The SmartFrog Reference Manual

A guide to programming with the SmartFrog Framework

For SmartFrog Version 3.18

Localized for UK English / A4 Paper

Table Of Contents

Part1: An Introduction to SmartFrog

1 Introduction

This manual is aimed at those wanting to use and understand the workings of SmartFrog. It is not a basic tutorial, though hopefully it is not too obscure, either. The notation is described fully, as is the component model. The framework, however, is only outlined. For a detailed reference description of the framework APIs, users should refer to the accompanying *Javadoc* files.

The manual is divided into several sections:

1. The aims of the SmartFrog system: defining the basic goals of the system, thus ensuring that there is an awareness of these aims to aid in understanding the technical details.

2. The SmartFrog notation, describing the details and semantics of the first configuration description notation to be supported by the SmartFrog framework; other notations are in preparation but are not included in this manual.

3. The SmartFrog component model and framework, defining how to write components and run them within the SmartFrog system.

4. The SmartFrog security infrastructure, describing how SmartFrog ensures that systems are appropriately protected.

A separate document covers the details of installing and running the SmartFrog system. A number of examples are also provided and documented as part of the framework.

This document contains sections that assume differing levels of knowledge and familiarity with the SmartFrog system. It is suggested that a first-time user read only those parts that are essential before experimenting, then progressing to more advanced topics as familiarity develops. To aid in this, sections or sub-sections are tagged with one of the following labels: *basic*, *advanced* and *expert* indicating progressively more advanced topics. If a section is tagged as a particular level of complexity, and a sub-section is considered to be of higher level, the sub-section will be tagged with this higher level.

2 Aims Of The SmartFrog Framework

Configuration

For many years HP Labs has been involved in the development of large-scale distributed systems, and in particular management and measurement systems. From this experience, it became clear that configuration is often *the* major hurdle in the development, adoption and use of such large systems. This experience is supported by evidence from other domains, such as telecom service platforms, large scale e-service hosting environments, and so on. The weight of evidence clearly indicates that many of the problematic aspects of developing, delivering and maintaining such systems are resolved by the introduction of a well-designed, intuitive configuration system. These observations led to the development of the SmartFrog configuration framework described in this manual.

There are several significant reasons for investing in a powerful and flexible configuration environment, which in combination illustrate why this area is in many cases essential for the success of a large system. These are discussed below as a clear understanding of these reasons help in determining the requirements for a supporting environment.

2.1 Increased operational reliability

Configuration errors are the major cause of system failure. It is no coincidence that at least one system development inside of HP has termed the development of a tailored configuration system as its ‘high-availability programme’. It is pointless spending money on expensive replicated databases and computation if they contain wrong data, or are carrying out the wrong calculations. From hard experience, they know that the human element is by far the weakest point in any system of even moderate complexity.

Many systems are required to be resilient to a (small) number of failures, providing support for dynamic system reconfiguration in the case of such failures. This should be provided via failure detection mechanisms triggering re-configuration actions within the system components themselves (such as instigating fail-over) and through the configuration system to ensure a consistent view of the current configuration and to provide appropriate re-configuration policy (for example, where to create the replacement components in the case of a processor failure).

2.1 Improved quality

After examining the architecture and design of several large-scale systems it became clear that the developers of the various component sub-systems had each created their own configuration infrastructure, often not realizing that this area is of great importance to the overall system. Each makes separate decisions as to format of the data, how it is stored, and so on. In addition, since some aspects of configuration such as configuration description or failure detection and recovery can be extremely complex, the separate development groups frequently do not utilize best practice.

2.1 Reduced cost

Costs can arise for several reasons and in several areas such as development, installation and maintenance. For each of these, providing well-defined best-practice procedures and well-implemented support environments for configuration can save significant time and hence money. From experience with several systems, the majority of support calls for these systems (and hence source of recurring cost to the platform provider) come from configuration issues.

2.1 Assured correctness and consistency

Validation rules need to be provided to ensure that a configuration is correct before it is deployed into the running system. These rules should include dependencies between various system components (e.g. version dependencies) as well as rules governing repetition (e.g. each web server should run the httpd process and …), replication (e.g. two cooperating instances of this component should exist for reliability...), location (e.g. this component should be close to the database...), and so on. Tools for modelling and reasoning about the configurations are required.

Given a configuration that has been defined and validated, the configuration must then be correctly and verifiably instantiated, preferably automatically, with appropriate error handling in the case of failure. Discovery services must be present to enable binding of services to each other as defined in the configuration, and status monitoring capabilities are required to provide management tools with the ability to monitor the overall state of the system and to ensure it is correct with respect to the desired configuration.

Complex systems may in fact be impossible to configure manually if the requirements change faster than individuals ability to track these changes and carry out the complex reconfiguration tasks. In these cases, automated, adaptive configuration, driven from general rules and auto discovery, is the only solution.

2.1 Increased security

System configurations are vital to the integrity of the system. Consequently, in many environments where physical and network isolation cannot be guaranteed, a high level of basic system security must be provided. This involves not only protecting the configuration data itself from unauthorized access, but also the run-time environment must be secure. This includes discovery protocols, component instantiation services, management services and so on. It is typically hard to provide a secure environment when many independent and diverse techniques are used to provide the configuration, so again a single solution implementing best practice is an essential step to ensure system integrity.

2.1 Improved Customer Experience

A major issue to be considered in designing systems is that different classes of user have different requirements. All too frequently, the configuration information is designed for the convenience of the system developer not the system operator. Data is required in a form that often does not reflect the skills of the administrator, or maybe is replicated in several files, or distributed over many processors, each of which can lead to a slow and error-prone configuration process. Configuration should be done in ways useful to the operator and adapted to the system and not by expecting the operator to adapt. This can be expensive and hard to implement unless there is extensive support for the systems developers.

The SmartFrog Framework

SmartFrog is a framework for the development of configuration-driven systems. It was originally designed as a framework for building and managing large monitoring systems where flexible configurations are essential. SmartFrog is currently in use within several products, though it is not a product in its own right.

The name reflects its basic design concept – the Smart Framework for Object Groups. It defines systems and sub-systems as collections of software components with certain properties. The framework provides mechanisms for describing these component collections, deploying and instantiating them and then managing them during their entire lifecycle.

The framework consists of three major aspects:

1. The SmartFrog configuration description environment, consisting of a description notation and tools to enable the storage, validation and manipulation of these descriptions.

2. The SmartFrog component model, defining the interfaces that a software component (or a management adapter for a component) should implement. These interfaces are to support the various lifecycle operations such as creation, versioning and termination, as well as management actions such as accessing status information.

3. The SmartFrog configuration management system, which uses these descriptions and management adapters to instantiate the software components and to monitor them throughout their lifecycle in a secure way, including an integrated run-time environment providing capabilities such as discovery and naming.

Notation

The SmartFrog 'notation' is in fact defined as a set of open data structures. In principle, this definition can support a number of parsers that provide different textual versions of the notation (for example using XML as a surface syntax). Additionally, it's possible to develop GUI tools that allow the users to "drag-and-drop" their configurations using the data structures as the common form. At this stage, no generic GUI tools are available for SmartFrog, though experimental versions have been built; usually such tools are normally best tailored to a specific class of system.

The notation is object-oriented, supporting inheritance and extension of configuration descriptions. These descriptions consist of component definitions, associations and relationships between the components, and workflows associated with the lifecycle of the components and the system as a whole. The descriptions may be parameterized enabling multiple instantiations with different configuration data, and validations may be provided which verify that these instances are correct before an attempt is made to deploy the configuration.

The current version of SmartFrog, though in principle able to support multiple textual languages, just provides its own specialized notation "out-of-the-box". Others are in preparation for future releases.

The notation is not used to define behaviour, merely the structure of collections of components and their relationships with other collections. It is not a programming language. The behavioural part of a component is assumed to be defined in an existing programming language (such as C or Java) and the component will be started as needed by the SmartFrog configuration management system. Currently only Java is tightly integrated. Java adaptors must be used to wrap code written in other languages, and these are relatively simple to implement.

Components

The component model supported by SmartFrog is a simple, extensible set of interfaces providing access to key management actions – such as instance creation, configuration, termination, and so on. A component may be fully integrated (i.e. it may implement the defined management interfaces directly, and hence be written in Java) or it may be independent, in which case a management adapter must be provided. Several standard management adapters or base integrated components have been written to provide common behaviours and these may be extended or modified as appropriate.

Each component (or adapter) must implement a standard lifecycle, implemented as a set of action routines that the environment invokes in the appropriate order and at the right time to carry out the configuration or other management task required. The lifecycle process is governed and controlled by the definition of workflows within the SmartFrog system to provide a very flexible and adaptable environment for carrying out the various configuration tasks.

A complete set of APIs is available to the components that allow them to access the configuration information, locate other components as defined in the configuration and to alter the running configuration if so desired.

Environment

The SmartFrog configuration and management infrastructure is supported by a collection of services, such as:

- deployment – the distribution of code, configuration data and the instantiation of components in the right place with certain 'transactional' guarantees

- discovery and naming – providing a number of binding services to allow components to locate each other and communicate

- management – every component is manageable via tools provided with the framework, via the web, or other consoles (if so configured) with no developer effort.

These services are incorporated so as to provide a seamless and coherent programming and configuration model. The benefits of this approach are in providing configuration abstractions to component developers that allow multiple configurations of different scale to be produced without altering the components in any way. The environment is broken into several well-defined functional units, each of which has some specific role to play. Furthermore each of these operates through well-defined and open interfaces, so it is easy to replace the existing functional units, or even to make the selection of which functional unit to use part of the configuration description.

For example, suppose a component, say an SS7 stack, requires the use of another, a real-time database for storing connection information to help the recovery process in the case of system failure. This may be done in many ways. For example, the database could name itself under some well-known name in some well-known naming service, and the stack could find it there. Alternatively, the system may use SLP discovery to locate the database, or perhaps look in a file for this location information. Each approach has advantages in different system contexts, but the programmer typically has to decide up front which to support.

Not so with the SmartFrog integrated environment. The SmartFrog system supports the notion of a binding and provides multiple ways – determined by the environment and driven by the configuration descriptions – for these bindings to be resolved. This includes all the above approaches and others may be added as required. So a programmer need only obtain its binding from the environment and the precise mechanism is handled by the SmartFrog environment as defined by the configuration.

SmartFrog is a framework, and is designed to make it easy to provide additional binding mechanisms as they are required – for example changing the naming service or adding a specialized binding service which uses some other technologies such as databases or directories.

This is equally true of the other services. Consider deployment; it is possible to provide different mechanisms for ensuring that a component is created in the right place. For instance, it might be by hostname, or perhaps by some computer's role within the system, or perhaps it needs to be close to another existing component. Each of these location mechanisms may be integrated into the run-time environment and then referenced freely within the configuration descriptions.

