shows the configuration for a test server which has specific requirements. Note that the public access is exactly the same as the default, so the configuration would be identical even if this line were removed. However, the intention here is clearly to make sure that public access is denied, *even if the default is later changed*. This is important since the second line permits access on all all ports from the development network.

```
testServer extends default, {
  public "-p ${PUB_PORT} DENY";
  private "-s ${DEV_NET} ALLOW";
  ...
}
```

```
testServer:
  public: "-p ${PUB_PORT} DENY"
  private: "-s ${DEV_NET} ALLOW"
  ...
```

Fig. 16: A prototype for a test server with corresponding compilation output.

Note that all of these prototypes are likely to include values for many other attributes. We have indicated this here with "...", but we will omit this in future for clarity.

10.3 Unintended consequences

Imagine that the default prototype is now changed so that access to the private service is permitted by default from the development network (figure 17).

Notice that public access continues to be denied by default, but the rule order has been changed to permit access to the public port from the development network.

```
default extends {
  private "-s ${DEV_NET} ALLOW";
  public "-p ${PUB_PORT} DENY";
}
```

Fig. 17: A new default prototype.

The new behaviour of the production and development servers is fairly clear, and seems reasonable: in particular, the rules specified in the `devServer` and `prodServer` prototypes are still honoured. However, the `testServer` prototype (which has not changed) explicitly specifies both rules, with the clear intention that these should take precedence over any defaults. But figure 18 shows the resulting configuration, which is now different from the previous configuration (figure 16).

```
testServer extends default, {
  public "-p ${PUB_PORT} DENY";
  private "-s ${DEV_NET} ALLOW";
}
```

```
testServer:
  private: "-s ${DEV_NET} ALLOW"
  public: "-p ${PUB_PORT} DENY"
```

Fig. 18: The configuration specification and the compilation output of the test server after the change to the default prototype.

The SmartFrog ordering semantics means that, although the *values* are overridden, the *order* is still inherited from the `default` prototype, so the attributes will now appear in the reverse order from that shown in the `testServer` prototype. This change in effective order is very non-intuitive, and in this case, produces a non-obvious security vulnerability – access to the public port is now allowed from the development network.

This may appear to be contrived example, but this is exactly the type of misunderstanding which can occur in practice, and can be very difficult to identify. And the context here has been considerably simplified – in practice, this would be part of configuration with thousands of firewall rules, a more complex inheritance hierarchy, and thousands of other parameters.

10.4 Mitigating the problem

As previously noted (§9.1), there is no obviously "correct" way to handle ordering. Generating a deterministic order from the composition of the prototypes will always lead to some cases such as the above which are extremely non-intuitive. A

non-deterministic order is equally problematic, because users may unintentionally come to depend on the specific ordering generated by a particular implementation.

However, the formal semantics clearly highlights problems such as this, providing an opportunity to mitigate them at the language design stage. For example, LCFG supports an additional meta-attribute which can be used to specify constraints on the (partial) ordering of the other attributes[14, 53-54]. Figure 19 shows how a similar feature might be incorporated into SmartFrog (using a hypothetical syntax).

```
1  default extends {
2    order private < public;
3    private "-s ${DEV_NET} ALLOW";
4    public "-p ${PUB_PORT} DENY";
5  }
6
7  testServer extends default, {
8    order public < private;
9    public "-p ${PUB_PORT} DENY";
10   private "ALLOW -s ${DEV_NET}";
11 }
```

Fig. 19: Hypothetical explicit ordering constraints.

The `order` attribute can be inherited in exactly the same way as the other attributes. The order of the attributes in the final configuration will conform to the corresponding ordering constraints, ignoring the order in which the attributes appear in the source (the order will be indeterminate where it is not constrained).

A lighter-weight approach may be to generate a warning (or even an error) from the compiler when the order of the compiled attributes is different from the order in which they appear in any particular prototype. However this may lead to small changes in the source producing many warnings, and may not identify more complex instantiations of the problem which result from the interaction of multiple inherited prototypes. Having a stronger type-system which defined the attribute order for any specific type would also clarify the issue, but would not permit the flexibility to change the ordering dynamically, as required in the above example.

11 Conclusions & further work

We have demonstrated that a declarative configuration language, developed using a more rigorous approach, can yield many of the benefits described in the introduction. In particular:

1. The formal semantics can be used to guide the practical implementation of a compiler for a real configuration language (§8).