Final Comments

The design goals for SmartFrog were to produce a very lightweight and flexible configuration and management infrastructure capable of scaling from small systems to very large. This has been achieved through the use of the framework concept and providing users with the ability to alter the low-level semantics by replacing functional units, yet providing standard capabilities by offering default implementations of these units. The system also provides a flexible configuration description notation, with potential for multiple textual or GUI syntaxes to be used targeted at specific system architectures.

Applications of SmartFrog have clearly demonstrated that systems are more quickly implemented using the technology, and that the structure imposed upon the implementations by the use of SmartFrog is beneficial to long-term reliability, usability and manageability.

3 The Anatomy of SmartFrog

This section attempts to lay out the main aspects of the SmartFrog service deployment framework, describe their relationships, and map them into the structure of the reference document.

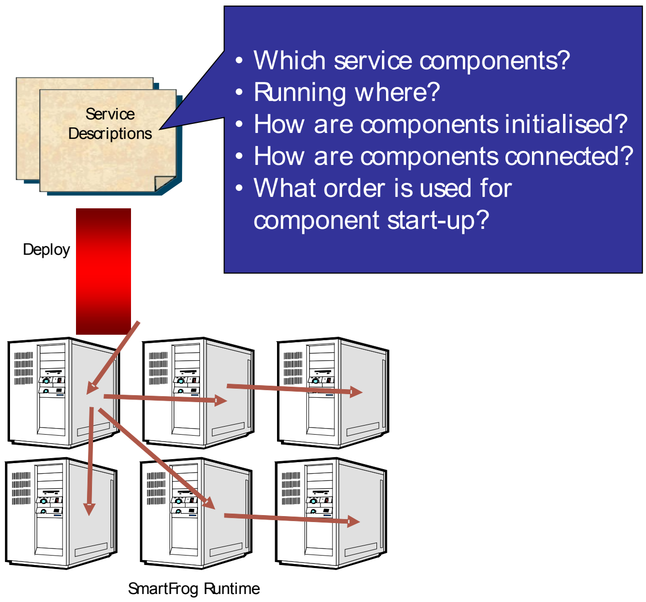
As described in the introduction, it consists of three main aspects:

1. The SmartFrog notation, a language in which to describe the configurations, also known as service descriptions.

2. The SmartFrog component model, the way in which programmers create components that are created and managed by SmartFrog as part of a service and which can interact with the system. These are deployed according to the service description.

3. The SmartFrog runtime, the collection of services that exist as part of the SmartFrog system. This is also know as the deployment engine, but is strictly a misnomer since it is in reality a collection of predefined components.

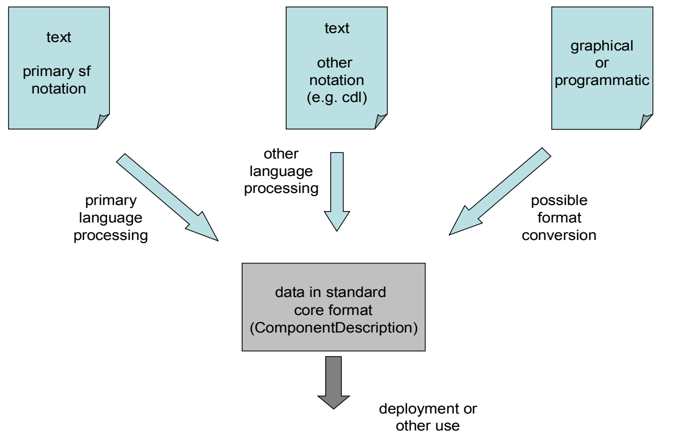
These various components can be seen from the following outline diagram of a SmartFrog system.



Languages and Language Processing

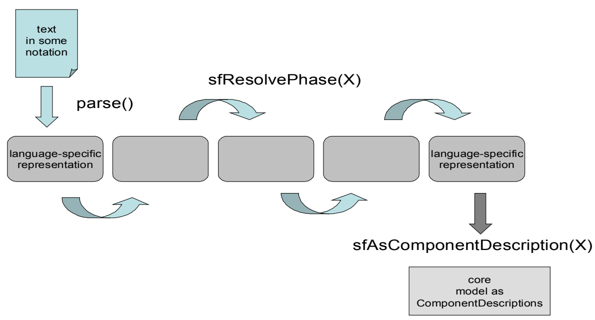
The statement that there is a SmartFrog notation is a simplification of reality. SmartFrog may support many notations, though it provides a 'standard' primary notation out of the box. To enable this, SmartFrog provides a well-defined interface between the language processing parts of SmartFrog and the run-time as a well-defined data model: the set of Java classes that must be used by the language processing to represent the data delivered to the runtime system.

Roughly speaking, the model of SmartFrog language handling is shown in the following diagram:



As is illustrated in this diagram, there may be many notations, each with their own language processing, which at the back-end of that processing produces an instance of the data model that can be understood by the remainder of the SmartFrog system. Alternatively, programmes such as a drag-and-drop gui can produce the data in the correct form directly.

To support the development and use of additional languages, the SmartFrog framework provides a rudimentary structure for integrating language processors. A language processor is assumed to consist of three major steps: parsing, executing some processing phases, and then conversion to the standard data format. The set of processing phases are assumed to be language specific, including having the empty set of phases.



This is illustrated in the diagram above, also showing the associated Java calls used within the framework. These are not important at this stage and are explained in detail later in the reference manual.

Note that the core data mode and the primary notation are closely coupled. This means that in effect the core model can in some ways be seen as a true subset of the primary notation – it could be unparsed into the primary notation and parsed back directly into the core form without requiring any language processing.

Indeed, the two are sufficiently close that the Java classes that are used to directly represent parse-trees of the notation are derived from those of the core model, and much of the same terminology is used in both. So for example, an attribute-set in both is called a *Component Description,* the only difference being that in the primary notation this may have a super-type from which it inherits, whereas in the core model it may not.

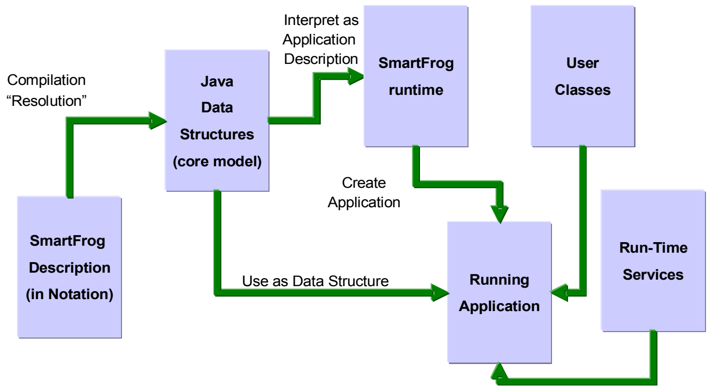
Each notation is assumed to have an associated name, and this name is used in the construction of a parser (selected via a standard language-name to parser-classname mapping). Furthermore, if text files or URLs are handled by the SmartFrog system, the extension associated with that file is assumed to indicate the name of the notation in use. Thus for the primary notation, files must end with ".sf".

Once converted to the core format, the data represented may be used in several ways:

1. It can be data that is passed to components in the same way as any other data. Indeed many of the components provided as part of the Smart frog distribution exchange such data through their APIs.

2. It can represent the set of components that should be deployed by the SmartFrog run-time.

Now the second case is in fact just a special case of the first, where the data is passed to one of the standard SmartFrog 'Compound' components, such as the `ProcessCompound`, that understands how to interpret these descriptions as that of a distributed set of components. This duality is described in the following diagram:



The reference manual the primary notation and the core data model.

1. The primary notation is covered in section REF. This is the only notation covered in the reference manual.

2. The programming model for interacting with the language framework is given in section REF. This provides details of how to invoke a parser for a specific notation, how to drive the phase-resolution steps of the language, and finally how to covert to the standardized form for handling within the rest of the SmartFrog framework.

3. The core data model is described in section REF. This is only a partial description and the primary source of this information should be the Javadoc for the classes involved.

Components and Deployment

TO BE PROVIDED

4 Building Systems with SmartFrog

TO BE PROVIDED

Part1: The SmartFrog Notation (sf)

5 BNF conventions

Note that the document is best printed in colour as the syntax descriptions makes use of colour to highlight certain aspects. The following conventions will be used:

1. examples of constructs will be given in dark green over a grey ground as in:

```
//This is an example of the use of SmartFrog
sfConfig extends Prim {
    sfClass "org.smartfrog....;
    attr1 42;
}
```

2. syntax definitions will be given in

   dark blue: non-terminals

   red delimited by <...> : terminal lexical classes

   red bold: terminal symbols

These will be over a grey ground as in:

```
//This is an sample of SmartFrog BNF
AttributeList::=  (Attribute
                  | #include String
                  | #include? String
                  | ;                  // allow arbitrary extra ";"
                  )*
```

The BNF descriptions will be given with the following fairly standard meta-syntax

```
::=   a non-terminal  introduction

( )   grouping

*     0 or more instances of the immediately preceding element or group

+     1 or more instances of the immediately preceding element or group

?     0 or 1 instance of the immediately preceding element or group

|     choice
```

Note that in the discussion of the syntax, the clauses provided are a simplification of the complete syntax – sometimes also transformed for clarity – as the underlying parser generator for the language sometimes requires a convoluted form of presenting that grammar. The grammar as implemented is provided in section REF, and this should be consulted if any specific detail of the grammar is required, for example to help understand some of the parser errors.

6 Comments

The SmartFrog notation follows most modern languages in providing both end-of-line comments and multi-line bounded comments. The syntax for these is identical to that of Java, namely

```
// this is a comment to the end of the line

/* this is a comment which is terminated by */
```

7 Attributes

A SmartFrog description consists of an ordered collection of attributes. The attributes are ordered because several of the operations in the SmartFrog framework require an order: an example being the order in which a configuration should be instantiated. Each attribute has a name, a value, and an optional set of tags.

The value is either a simple value (integer, Boolean, string, etc.), or an ordered collection of attributes known as a component description. This recursion provides a tree of attributes, the leaves of which are the basic values or empty component descriptions. A value may also be provided by reference to another attribute.

This is described by the following BNF, where *Stream* indicates the entry point to the SmartFrog parser.

```
Stream::= AttributeList

AttributeList::=  ( Attribute
                  | #include String
                  | #include? String
                  | ;                  // allow arbitrary extra ";"
                  )*

Attribute::=   Tags? Name Value

Tags::= [ <WORD> (,? <WORD>)* ]        // the , separator is optional

Name::=   --  | (<WORD> [ : Name ])

Value::= Component
       | SimpleValue ;
       | ;                             // instance of SFNull
```

From this it is clear that the input to the parser is a collection of attributes, each named and having an optional value and an optional set of tags. If the value is not present, the value is defined to be an instance of the class `SFNull` (note that the other way of defining a value of class `SFNull` is to use the basic value `NULL`). The reason for providing this feature is to enable the use of attributes where the presence of the attribute is what is important, not its value. If the tags are not provided, the set of tags is empty.

The syntax for a name will be covered later, but for now it can be considered to be either a simple sequence of letters and digits, starting with a letter, or the double-hyphen "`--`"). The double hyphen is for use at times when the attribute name is not important and so a new unique name is generated and used. This is particularly useful with the function syntax described in REF, and most specifically the n-ary operators.

Include files are covered in more detail in REF, but in general they consist of parseable SmartFrog text which are parsed as attribute lists and unpacked into place within the container attribute list.

Values can be divided up into two main categories: nested attribute sets (components) and the rest (simple values) which include numbers, strings, vectors of these, and so on. In addition it is possible not to provide a value for the attribute, or more precisely to give a null value to it (an instance of the `SFNull` class). This is captured by the third clause of the BNF for values above.

Tags are simply meta-data associated with the attribute (not its value). They may be used for any purpose, but there are some pre-defined tags that have specific meaning in the context of the SmartFrog language. The use of tags, and their semantics, is covered in REF.

Note that attributes may be defined as having values that are provided “late”, that their value is not available at the time of definition but will be provided programmatically at some point in the future. This is known as LAZY binding in SmartFrog and is an important aspect of the language design. It is covered in detail in section REF, but it should be kept in mind that this is possible when considering the various ways that attribute values can be defined.

8 sfConfig

A stream contains a whole collection of attributes at the top level. Most are merely there to act as building blocks – prototypes for building others. Typically, there is only a single attribute that is the essence of the description – that which describes the desired configuration and is not merely a building block on the way. By convention in SmartFrog, the reserved attribute name *sfConfig* defines this special attribute and all the tools provided respect this convention.

Thus, when a stream is parsed to an attribute set, the top-level attribute *sfConfig* defines the system; the rest are ignored, apart from providing definitions for extensions and other resolutions. This is equivalent to the Java language use of the "special" method *main* to indicate the entry point to a program. The entry point to a configuration description is *sfConfig*.

Thus in the following example representing the contents of a file, the attributes *def1*, *def2* and *def3* are only present for the purposes of defining *sfConfig*, and it is only this last attribute that represents the actual configuration description.

```
def1 extends {...};

def2 extends {...};

def3 extends {...};

sfConfig extends {
    d1 extends def1;
    d2 extends def2;
    d3 extends def3;
};
```

Note that since `sfConfig` is the *meaning* of a description file, it is only this attribute whose well-formedness predicates (REF) are checked for validity and value expressions (REF) are evaluated. Verifying or fully evaluating the other top-level attributes, which are just partially defined templates, would make no sense since they are by intent incomplete.

9 Simple Values

Values are expressible in several syntactic forms.

```
SimpleValue::= Basic
             | TBD
             | Function
             | Predicate
             | LinkReference

Function ::= CanonicalFunctionForm
             | Operator
             | IfThenElse
             | Vector

Predicate::= CanonicalPredicateForm
```

Basic Values

The primary way is to provide a basic value, a literal syntactic form for the basic core values in the SmartFrog language. The syntax for the basic values is best given by example.

```
Integer:       345

Long:          65325L or  65325l

Float:          34.76F or
               34.76f or
               34.76E-10F or
               34.76e+10f or
               34.76E10f

Double:        1534.45 or
               1534.45D or
               1534.45d or
               1534.45E10 or
               1534.45E-10D

String:        "this is a string"

Multi-line String: ## This is a string
                       Over many lines   #
Boolean: true

SFNull: NULL   // alternatively, leave the value empty

Byte Array: #HEX#AB348eAb#

ReferenceValue: DATA x;   // a reference to the attribute x
```

Consequently, an example of a piece of SmartFrog text is as follows

```
portNum 4074;
hostname "ahost.smartfrog.org";
isHighPriority false
```

defining three attributes with the appropriate values.

In addition to these basic values, it is also possible to give vectors of basic values (as opposed to the more extensive vector syntax given below). These vectors are limited to containing basic values, and other vectors of basic values.

```
userList [| "fred", "harry" |];
empty    [|  |];
listOfLists  [| [| 1,2,3 |], [| 4,5,6 |] |];
```

The full syntax for the basic values is

```
Basic::= String
       | Number
       | Boolean
       | ByteArray
       | ReferenceValue
       | [| (Basic (, Basic)*)? |]
       | NULL
```

```
Number::= <DOUBLE>
        | <FLOAT>
        | <INTEGER>
        | <LONG>

String::= <STRING>          // "....."
        | <MULTILINESTRING> // ##....#

Boolean::=  true | false

ReferenceValue::= DATA BaseReference

ByteArray::= #HEX#....#
           | #DEC#....#
           | #OCT#....#
           | #BIN#....#
           | #B64#....#
```

Note that byte arrays will be definable as hexadecimal (HEX), decimal (DEC), octal (OCT), binary (BIN) and base64 (B64). However B64 is currently not implemented. Depending on the definitional form, the characters that may be used and the number that must be present are different. White space characters are ignored so that neat tabbed layouts may be used. They are treated in the syntax as single tokens.

TBD

The `TBD` value is used to indicate that a specific attribute still requires to be assigned a value. If it has not been assigned, and an attempt is made to use it, an appropriate error message is given.

An example of the predicate is as follows:

```
#include "/org/smartfrog/predicates.sf"

aTemplate extends Prim {
    sfClass "org.smartfrog....";
    anAttribute TBD;
}

sfConfig extends Compound {
   anInstance extends aTemplate;        // TBD not overridden
   anotherInstance extends aTemplate {
       anAttribute 45;                  // TBD overridden
   }
}
```

Here, the attribute *`anAttribute`* of *`aTemplate`* is defined as *`TBD`*, so any use of the template that does not set this value will generate an error. In the definition of *`sfConfig`*, the first use, to define *`anInstance`*, is erroneous whereas the second to define *`anotherInstance`* is valid.

Note that a TBD which is not overridden is only checked within the main `sfConfig` attribute.

Link References and Reference Values

A reference used in a value context normally refers to a value defined elsewhere as described in section REF. These are known as *link references* as they link one attribute to another. Link references may also be used in functions and assertions. As such, they are represent, and are replaced by, the value to which they refer within the expressions in which they are used. This substitution happens as part of the language processing: an activity known as *link resolution*.

However there are other times when it is necessary to define the reference itself as the value. In these cases, the value is defined as follows:

```
ReferenceValue::= DATA BaseReference
```

The keyword *`DATA`* preceding the reference definition indicates that the following reference *is* the value and is *not* a reference to another value. The full base reference syntax described in section REF may be used to create the reference value. A reference value is considered a basic value in that no further processing (resolution) is required to determine the value. Thus in the following:

```
x 10;
y DATA x;
z x;
```

the value of *y* is a reference to *x*, whereas the value of *z* is the same as that of *x*, namely *10*.

Functions

Using functions allows a SmartFrog user to provide more complex expressions to define the value of an attribute. Functions have several syntactic forms, some which are more convenient to the user, and one of which is the canonical form of function application – the internal form into which all other forms of function syntax are eventually mapped. The simplest syntactic forms are described here. The more complex form – the templated form and the canonical form – are left to section REF.

9.1 Operators

The remaining three forms of value definition are syntactic sugar for the use of functions. The semantics of functions are outlined in section Error: Reference source not found and described in detail in section REF.

Some set of functions may be represented as prefix or infix operators to match the normal mathematical and programming language conventions. The syntax is somewhat simplified over the usual practice to avoid issues of operator precedence and associativity.

```
Operator::=
       (
           (UnaryOp SimpleValue )
         | (SimpleValue ( BinaryOp SimpleValue )?)
         | (SimpleValue ( NaryOp SimpleValue )*)
       )

UnaryOp::= !

BinaryOp::= - | / | == | != | >= | > | <= | <

NaryOp::= + | * | ++ | <> | && | ||
```

This states that the use of an operator is always defined within brackets (...) and that there are three types of operator: unary, binary and nary. Although with the nary operators, more than one instance of the operator symbol is present, it must always be the same operator; they cannot be mixed. However, other operators may be nested within another set of ( ) within the expression. The following examples may help to make the syntax clear:

```
aTruthValue    true;
anotherValue   (! aTruthValue );      // the only unary operator: not
aNumber  45;
aMinus   (100 - aNumber);             // a binary operator
aMix   (100 - (aNumber + 5));         // nesting operators
aString ("the meaning " ++ "of life is " ++ 42); // an n-ary operator
aSum     (aNumber + aMinus + 100);    // another n-ary operator
```

These operators are all converted at time of parsing into the canonical representation of a function, and hence at no time will operators appear in an description generated from the parsed form.

Note that attribute names can contain rather a large number of special symbols, such as “+” and “-”. This means that there is a danger that an operator may lexically stick to a name if not separated from it by white space. As a consequence, it is good practise to always use white space around operator symbols.

9.1 If-Then-Else

Similarly to operators, if-then-else expressions are shorthand for the application of a function in the canonical form. This is described in detail in section REF. The syntax for if-then-else expressions is

```
IfThenElse::= IF SimpleValue
              THEN SimpleValue
              ELSE SimpleValue
              FI
```

The line breaks being, of course, optional. The “if” value is a boolean and depending on the result the expression takes the value of the “then” or “else” values. The `FI` is merely a closing keyword. An example of its use is:-

```
val1 42;
val2 43;
diff   IF (val1 > val2) THEN (val1 - val2) ELSE (val2 - val1) FI;
```

Note that unlike some programming languages, the evaluations of the `THEN` and `ELSE` values are both carried out. Of course, since the SmartFrog language is side-effect free this normally doesn't matter. However it is possible for users to add their own functions which are not so (and indeed there are one or two provided by SmartFrog that have minor side effects), or for there to be an error in the non-selected part, and in either case the fact that all parts are fully resolved is clearly important.

9.1 Vectors

The final form of simple value is the vector. Vectors are lists of values and are constructed using the vector function described in section REF However, to simplify its use, the following syntactic form has been provided.

```
Vector::= [  (SimpleValue ( , SimpleValue) *)?  ]
```

Thus a vector is a sequence of values separated by “,” and delimited by “`[ ]`”. If no value is provided within the vector, an empty vector is returned. Vectors may be nested to produce vectors of vectors. Example uses of vectors are:

```
v1  [1,2,3];
v2  [9,8,7];
v3  [v1, v2];       // same as [[1,2,3],[9,8,7]]
```

Note that there are two syntaxes for vectors – the one given here which provides the ability to embed references and which therefore requires a degree of processing (known as resolution). It is parsed into the use of the vector function rather than directly into a vector. The other form, using the “[| |]” delimiters as described in section REF, parses directly into a vector and hence may not have references within the definition.

The reason for having simpler primitive form `[|...|]` in addition to the more general vector function form `[...]` is is that there are times when the fully processed (resolved) data structures need to be unparsed and then re-parsed at some future time without rerunning language processing. A example of this is during the signing of a description for security purposes. The primitive form is parsed directly into a fully processed form.