2. This supports the creation of independent implementations with a high degree of confidence in their compatibility (§8.2).
3. And it enables the creation of other types of inter-operable tool, such as the compatibility testing tool (§8.2).
4. It also allows us to prove important properties of the configuration, increasing reliability and security (§7).
5. And the overall process leads to a deeper understanding of the language which exposes ambiguities, potential problems, and alternative interpretations (§9,§10).

It should be noted that the difficulties associated with the development of the formal semantics fall only on the language designer, and not on the user of the language who simply benefits from the improvements noted above. As configuration languages become more widely and heavily used, it becomes increasingly worthwhile to invest effort at the language design stage to improve the utility of the language for the end-user.

It has not been an explicit aim to evaluate the SF language itself – either in terms of design, or implementation. However, development of the formal semantics has considerably increased our understanding of the language, highlighted difficult areas, and identified problems with a production implementation. This has been a valuable guide to the practical development of language extensions and the corresponding compiler.

Although SmartFrog is a comparatively simple language, we have shown that apparently small extensions can have subtle, but significant consequences which can make the language more difficult to understand, error-prone, and complex to implement. There is a tendency for popular, practical configuration languages to change and acquire new features more regularly, in ad-hoc ways, and we suspect that these would benefit from a similar semantic analysis.

In the future, we would like to perform a similar analysis of other configuration languages. We are also planning to use the formal semantics as a basis to explore aspects such as provenance [15] in configuration languages.

Appendix A: Concrete syntax

SF	::=	B
B	::=	A B | ϵ
A	::=	R V
P	::=	R | `{` B `}`
PS	::=	P (`,` P)* | ϵ
V	::=	BV `;` | LR `;` | `extends` PS
R	::=	I (`:` I)*
DR	::=	`DATA` R
LR	::=	R
Vec	::=	`[` (BV (`,` BV)* | ϵ) `]`
Null	::=	`NULL`
Bool	::=	`true` | `false`
BV	::=	Bool | Num | Str | DR | Vec | Null

Appendix B: Manually created tests

The following examples show some of the manually-created test cases:

```
/*
 * syntactic features
 */
blob 34 ; a:b NULL; x23 "stuff"; _boolvar false;
p extends { q 2; } proto extends p:q
myref DATA x:y:zzz;
v [true,95,[1,2],"foo"]; // eol comment
```

```
// a test for references
sfConfig extends {
  A extends {
    A extends {}
    B 11;
    A:X extends { C 22; }
    D 33;
  }
}
```

```
// error 4
sfConfig extends {
        P "oops";
        Q extends P, { test 2; }
}
```

Appendix C: Auto-generated tests

The following SF code is part of a randomly-generated, semantically-correct specification used for testing compatibility between the implementations:

```
e extends {
  g 1234;
  e:g e:g;
  h e:g;
}
q e:g;
m false;
b extends {
  d e:g;
}, {
  b b:d;
  e:h DATA h;
  r "string";
}
g extends {}
e:g e:h;
e extends {}
b extends {
  x "noref";
  b:x extends {
    s extends {
      j 1234;
    }
    v extends {
      r extends {
        r b:x:v:r;
      }, {
        a extends {
          d extends {
            b true;
          }
        }
        h b:x:v;
        y 1234;
      }
      f extends b:x:v:r:a:d,
                    b:x:v:r:h
    }

... 200+ lines omitted ...
```