Clearly using the primitive form is more efficient in the cases that it suits the requirements, but it is also much more restrictive in its use. It is never wrong to use the more general form whenever the language processing will be run in its entirety.

Predicates

Predicates are rather different to normal attributes, in that they provide a validation over attribute values rather than a value in itself. They can be viewed as Boolean-valued attributes, but ones which cause a failure if their evaluation does not result in "true". There are two forms of predicate – assertions and schemas.

A complete discussion of predicates is provided in section REF.

Values of Other Classes

The set of values that can be described by the use of the language is limited to a few basic classes and collections of these. It would be useful to be able to include values from other classes in Java. These in principle can be generated in functions, or some user-defined phase, and added to the attribute sets. However, there are problems with this for SmartFrog, and in particular with some aspects of the security where descriptions transformed to core form need to be signed and this is restricted to the known classes.

Consequently the conversion to the core form ensures that the values represented in the attribute sets, the component descriptions, are limited to these core classes. If other values need to be held within the tree, it is recommended that they are held in serialized form within a `ByteArray` value. This will need to be de-serialized at the time of use.

10 Component Descriptions

A file contains a list of attributes. Furthermore, attributes may have values that are also collections of attributes. Collections of attributes are known as *component descriptions*. They obtain their name from the fact that they may be interpreted by the SmartFrog framework as the description of a software component, though they may equally and more simply be used to describe structured data.

At the top level, i.e. at the level of the parsed file or stream, some of the syntax associated with providing a set of attributes does not need to be provided: it is represented merely as a list of attribute definitions. For attributes whose values are defined to be an attribute set, the syntax is much richer and provides a capability similar to inheritance called prototyping.

A component description is defined by providing a sequence of contributing attribute sets known as the prototypes. Prototypes can be specified in one of two ways: a reference to another component description to act as a prototype source of attributes, or a collection of attributes provided explicitly to act as a prototype.

Note that the syntax described here is a slight simplification of the full syntax as it is somewhat complicated by the provisions for backward compatibility with earlier versions of SmartFrog. In this first part of the section, however, the preferred syntax will be described followed at the end of the section by the more complete form with examples of its use.

```
Component::= extends (DATA)? Prototype (, Prototype)* ;

Prototype::= ( BaseReference | {  AttributeList } )
```

The `DATA` keyword may be largely ignored when considering the language; it is merely a boolean flag and only has a semantic effect during the deployment of a SmartFrog application. Extension of a `DATA`-flagged component description does not inherit the flag.

A component description is defined as formed from a list of prototypes. Each prototype is a list of attributes either defined explicitly or through reference to another component description which is copied to provide a set of attributes.

Simple Example

Consider the following:

```
aService extends {        // a list of attributes
    portNum 4047;
    hostname "ahost.smartfrog.org";
    administrators ["patrick"];
};

sfConfig extends aService, { // an extension of the previous component
    portNum 4048;            // override the definition of portNum
    users ["fred", "harry"];  // add a new attribute
};
```

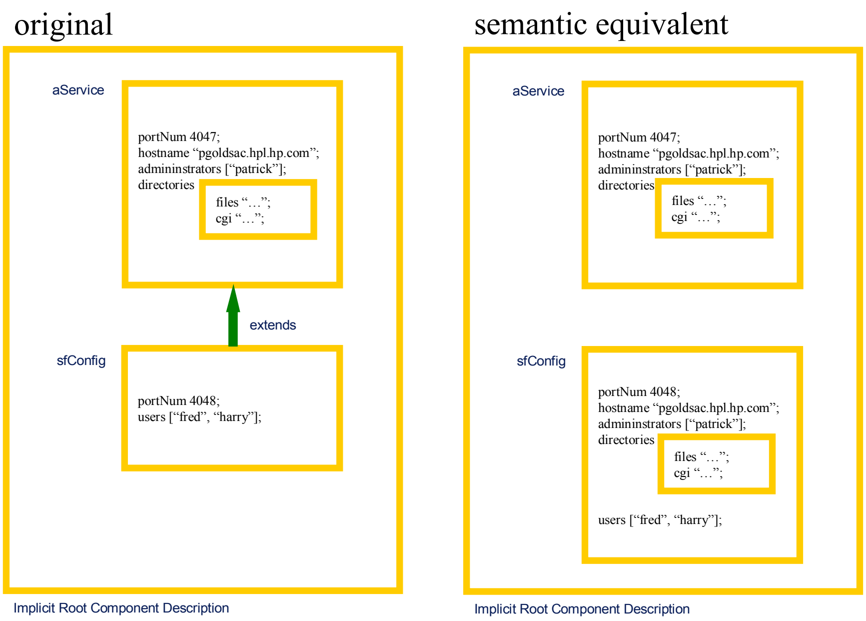
The text consists of two attributes, both of which have values that are collections of attributes. The first component description, `aService`, is defined explicitly as the given set of attributes. The second, `sfConfig`, is defined as an *extension* of the first, `aService`, with two attributes that are explicitly provided.

The semantics of extension is that a new component description is created, and the attributes of the first prototype is added to it, then the attributes from the second set are also added – either overriding existing attributes if the names are identical or new attributes being added to the end.

So the semantics of the example above is identical to:

```
aService extends {
    portNum 4047;
    hostname "ahost.smartfrog.org";
    administrators ["patrick"];
    directories extends {
        files "/home/www/files";
        cgi   "/home/www/cgi";
    }
};

sfConfig extends {
    portNum 4048;
    hostname "ahost.smartfrog.org";
    administrators ["patrick"];
    users ["fred", "harry"];
};
```

Note that the attributes in a component description are ordered and that when an attribute is overwritten it maintains its position in the first (extended) component description, but when it is a new attribute it is added to the end in the order defined by the second attribute set. The process of expansion of the inheritance in this way is known as *Type Resolution* (REF).



The example is also shown in the diagram. It clearly shows that there are two kinds of relationship between component descriptions. One is the containment relationship , where a component description contains an attribute that is itself a component description (`ROOT` contains `aService` which contains `directories`). The second is the inheritance or extension relationship (`sfConfig` extends `aService`). This second class of relationship is one that can be transformed, by type resolution, to an equivalent simple set of attributes.

Whilst the extension relationship is merely a convenient way of defining attributes, the containment hierarchy is a more fundamental construct. It should be noticed that containment hierarchy effectively provides a naming scheme by which attributes may be referenced. In this it is similar to other such named hierarchies, such as directory hierarchies common in files systems.

The Empty Component Description

Notice that the empty set of attributes can be defined as follows:

```
emptyList extends { };
```

There are other ways of describing this set (REF), but the above is now the preferred syntax and the other forms are deprecated.

Multiple Extension

The syntax defined above allows a list of prototypes that contains many prototypes. Each prototype may be an explicit set and or a reference. The semantics of this more general form is an obvious generalisation of the simple example above – where `sfConfig` is defined with two prototypes, one a reference and the other explicit.

Consider

```
Threads extends {
    maxThreads 10;
    minThreads 1;
    valid extends Assertions {
        minMax (minTreads <= maxThreads);
    };
};

Memory extends {
    memSize 128;
    unit "MB";
    pinned false;
};

Disk extends {
    volSize 1;
    unit "TB";
    type "NTFS";
};

SystemDescription1 extends Threads, {
        minThreads 2;
    }, Memory, {
        memSize 256;
    }, Disk;

SystemDescription2 extends Threads, Memory, Disk, {
        minThreads 2;
        memSize 256;
    };
```

The two `SystemDescription` descriptions define the same thing, but in a slightly different way. The semantics of multiple extensions of this form are that :

- a new *working* component description is created
- iterating through the list of prototypes from first to last (be they references to previously defined component descriptions or explicitly provided attribute sets)
- the attributes of each are taken in order and either
  - override a previous attribute definition of the same name in the working component description (thus maintaining previous order)

  or

  - are added to the end of the component description.

The ROOT component description

Note that since the parsed stream or file consists at the top level of a set of attributes that this set is no different in concept to any other set of attributes contained in a component description. Consequently the top level is considered to be an implicit, anonymous (i.e. not named in an outer component description) component description known as the ROOT component description.

Full Component Description Syntax

The syntax is in fact more complex than described above. There are a number of forms supported for backward compatibility which are now largely deprecated.

The first is a couple of alternative ways of describing an empty set of attributes. Nether of these are particularly recommended, and the second is most definitely not so.

```
emptyList extends;

emptyList extends NULL;
```

The first of these simply provides no prototypes and so by definition denotes the empty set. The second uses the `NULL` keyword in an extension context to indicate the empty set whereas in a simple *value* context it indicates the value `SFNull` (REF). This confusion is not recommended, so the use to indicate an empty set is now deprecated.

The other syntactic complication is to support the original syntax for extension previous to supporting multiple extension.

- When a reference was followed by an explicit set they were space separated, there was no need for a comma (,).
- If the extends clause had the explicit set, there was no need terminate the extension by the semicolon(;).

Thus the normal form for extension used to be

```
component::= extends (BaseReference)? ({ AttributeList } | ;)
```

In order to support the very many examples of this form that exist, the following extended syntax is allowed:

"If the last prototype in the list is an explicit attribute list (not a reference), then both the preceding comma and trailing semicolon may be left out. If the comma is present, the semicolon must be present."

This allows the following examples:

```
A extends {
    x 10;
}     // optionally no ;

B extends A {
    y 20;
}     // no , before last so optionally no ;

C extends A, {
    y 20
};    // , present so ; must be too

D extends A, B, {...}, C {
         z 40;
}     // it doesn't matter how many prototypes are in the list
      // the last can still have the , and ; missing.
```

The syntax is basically straight forward even if the rules are hard to express.

The final backward compatibility syntax is that the keyword `LAZY` can be used instead of `DATA` for a component description (note that for references these two keywords mean different things, but for component descriptions they mean the same). This use of the `LAZY` keyword is most definitely deprecated.

The complete syntax as implemented by the parser is:

```
Component::= extends (DATA | LAZY)? BaseComponent

BaseComponent::= ;
               | NULL                         (BaseComponentRest | ;)
               | BaseComponentLinkType        (BaseComponentRest | ;)
               | BaseComponentAttributesType (BaseComponentRest)?


BaseComponentRest::=
         , (    BaseComponentLinkType        (BaseComponentRest | ;)
              | BaseComponentAttributesType (BaseComponentRest | ;)
              | NULL                         (BaseComponentRest | ;)
           )
         | BaseComponentAttributesType


BaseComponentLinkType::=      LinkReference

BaseComponentAttributesType::= { AttributeList }
```

11 References

References are "pointers" from one part of a description to another (or to something outside of the description itself (REF)). References may occur in three places in the syntax:

- as the name of an attribute – known as a placement reference, pointing to where an attribute should be placed
- as a reference to a component description to be used as a prototype of another component description – known as a prototype reference,
- and as an attribute value referring to another attribute whose value is to be used – known as a link reference.

The primary purpose of a reference is to indicate a value or component by providing a path through the containment hierarchy defined by the components. In this, it is similar to the notion of path common in file systems in operating systems such Linux. A path defines a traversal of the directory hierarchy, a structure similar to the component hierarchy.

The underlying syntax for references is as follows:

```
BaseReference::= ReferencePart ( : ReferencePart)*

ReferencePart::= ROOT
               | PARENT
               | ATTRIB <WORD>
               | HERE <WORD>
               | THIS
               | <WORD>
               | ...
```

Note that some of the reference parts are discussed in section REF when considering late binding as they only make sense in this context.

The syntax states that a reference is a colon-separated list of parts each of which, for the parts described here, indicates a step in the path through the containment tree defined by the hierarchy of component descriptions. Examples of references are:

```
PARENT:PARENT:foo:bar
a:b
ROOT
x
ROOT:x:y
```

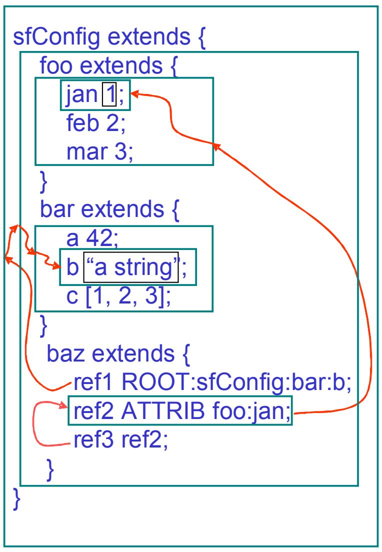
Normally a reference indicates a path through the containment tree to an attribute whose value should be copied, or a component description in which an attribute should be placed. These references are "resolved" during the language processing to eliminate them and to carry out the appropriate copying or placement.

The general rule for the interpretation of a reference is that the reference is de-referenced in a context (a component description somewhere in the description containment tree), and that each step moves the context to a possibly different component for the remainder of the reference to be de-referenced. This is equivalent to path evaluation in a Linux file system, the path is evaluated in a current directory, and each part of the path moves the context to another directory.

The semantics of each of the reference parts is as follows: starting at component in which the reference is defined…

- **PARENT** - move context to the parent (container) component if it exists, fail otherwise (c.f. Linux "..")
- **HERE** WORD - look for the attribute named "word" in the current context, fail otherwise
- **ATTRIB** WORD - look for the attribute named "word" in the current context or anywhere in the containment hierarchy (the closest is chosen), move to the context defined by this attribute, fail if no attribute is found in the containment hierarchy
- **ROOT** - switch context to the outer-most component description(normally the implicit root component description (c.f. Linux "/ ")
- **THIS** – keep the context the same, don't switch (c.f. Linux ".")
- WORD – the interpretation of the WORD depends on the location. If it is the only part in the reference, or the first part, it is interpreted as `ATTRIB`. If it is the second or later part of a reference it is interpreted as `HERE`.

Some examples of references (in this case link references) are as follows:




```
sfConfig extends {
    foo extends {
        jan 1;
        feb 2;
        mar 3;
    }
    bar extends {
        a 42;
        b "a string";
        c [1, 2, 3];
    }
    baz extends {
        ref1 "a string";
        ref2 1;
        ref3 1;
    }
}
```

The arrows in the left-hand text show the path followed as the references are resolved to obtain the referenced attribute values, noting that the resolution of *ref3* will follow the resolution of *ref2*. The contexts traversed as the resolutions progress are shown boxed and the right-hand text shows the result of resolving the three links.

The above rules determine the general interpretation of references. However, each of the syntactic contexts has its own slight semantic variation; these variations appear in the detailed definition of the semantics for references.

Prototype References

References to prototypes, as defined in the following syntactic context,

```
Component ::= extends (DATA)? BaseComponent
BaseComponent ::= (BaseReference)? ( ; | { AttributeList } )
```

are resolved as described above. The following synthetic example demonstrates most of the situations:

```
Foo extends { a 1; };

Bar extends {
    foo extends Foo;
};

Baz extends {
    foo extends {
        b 2;
    };

    foo1 extends Foo;  // recall - this is equivalent to ATTRIB Foo
    foo2 extends ROOT:Foo;
    foo3 extends PARENT:Foo;
    foo4 extends PARENT:PARENT:Foo;
};
```

After type resolution, which includes the merging and overwrite of attributes as described in section REF, the example is equivalent to:

```
Foo extends { a 1; };

Bar extends {
      foo extends { a 1; }   // ATTRIB Foo finds the outermost
};

Baz extends {
      foo  extends { b 2; };
      foo1 extends { b 2; }; // ATTRIB Foo finds the closest enclosing
      foo2 extends { a 1; }; // ROOT:Foo finds the one in the root
      foo3 extends { b 2; }; // PARENT:Foo finds that in the parent
      foo4 extends { a 1; }; // PARENT:PARENT:Foo finds that in
                             // the root (in this case)
};
```

Placement References

An attribute's name may be a reference, as described in the syntactic clauses

```
Attribute ::= Name Value
Name ::= BaseReference
```

This is not completely accurate, as the syntax in fact limits references to being a reference containing `WORD` parts only, the other reference parts are considered erroneous.

The resolution of the reference is again largely as described above, with the following modification.

The last reference part of the reference is treated differently. This final word part is not strictly part of the reference, but is used to identify the name of an attribute that is to be created or modified (as opposed to referenced) in the context of the prefix part of the name reference. Thus in the attribute definition

```
foo:baz:bar 42;
```

the `foo:baz` is a reference to a location, `bar` is the name of the attribute to be created in that context.

In most cases, the name consists only of that final *`WORD`* leaving the prefix reference empty, indicating the current context. Thus, the attribute is defined in that current context. Where a non-empty reference prefixes the final word, the reference is used to determine the appropriate context and the attribute with the given name is placed into that context.

Consider the example

```
Service extends {
    portNum 4089;
```

```
};

Service:portNum 4074;
Service:hostname "ahost.smartfrog.org";
```

The prefix reference `Service:` is de-referenced to indicate the Service attribute. The two prefixed attributes are therefore placed within that reference context, overriding or being placed at the end of the context as appropriate. Thus, the example is roughly equivalent to the following (there are some differences in their behaviour as prototypes):

```
Service extends {
    portNum 4074;
    hostname "ahost.smartfrog.org";
};
```

The act of placing the attributes into a location is known as placement resolution, and it occurs simultaneously with the removal of the reference-prefixed attribute from its defining context.

Placement of attributes can lead to a great deal of confusion if not used properly. It reacts in interesting ways with type resolution; this interaction explained in REF.

Link References

Frequently, attributes need to take on the same values as other attributes. This can be for many reasons:

- to avoid repetition of values at many points in a description making it easier to maintain that description
- to hide the structure of the description to a program; explained further in REF.
- to provide a means of simple parametrization; explained further in the REF.

Syntactically, link references are identical to other references apart from

- the additional optional flag `LAZY` which indicates that the reference is late bound. This is described in REF and for the remainder of the section can be ignored.
- The provision of an `OPTIONAL` clause which allows a link reference to have a value if the attribute to which the link refers does not exist.

The full syntax is therefore

```
LinkReference::=  (OPTIONAL ( SimpleValue )? (LAZY)? BaseReference
```

So an association between the value of one attribute and that of another is defined by providing a reference in the place of a value of the attribute or indeed as part of an expression containing operators. This reference is resolved relative to the context at the point of definition.

Consider the following example, in which a server and a client both need to know the TCP/IP port on which the server will listen.

```
System extends {
    server extends {
         portNum 4089;
    };
    client extends {
         portNum server:portNum;
    };
};
```

The system contains a server and a client. The server and client both have an attribute `portNum`, with that of the client being defined as a link to that of the server.

There is a resolution step, known as link resolution, which replaces references by the values that they reference. During the resolution phase, chains of links are resolved appropriately.

In the above example, the definition of `System` is equivalent to the following:

```
System extends {
    server extends {
         portNum 4089;
    };
    client extends {
         portNum 4089;
    };
};
```

Consequently, both the server and client share the same value and maintenance is eased in that should the port number need be changed, this need happen in only one place in the description.

Optional links are used when it is convenient to provide a default value for an attribute if an attribute on which it depends does not exist. The *OPTIONAL* can occur wherever a link occurs, including in an operator application.

```
sfConfig extends {
   x 10;
   y 20;
   sumxyz (OPTIONAL(0) x +
           OPTIONAL(0) y +
           OPTIONAL(0) z  );
};
```

Note that the use of `OPTIONAL` can be dangerous as it can hide a miss-spelling of an attribute name which will now return a default value as opposed to returning a language processing error. However there are a few circumstances where it is necessary to use this feature. Note that optionality also works with late bound `LAZY` references.

12 Tags

Tags are a simple way of adding basic meta-data to an attribute. This meta-data is in the form of a set of simple words prefixing the definition of the attribute. Note that the meta-data is not associated with the value but only the attribute, thus it is not carried with the value when an attribute's value is resolved through references.

```
Attribute::=   Tags? Name Value

Tags::= [ <WORD> (,? <WORD>)* ]       // the , separator is optional
```

Tags may be used for any number of purposes determined by the user of the language. However, all tags beginning with the prefix "sf" should be considered reserved for future use by the SmartFrog system itself.

Tags are maintained through inheritance as illustrated in the following example.

```
System extends {
    [optional, core]   component1 extends {...};
    [compulsory, core] component2 extends {...};
    [peripheral]       component3 TBD;
};

mySystem extends System {
    [optional] component3 extends {...};
};
```

In this example, the definition of mySystem is equivalent to

```
mySystem extends {
    [optional, core]       component1 extends {...};
    [compulsory, core]     component2 extends {...};
    [optional, peripheral] component3 TBD;
};
```

for some meaning of optional, etc. defined by the user. Note that the order of tags is not relevant as the structure is a set.

To illustrate the fact that tags belong to the attribute, not the value, consider the following.

```
mySystem extends {
    [server] port 80;
    webServer extends {
        serverPort port;
    };
};
```

In this case the definition of serverPort is the value 80, but without the tag, so the definition above is equivalent to:

```
mySystem extends {
    [server] port 80;
    webServer extends {
        serverPort 80;
    }
}
```

again for some user-defined meaning of the tag *server*.

The tags may be accessed programmatically through APIs provided for the purpose, see REF.

Pre-defined Tags

The SmartFrog system provides a number of tags with specific semantics.

12.1 The sfLocal Tag

The `sfLocal` tag provides a simple scoping capability – references may only refer to an attribute tagged `sfLocal` from within the same defining context (the same Component Description), or from one contained within it.

More accurately, the rule is that the attribute must only be referenced by the first part of the reference – but that effectively limits access through a "`HERE`" or "`ATTRIB`" reference part as the first part of the reference.