References

1. Ansible. http://ansibleworks.com. Accessed: 25th february 2014
2. Cfengine. http://cfgengine.com. Accessed: 25th february 2014
3. Chef. http://getchef.com. Accessed: 25th february 2014
4. Demonstration SmartFrog compilers based on the formal semantics. https://github.com/herry13/smartfrog-lang. Accessed: 28th march 2014
5. DMTF MOF Format. http://www.dmtf.org/sites/default/files/standards/documents/dsp0221_3.0.0.pdf. Accessed: 13th november 2014
6. IPTables. http://en.wikipedia.org/wiki/iptables. Accessed: 10th november 2014
7. LCFG. http://www.lcfg.org. Accessed: 25th february 2014
8. Microsoft Desired State Configuration. http://technet.microsoft.com/en-us/library/dn249912.aspx. Accessed: 13th november 2014
9. Puppet. http://puppetlabs.com. Accessed: 25th february 2014
10. Self. http://selflanguage.org. Accessed: 1st march 2014
11. SmartFrog. http://www.smartfrog.org. Accessed: 25th february 2014
12. YAML. http://www.yaml.org/spec/. Accessed: 10th april 2014
13. Aktug, I., Naliuka, K.: Conspec — a formal language for policy specification. Science of Computer Programming **74**(1–2), 2 – 12 (2008). DOI http://dx.doi.org/10.1016/j.scico.2008.09.004. URL http://www.sciencedirect.com/science/article/pii/S0167642308001056. Special Issue on Security and Trust
14. Anderson, P.: LCFG: a Practical Tool for System Configuration, *Short Topics in System Administration*, vol. 17. Usenix Association (2008). URL http://www.sage.org/pubs/17_lcfg/
15. Anderson, P., Cheney, J.: Toward provenance-based security for configuration languages. In: The 4th Usenix Workshop on the Theory and Practice of Provenance (2012)
16. Bekezhanova, A.: Denotational semantics of puppet. Master's thesis, School of Informatics, University of Edinburgh (2013)
17. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00, pp. 268–279. ACM, New York, NY, USA (2000). DOI 10.1145/351240.351266. URL http://doi.acm.org/10.1145/351240.351266
18. Classen, A., Hubaux, A., Heymans, P.: A formal semantics for multi-level staged configuration. Tech. rep., University of Namur (2009)
19. Couch, A.: From x-1 to (setf x 1): what does configuration management mean? login; **33**(1) (2008)
20. Couch, A., Sun, Y.: On observed reproducibility in network configuration management. Science of Computer Programming **53**(2), 215 – 253 (2004). DOI http://dx.doi.org/10.1016/j.scico.2004.04.002. URL http://www.sciencedirect.com/science/article/pii/S0167642304000796. Topics in System Administration
21. Damianou, N.: A policy framework for management of distributed systems. Ph.D. thesis, Imperial College (2002). URL http://www-dse.doc.ic.ac.uk/Research/policies/ponder/thesis-ncd.pdf
22. Favre, J.M.: CPP Denotational Semantics. In: Proceedings of the Third International Workshop on Source Code Analysis and Manipulation, pp. 22–None. IEEE Computer Society (2003). DOI 10.1109/SCAM.2003.1238028
23. Herry, H.: Automated planning for cloud service configurations. Ph.D. thesis, School of Informatics, University of Edinburgh (2014). URL http://homepages.inf.ed.ac.uk/s0978621/herry-thesis.pdf
24. Herry, H., Anderson, P.: Planning with global constraints for computing infrastructure reconfiguration. In: CP4PS-12 - The AAAI-12 Workshop on Problem Solving using Classical Planners (2012)
25. Hewson, J.A., Anderson, P., Gordon, A.D.: A declarative approach to automated configuration. In: Proceedings of the 2012 LISA Conference. Usenix Association (2012)
26. Hewson, J.A., Anderson, P., Gordon, A.D.: Constraint-based autonomic configuration. In: Proceedings of 2013 Self-Adaptive and Self-Organizing systems conference (SASO) (2013)
27. Hilty, M., Pretschner, A., Basin, D., Schaefer, C., Walter, T.: A policy language for distributed usage control. In: J. Biskup, J. López (eds.) Computer Security – ESORICS

2007, *Lecture Notes in Computer Science*, vol. 4734, pp. 531–546. Springer Berlin Heidelberg (2007). DOI 10.1007/978-3-540-74835-9_35. URL http://dx.doi.org/10.1007/978-3-540-74835-9_35
28. Oppenheimer, D., Ganapathy, G., Patterson, D.: Why do internet services fail and what can be done about it? In: 4th Usenix Symposium on Internet Technologies and Systems (2003)
29. Schmidt, D.A.: Denotational Semantics: A Methodology for Language Development. Allyn and Bacon (1986). URL http://people.cis.ksu.edu/~schmidt/text/densem.html
30. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. The MIT Press (1977)
31. Unruh, I., Bardas, A.G., Zhuang, R., Ou, Z., DeLoach, S.A.: Compiling abstract specifications into concrete systems - bringing order to the cloud. In: Proceedings of the 28th Large Installation Systems Adminstration Conference (LISA14), pp. 17–33. Usenix Association (2014)
32. Vervloesem, K.: FOSDEM Configuration Management Meeting Report (2011)

Authors

Paul Anderson is a Research Fellow with the School of Informatics at Edinburgh University where he is interested in applying knowledge from various areas in the School to the practical problems of large-scale system configuration. He is particularly interested in configuration languages, including provenance, security, and usability - but also in the deployment of configurations including automated planning, and agent-based approaches to distributed management.

Herry received the BSc degree and the MSc degree from Universitas Indonesia. He received the PhD degree in informatics from the University of Edinburgh. He is currently a research scientist in Hewlett-Packard Labs in Bristol, United Kingdom. His primary research interests concentrate on developing a reliable, trusted, and intelligent system configuration management for large-scale distributed system.