Consider this highly synthetic example:

```
sfConfig extends {
     one extends {
        [sfLocal] foo 10;
                  bar foo;              // OK - foo first part
        [sfLocal] baz extends {
                                x foo;  // OK - foo first part
                      [sfLocal] y 20;
                  };
                  bongo baz:x;          // OK - baz first part
                  bingo baz:y;          // Not OK - y not first part
     };

     two extends {
         myFoo one:foo;                 // Not OK - foo not first part
     };
};
```

There is one limitation in the way the rule is currently implemented as an interpretation of locality and that is when an inner context needs an indirection (typically `PARENT`) to reference the attribute the attribute is not visible as shown below. This may be fixed in a later release.

```
sfConfig extends {
    [sfLocal] port 80;
              server extends {
                  port PARENT:port;    // Not OK - port not first part
              };
};
```

12.1 The sfTemp Tag

The `sfTemp` tag is provided as a recognition that it can be very useful to use temporary attributes to hold intermediate values or temporary parameters, but that these have no place in the final data. Any attribute tagged *`sfTemp`* will be removed as part of the language processing.

```
httpURL extends {
    [sfTemp] URLPrefix "http://";
    [sfTemp] URLPostfix "/index.html";
    [sfTemp] URLBody TBD;
    theURL (URLPrefix ++ URLBody ++ URLPostfix);
};

myURL extends httpURL {
    URLBody "smartfrog.org";
};
```

This will result in the definition of `myURL` being

```
myURL extends httpURL {
    theURL "http://smartfrog.org/index.html";
};
```

without the attributes `URLPrefix`, `URLPostfix` and `URLBody` being present.

12.1 The sfFinal Tag

The `sfFinal` tag indicates that the value of this attribute is not allowed to change further – either through placement or through overwrite in an `extends`. Consider the following example.

```
WebServer extends {
    [sfFinal] port 80;
              directory "/webPages";
```

```
};

sfConfig extends {
    server1 extends WebServer {
        directory "/myPages";      // OK, not final
    };
    server2 extends WebServer {
        port 8080 ;                // NOT OK, port is final
    };
};
```

Note that the interpretation of `sfFinal` with an attribute whose value is a component description needs further explanation. That attribute may not be set to another value (either to a component description or any other value), but an attribute *of* that component description may be set to a different value unless it too is tagged `sfFinal`. Consider

```
System extends {
    [sfFinal]someData extends {
        x 10;
    };
};

mySystem0 extends System {
    someData 1;                 // NOT OK, changing value of someData
};

mySystem1 extends System {
    someData extends {...};     // NOT OK, changing value of someData
};

mySystem2 extends System {
    someData:x 20;              // OK, not changing someData itself!!
};
```

Whether this is considered a feature or a bug depends on point of view – perhaps two different tags are required – however in the mean time it is a fact worth knowing.

13 Include Files

A stream of text may reference include files at certain points in that text. Unlike a C include file, though, the include file is not merely textually embedded into the original stream. Rather the include file is itself parsed (and must be syntactically correct) as a stream in its own right. Every stream must parse as a collection of attribute definitions, and this is equally true of the include files.

Include files may only be used within attribute lists (i.e. at the top level or within a component definition). The collection of attributes from the include file are simply added to the attribute list being parsed in the container stream.

Consider the following example:

- file `foo.sf` contains:

```
foo extends {
    a 42;
}
```

- the primary stream is:

```
#include "foo.sf"
sfConfig extends {
    myFoo extends foo;
    #include "foo.sf"
}
```

After the parsing is complete (but before type resolution), the following is obtained:

```
foo extends {
    a 42;
}

sfConfig extends {
    myFoo extends foo;
    foo extends {
        a 42;
    }
}
```

It should be noted that because includes may occur within other component descriptions, this may be used as a naming mechanism to prevent clashes of attribute name within multiple include files. Consider

- file `foo1.sf` contains

```
foo extends { a 42; }
```

- file `foo2.sf` contains

```
foo extends { b 42; }
```

- the primary stream contains

```
foo1 extends { #include "foo1.sf" }
foo2 extends { #include "foo2.sf" }
sfConfig extends {
    bar extends ATTRIB foo1:foo;
    baz extends ATTRIB foo2:foo;
}
```

If the includes had not been buried within separately named components, but both had been included into the top level, only the second of the two mentioned `foo` attributes would have been available for extension. The second would override the first.

If the include file may not exist, the include may be made optional in which case the file is included if it exists, or is **silently** ignored if it does not. Optionality is indicated by adding a ? to the end of the #include keyword.

```
#include? "foo1.sf"
```

Loading Include Files

#codebase stuff

Embedding Other Languages

#language stuff

Pre-defined include files

The SmartFrog system provides a number of include files containing the templates for the core system and for each of the pre-defined component sets that are provided, such as components for managing shell scripts, web servers, and so on. Here, only the include files that cover the core features are described. Those for each of the documents are described within the individual component documentation.

There are three main include files:

- `/org/smartfrog/components.sf`
- This include file contains all of the definitions required for writing descriptions that will be deployed on the SmartFrog run-time. It also recursively includes the following two files so that these do not need to be included as well.
- `/org/smartfrog/functions.sf`
- This file includes the definitions of all the templates required to use the functions in the SmartFrog language.
- `/org/smartfrog/predicates.sf`
- This file includes all the definitions of all the templates required to use the assertions and schemas within the SmartFrog language.

14 Late Binding: LAZY and LAZY propagation

Frequently in the world of configuration management, the situation arises where some aspects of a configuration can only be determined at run-time from local context and cannot be determined statically and the data provided in a configuration file. Examples of this might be the IP address of a host set through DHCP, the location and/or port of some service, the remote object reference (RMI or otherwise) of some component, or perhaps even the host operating system if we are trying to provide descriptions that work on many such operating systems. Consequently it is important to be able to state which data will be bound late and how this is to be found.

To complicate the situation, suppose that other configuration data will be determined based upon this late-bound data – perhaps through the use of an operator. It is clear that this data also cannot be determined statically and should therefore also be declared as late-bound. SmartFrog terms the notion of late binding a *lazy binding*, and the propagation of this late binding to dependant attributes *lazy propagation*.

Consider the following:

```
File extends {
    filename "aFilename";
    fullPath ("/tmp/" ++ filename);
}
```

In this example both the `filename` and the `fullPath` can be determined from the data given. However, now consider the next example, where a host name is late bound – in this case to be provided by the property *`theFilename`* in the runtime system using the *`PROPERTY`* reference part which returns the value of a Java system property. (This and other reference parts useful with late bindings are described below.)

```
File extends {
    filename LAZY PROPERTY "theFilename";
    fullPath ("/tmp/" ++ filename);
}
```

In this case neither attribute can be evaluated until runtime – the attribute `filename` has been explicitly stated as late bound and this property propagates to the attribute `fullPath`. Note that if the property did not propagate automatically it would be impossible to provide a definition that worked both statically and at runtime depending on the situation, as in:

```
sfConfig extends {
   s1 extends File { filename "foo.txt"; };
   s2 extends File { filename LAZY PROPERTY ...; };
};
```

If `fullPath` did not inherit the propagated late binding property, it would be impossible to define `File` correctly. If `fullPath` were defined as statically bound it would be wrong in the second case, and if defined as late-bound it would at very least be less efficiently handled. Indeed without propagation the language would have to leave all resolution of expressions and links to run-time “just-in-case”.

Note that `TBD` does not mean the same as `LAZY`, in that any remaining occurrence of `TBD` is an error stating that some attribute has been overlooked, whereas there may be many explicitly stated late bindings.

Full Reference Syntax and Late-Bound Reference Parts

Syntactically the language features that can be declared as late bound are all link references and the internal canonical forms for functions and assertions. These latter two are normally generated through propagation as they rarely occur in user descriptions.

The full syntax for references is

```
BaseReference::= ReferencePart ( : ReferencePart)*

ReferencePart::= ROOT
               | PARENT
               | ATTRIB <WORD>
               | HERE <WORD>
               | THIS
               | <WORD>
               | <CONSTANT> (<WORD> | STRING)
               | PROPERTY <WORD>
               | IPROPERTY <WORD>
               | HOST (<WORD> | STRING)
               | PROCESS
```

Note that it does not make sense to have a data reference be late bound as it is a value and not a reference to be dereference in-situ. Consequently it is not valid to have both the flags present.

The additional reference parts presented here all make sense only when used with the late binding. In addition to the structural reference parts described in section REF (*ROOT, PARENT, THIS, HERE* and *ATTRIB*), there are five others that are not appropriate for all circumstances and are not related to the containment hierarchy. They are generally of most use with the binding. These are

- **CONSTANT** (WORD | STRING) – extract the value of a static field of a class (normally final, unless LAZY). The syntax is normally to follow the keyword with a dot-separated fully resolved classname followed by the field name, such as in

  ```
  CONSTANT "org.smartfrog.sfcore.SFSystem.aField";
  ```

  if not marked as lazy, this filed value will be resolved by the parser and its class-loading context, otherwise in that of the component within which the reference is being resolved.

- **PROPERTY** WORD – return the value that is the Java system property named WORD. It may only occur at the end of a reference, and only in a link. Syntactically it may occur anywhere in the link, however the remainder of the link is ignored. It is usually used in conjunction with late binding. Without being marked as `LAZY`, the value of the property at the time of parsing will be used; with `LAZY` the application run-time value of the property will be used when the link is resolved – see section REF. A property value is always a string, and the `PROPERTY` reference part dereferences to that string.

- **IPROPERTY** WORD – as for `PROPERTY`, but the property string is interpreted as indicating an integer which is parsed and returned as such.

- **HOST** (WORD | STRING) – switch to the context of the process compound on the host name WORD (or STRING – which must be used if supplying an IP address, but may also be used with a host name). This reference part really only makes sense with late binding and is described in greater detail in section REF as part of the run-time system.

- **PROCESS** – switch to the context of the process compound of the current process. This reference part really only makes sense with late binding and is described in greater detail in section REF as part of the run-time system.

LAZY Link References to Component Descriptions

There is an interesting, extremely common and very important use case for late bound `LAZY` references: these being references to component descriptions that are converted at run-time to SmartFrog components as described in REF.

Consider the following example.

```
sfConfig extends Compound {
    server extends Prim { ... };
    client extends Prim {
        myServer server;
    };
};
```

In this case, the attribute `myServer` refers to the description of the server, and this description is copied to be the value of the attribute as part of the language processing. Note that this might result in a second server component being created during the SmartFrog deployment process if the description is used for this purpose.

Compare this with

```
sfConfig extends Compound {
    server extends Prim { ... };
    client extends Prim {
        myServer LAZY server;
    };
};
```

in which the `myServer` attribute is defined as `LAZY`. In this case, when the value of attribute `myServer` is inspected at run-time (after SmartFrog deployment), its value is not the description, but rather the Java Object Reference of the component that implements the server description and was created as part of the deployment process. This is described much more fully in REF.

Just for completeness, compare this further with

```
sfConfig extends Compound {
    server extends Prim { ... };
    client extends Prim {
        myServer DATA server;
    };
};
```

Now when the value of `myServer` is inspected at run-time its value is not the Java Object Reference for the implementing component, nor is it a copy of the description to which the reference points, but it is the SmartFrog reference itself that is the value.

15 Functions and Operators

SmartFrog provides users with a small number of predefined functions to improve the expressiveness of the descriptions. In addition, it provides mechanisms by which users may add their own functions, effectively providing an escape mechanism into Java by which users may easily customize the way in which attribute values may be specified. However this section only discusses the way in which functions are used within the language. Providing new functions in Java is discussed in REF.

There are three ways in which functions may be invoked in SmartFrog.

- Operator syntax which is parsed directly into the canonical form for function invocation.
- A component description form which is transformed into the canonical form for function invocation as part of the language processing (the function phase).
- The canonical form for function invocation which is rarely used directly by users, but is used internally and the syntax is used whenever a description is unparsed. This form may occur in error messages so is worth understanding.

Canonical Function Form

Clearly, to understand the way in which functions are defined in SmartFrog it is necessary to understand the canonical form. The syntax for this is as follows:

```
CanonicalFunctionForm::= (DATA)? (LAZY)? APPLY { AttributeList }
```

There is one key attribute within the defined attribute list, and that is the attribute which specifies the class name of the Java class that implements the function's semantics. The other attributes define the parameters of the function. Note that since every parameter is named and the order known, the function class can choose to identify specific parameters

- by name (and ignoring order).
- the order of occurrence (and ignoring the names).

The different functions provided as part of SmartFrog use different models. So the unary and binary operators use specific names, whereas the n-ary operators use order. It might even be conceivable to have a function that uses order, attribute names and attribute values in its semantics – however none of the built-in operators or functions do so.

The attribute which defines the function class to use is `sfFunctionClass`. As an example, the following expression

```
result1 (1 + x + y);
```

is parsed directly into the following equivalent canonical form

```
result1 APPLY {
   sfFunctionClass "org.smartfrog.sfcore.languages.sf.functions.Sum";
   -- 1;
   -- x;
   -- y;
};
```

and

```
result2 (10 - x);
```

becomes directly

```
result2 APPLY {
   sfFunctionClass "org.smartfrog.sfcore.languages.sf.functions.Minus";
   left 10;
   right x;
};
```

These two examples show the difference in the binary and n-ary operators and the way they use order or naming to indicate the parameters.

Operator Syntax

This was covered in section REF. It is just worth understanding better the parsing process and the way in which the operators are mapped into the canonical form.

There are three types of operator: unary, binary and n-ary. These are mapped into the canonical form as follows:

- Each operator is associated with a specific Java class for the `sfFunctionClass` attribute. This mapping is effectively coded in the parser.
- Each operator type uses a different method for encoding the parameters that is appropriate for that type.
  - Unary operators use the named attribute `data` to indicate which attribute to use for its parameter.
  - Binary operators use the named attributes `left` and `right` to indicate the two parameters
  - N-ary operators generates a unique name for each parameter (equivalent to using `--`) and they are placed into the attribute set in the order in which they occur in the operator expression.

Examples of the mapping from operator to canonical form are given in the previous section REF.

Component Description Form.

This is the most complex, but also the most flexible form, of creating function applications. In particular it provides the ability to use the full templating capabilities of the SmartFrog language for defining function application.

In order to understand this form completely, it is necessary to understand the full phase model for language processing described in REF.

SmartFrog provides the predefined template:

```
Function extends {
  phase.function "org.smartfrog.sfcore.languages.sf.ConstructFunction";
};
```

which states that anything which extends *`Function`* will be converted to the canonical form (by the *`ConstructFunction`* class) during the function phase of the language processing.

So, this template can now be used to define function applications, and

```
foo extends Function {
    sfFunctionClass "org.smartfrog.sfcore.languages.sf.functions.Sum";
    -- 1;
    -- 2;
};
```

becomes transformed during the language processing into:

```
foo APPLY {
    sfFunctionClass "org.smartfrog.sfcore.languages.sf.functions.Sum";
    -- 1;
    -- 2;
};
```

Once we have a template form for the application of functions, we can start to be more creative.

For example, for each of the functions we can provide templates which encapsulate the function class:

```
sum extends Function {
   sfFunctionClass "org.smartfrog.sfcore.languages.sf.functions.Sum";
};

minus extends Function {
   sfFunctionClass "org.smartfrog.sfcore.languages.sf.functions.Minus";
};
```

and so on for all the functions and operators. We can even get more creative by providing default values for some of the parameters:

```
plus10 extends sum {
    -- 10;
};

sfConfig extends {
    x 30;
    y extends plus10 {
        -- x;
    };
};
```

It should be clear that this is an extremely powerful way of handling functions, providing great flexibility at the admitted cost of some syntactic weight. However the combination of the light-weight syntax of operators with the flexibility of the templated function form is a rich combination.

LAZY Function Application

A function can be evaluated if all its parameters are defined. However frequently some of the parameters depend on late-bound values and in this case the function evaluation must be delayed until this data is available. There are a number of ways of doing this.

The first is to annotate the canonical form with the `LAZY` flag, in the same way as a reference, and this will cause the function evaluation to be delayed until run-time. This method of indicating late evaluation is limited to the canonical form as in

```
X LAZY APPLY {
    sfFunctionClass ...;
    ...
};
```

For the other forms, as well as this form, the easiest way to indicate that late binding is required is to use `LAZY` propagation. So for example

```
X (LAZY x + 10 );
```

In this case, the fact that one of the parameters is `LAZY` propagates to the evaluation of the function. The same will also work with both the canonical and templated form.

A final form that works with the template form only, and that is to use the additional boolean-valued attribute `sfFunctionLazy` which if set to true, will be cause the template to be transformed to `LAZY`-flagged version of the canonical form. If set to anything else, or if it is not present, it will be transformed as described in REF to the form which is not flagged.

DATA Function Application

In just the same way as references can be considered as values rather than references to values, it can be useful to consider a function application itself as a value. This can be done by flagging the function application as `DATA`.

```
X DATA APPLY {
    sfFunctionClass ...;
    ...
};
```

In this case, whenever the attribute *x* is inspected, the *APPLY* value will be returned.

This can also be done in the template form by using the boolean-valued attribute `sfFunctionData`. If this attribute is true, the transformed canonical form will be flagged as `DATA`.

Functions as Link References

Early versions of SmartFrog did not support functions that required evaluation as part of the value space. Indeed, the only value that required processing was the link reference – and the term for de-referencing a link was link resolution. Equally, with the internal data structures (REF) only one Java interface that implemented value resolution, and that was the Reference interface.

Consequently, when functions (and indeed predicates) were introduced, rather than reworking everything towards a more general notion of expression with expression resolution and thus creating a backward compatibility issue, it was decided to keep the existing structure and naming and to make functions and predicates specialized forms of link reference.

In most cases this causes no confusion, but it does explain why in the full syntax the function and predicate syntax are part of the link syntax. Also, it explains why `DATA` and `LAZY` are applicable to these in exactly the same way as for references. It also explains why some language processing error messages may refer to references when dealing with function evaluation.

The confusion can occur when considering the parser, its APIs and the resultant data structures that are generated. In this case links, functions and predicates all derive from the notion of Reference.

16 Predicates: Assertions and Schemas

This section is closely related to that on functions and operators (REF), and this should be read and understood before continuing with this section.

It is frequently useful to be able to define a set of well-formedness conditions on the use of a template in order to guarantee that its use is correct. However, this should be done in a way in which all the benefits of template extension are not lost. To this end a mechanism similar to that defined for functions, is included which will check predicates defined and attached to a template.

There are two predicate types provided as part of the SmartFrog framework. These are the assertion predicates and schema predicates.

In a similar way to functions, predicates have a number of syntactic form, but each is transformed at some point into the canonical form for a predicate. Also in the same way as functions (REF), for historical reasons predicates are considered as specialized link references (i.e. resolvable values). Consequently syntactically the canonical form may occur wherever a link reference may occur, and with the same `LAZY` and `DATA` decorations.

Canonical Form

The canonical form for a predicate is very similar to that for a function

```
CanonicalPredicateForm::= (DATA)? (LAZY)? ASSERT { AttributeList }
```

So in just the same way as with functions, the different predicates are defined through a set of attributes. In the same way also, the specific type of assertion checking is defined by the attribute `sfFunctionClass`. However, it happens that both assertions and schemas use the same class to validate the associated assertions.

The common canonical form for both assertions and schemas is the following:

```
validationExample ASSERT {
    sfFunctionClass "....sf.functions.CheckAssertions";
    attr1 (...);    //boolean expression
    attr2 (...);    //boolean expression
    ...
}
```

Note that the form of `ASSERT` is almost identical to that for `APPLY`, differing only in that

- The function class must return a boolean value
- The “result” of the evaluation is handled differently. With functions the `APPLY` is replaced by the result of the evaluation, with predicates the `ASSERT` simply reports a violation if the function does not return `true`.

Users are free to extend SmartFrog by adding new assertion evaluators; this is identical to defining a new SmartFrog function (REF).

The only real difference between assertions and schemas is the way that the above canonical form is created from templates. For each of these two types of predicate the user model using templates is described first, followed by a description of how these templates are converted to the canonical form.

Assertions

Assertions are descriptions that are interpreted as a predicate and converted to the canonical form. An assertion consists of a description that contains attributes that should all evaluate to `true` - any attribute that evaluates to `false`, or indeed any other value, is considered to be an assertion failure. The names of these boolean attributes are not significant other than as documentation and for error reporting. There is an implicit conjunction (and) between the various assertion attributes given.

An assertion description must extend `Assertion`, and must be included as the value of an attribute in the description to which it applies. This attribute names has no semantics.

An example of an assertion is

```
WebServerAssertion extends Assertions {
        portValid ((port == 80) || (port == 8080) || (port == 8088));
}

WebServerTemplate extends Prim {
        validation extends WebServerAssertion;
        port 80;  // default value
}
```

In the same way that attributes may be added to an existing schema, attributes may also be placed into an `Assertions` description, or more than one `Assertions` description may be provided. As stated above, the names `validation` and `portValid` have no semantics and may have been anything. However it is useful to choose meaningful names since these are used to report assertion violations.

Assertion descriptions may be extended in the same way as any other component description. They may also be enhanced through placement. Consider the following extended example.

```
WebServerAssertion extends Assertions {
    portValid ((port == 80) || (port == 8080) || (port == 8088));
};

WebServerTemplate extends Prim {
    validation extends WebServerAssertion;
    port 80;  // default value
};

ThreadedWebServerTemplate extends WebServerTemplate {
    validation:maxThreads (threads < 100);
    validation:minThreads (threads > 10);
    threads 20;   // default value
};

AlternateThreadedWebServerTemplate extends WebServerTemplate {
    threadValidation extends Assertions {
        maxThreads (threads < 100);
        minThreads (threads > 10);
    };
    threads 20;   // default value
};
```

Or even

```
WebServerAssertion extends Assertions {
        portValid ((port == 80) || (port == 8080) || (port == 8088));
};

WebServerTemplate extends Prim {
    validation extends WebServerAssertion;
    port 80;  // default value
};

ThreadedWebServerAssertion extends WebServerAssertion {
    maxThreads (threads < 100);
    minThreads (threads > 10);
};

ThreadedWebServerTemplate extends WebServerTemplate {
    validation extends ThreadedWebServerAssertion;
    threads 20;   // default value
};
```

Schemas

A schema is a component description that describes the set of attributes a template should contain and some properties about the values of these attributes.

Schemas are best described through the use of an example, in this case of a template for a web server component. The example defines a schema for a web server template, and defines the template linked to the schema.

```
WebServerSchema extends Schema {
        port extends Integer;
        directory extends OptionalString;
}

WebServerTemplate extends Prim {
        schema extends WebServerSchema;
        port 80;  // default value
}
```

Note that the name for the attribute linking the template to its schema need not be, as in this case, `schema`. Indeed, a template may have more than one schema attached as attributes, in which case the uses of the templates are checked against all schemas attached. Schemas must extend the base schema template `Schema`.

However, unlike with assertions, the attribute names within the schema itself do matter - their names should be the same as the names of the attributes they are constraining. These constrained attributes are in the container component description. So in the example above, the name `port` in the schema has to be the same as the name of the attribute `port` in the template (which contains the schema).

Schemas may be extended in the same way as other templates, and their uses may easily be extended through placement as illustrated in the following examples.

```
ThreadedWebServerSchema extends WebServerSchema {
    minimumThreads extends Integer;
}

ThreadedWebServerTemplate extends WebServerTemplate {
    // overwrite existing schema with extended schema
    schema extends ThreadedWebServerSchema;

    minimumThreads 7;
}

AlternativeThreadedWebServerTemplate extends WebServerTemplate {
    // add to existing schema
    schema:minimumThreads extends Integer;

    minimumThreads 7;
}
```

Note that schemas are entirely optional and need be used only if desired. The value of a schema is that it provides a strict definition and the potentially type of the attributes, both required and optional, of a component. This should make it easier to work with, and so benefit users of the component.

The full set of attribute descriptions (e.g. `Integer`, `OptionalInteger`, etc.) that can be used in a schema is given in REF.

Mapping to the Canonical Form

The template forms described above are the usual way in which users will define schemas and assertions. However it is worth understanding how this is mapped to the canonical form so that errors, unparsing, etc. are understood as these will often use the canonical form.

Considering assertions first. These are handled almost exactly the same as template functions. The conversion from template to canonical form is carried out as a phase of the language processing (REF).

The definition of `Assertions` is as follows:

```
Assertions extends {
    phase.function "org.smartfrog...languages.sf.ConstructAssertion";
}
```

and it is the class `ConstructAssertion` that carries out the task of creating the canonical form as part of the function phase.

Each of the attributes, which for assertions are boolean-valued expressions, are transferred into the canonical form without further manipulation.

Schemas are handled similarly, but with some additional manipulation of each of the attributes. The definition of *`Schema`* is as follows:

```
Schem extends {
    phase.function "org.smartfrog...languages.sf.ConstructSchema";
}
```

Each of the attributes of the schema are converted to the invocation of the boolean function `CheckSchemaElement` with its parameters being the properties the attribute should have. In this way, each of the attributes of a schema actually becomes an attribute of an assertion.

For example

```
validate extends Schema {
    x extends Integer;
    y extends OptionalString;
};
```

will become transformed to

```
validate ASSERT {
  sfFunctionClass "... .sf.functions.CheckAssertions";
  x APPLY {sfFunctionClass "... .sf.functions.CheckSchemaElement";
           name "x";
           optional false;
           binding "anyBinding";
           class "java.lang.Integer";
          };
  y APPLY {sfFunctionClass "... .sf.functions.CheckSchemaElement";
           name "y";
           optional true;
           binding "anyBinding";
           class "java.lang.String";
          };
};
```

Indeed, users may decide to make use of the `CheckSchemaElement` boolean function directly themselves within an assertion without making use of the Schema template form. Notice that the schema transform has also added the `name` attribute -whose value is the name of the attribute to be checked, appropriately generated from the original attribute name in the schema.

The full set of attributes required by `CheckSchemaElement` and their possible values is given in REF.

Predicate Evaluation: Static, Run-Time and LAZY

Predicates, when evaluated, either pass or generate an error. However there are different times when predicates could be evaluated:

- statically over a description as part of the parsing
- dynamically at runtime over the components that are created from the description and whose attribute values may change over times

- both statically and dynamically.

All of these must be appropriately handled in the context of late-bound data.

It is possible to indicate to the predicate when it should be checked through the use of the attribute `sfAssertionPhase`. This attribute may have the following string values:

- `static` – the predicate is checked during the parsing and is discarded afterwards, the attribute defining the assertion is eliminated from the description. An error is reported if the check cannot be made because some of the data required is late bound (`LAZY`).

- `staticLazy` – as for static, but no error is reported if it cannot be checked due to late bound data. In these circumstances, it is just ignored and discarded.

- `dynamic` – checked statically and left as an attribute to be checked dynamically as well using the run-time APIs to control when this is done (REF).

If the static check is to be left out completely, and only the dynamic check done, select the assertion phase as dynamic and ensure that the predicate is made `LAZY`. This can be done:

- By using the `LAZY` flag if using the canonical form

- By making use of lazy propagation and either ensuring that one of the references within the assertion is flagged `LAZY` or that one of the attributes referred to is `LAZY`.

- By setting the attribute `sfFunctionLazy` to `true` in the assertion or schema template. This is the same mechanism for signalling that a template form for a function should be late evaluated.

17 Link Reference Usage Patterns

This section describes a number of patterns for the use of links to provide a degree of abstraction in the provision of template component descriptions. These abstractions help in creating more reusable templates.

Template Parametrization Pattern

When extending a prototype, it is normal to override the values of certain attributes to customize the prototype to its actual use. The simplest way is to extend with the replacement attribute – however this only works for a top-level attribute. Modification of attributes deep in the structure requires the placement of the overriding attribute into the correct context, as in the example:

```
Service extends {
    hostname "localhost";
    portNum 4567;
}
ServicePair extends {
    service1 extends Service ;
    service2 extends Service ;
}
sfConfig extends ServicePair {
    // user needs to know structure of ServicePair
    service1:hostname  "riker.smartfrog.org";
    service2:hostname "ackbar.smartfrog.org";
}
```

This works adequately, but it has the disadvantage that the use of the `ServicePair` prototype requires knowledge of its structure, though it does have the advantage that any attribute in the structure may be changed if necessary. However, under normal circumstances, there are attributes whose values are expected to change, and others that are not. Under these circumstances, it would be good if the description could be parameterized on these attributes. However, the normal form of parameterization as provided in programming language functions is not a good fit to the SmartFrog notation semantics – so the language provides a way of finding a way of hiding the structure of a description and making it easier to override “deep” attributes.

This technique, more of a pattern for the use of links, is shown in the following example:

```
Service extends {
    hostname "localhost"; // default value
    portNum 4567;
};

ServicePair extends {
    s1Host "localhost"; // provide default value
    s2Host "localhost";
    service1 extends Service { hostname s1host; } // lift attribute
    service2 extends Service { hostname s2host; } // ditto
};

sfConfig extends ServicePair {
    // user needn't know structure of ServicePair
    s1host "riker.smartfrog.org";
    s2host "ackbar.smartfrog.org";
};
```

It is clear that the use of `ServicePair` requires only the extension with top-level attributes to set the attributes deeply defined in the `Service` prototype. This pattern, of the use of links lifting an attribute value to one provided in the outermost context, is called the parameterization pattern and is very frequently used.

Note that if a default value for a lifted attribute is not given within the description (in this case `ServicePair` provides defaults for both the lifted attributes `s1Host` and `s2Host`), a deploy resolution error will occur if the parameter is not provided at time of use, since the value to resolve the link will not be found.

Structure Hiding Pattern

A combination of links and the `sfLocal` tag can be used to provide abstraction of the structure of a description. With `LAZY` propagation, this can also provide abstraction as to whether data will be late or early bound – users do not need to know.

Consider the following example, the description of a service containing several components only one of which should be visible:

```
Service extends Compound {
    [sfFinal]
    mainAPI LAZY body:comp1;

    [sfLocal]
    body extends Compound {
         comp0 extends Prim {...};
         comp1 extends Prim {...};
         comp2 extends Prim {...};
    };
};
```

This description can now be used in the definition of a deployed system

```
sfConfig extends Compound {
    service extends Service;

    client extends Prim {
        serviceAPI service:mainAPI;
    };
};
```

The client therefore obtains the service API from its `serviceAPI` attribute. This has been tagged as late bound through propagation and its value, when the references are followed at run-time, is the Java Object Reference to the component that implements `comp1`.

This is in some ways the dual of the parametrization pattern, but they can quite happily be used together in the same template:

```
Service extends Compound {
    keyData TBD;

    [sfFinal]
    mainAPI LAZY body:comp1;

    [sfLocal]
    body extends Compound {
         comp0 extends Prim {
            ...
            data keyData;
            ...
         };
         comp1 extends Prim {...};
         comp2 extends Prim {...};
    };
};

sfConfig extends Compound {
    service extends Service {
        keyData 10;
    };

    client extends Prim {
        serviceAPI service:mainAPI;
    };
};
```

18 Resolution – Semantics For The SmartFrog Notation

Resolution is the process by which the raw SmartFrog definitions, with their extensions, placements and links, are turned into the set of attributes that they semantically represent.

In addition to these three steps, there are other steps (phases) in the complete semantic description of the SmartFrog notation, such as function and predicate transformation to canonical form and any user-defined phases.

The semantics are described here through an operational description of the process used to transform the input form to the intermediate representation required for the run-time part of SmartFrog. This is less abstract than it might be, but more directly an accurate representation of the actual processes used.

The transformation steps are known in SmartFrog as resolution steps. These are respectively type resolution, placement resolution, function resolution and link resolution. They are carried out in that order: first the types are expanded, then attributes placed into the correct context from the context in which they were defined, functions and predicates are transformed to their respective canonical form and finally links are resolved including the evaluation of any functions and predicates (as expressions and predicates are regarded as special kinds of link reference - REF).

It should be noted that the entire description is type, place and function resolved, but only the top-level `sfConfig` attribute is normally link resolved. In general if the other top-level attributes are link resolved, errors will occur; they are only present to be available as prototypes.

Type Resolution

Type resolution is the expansion of the prototypes provided in the `extends` part of a component description. The syntactic form for a component description is roughly (REF)

```
Component::= extends (DATA)? Prototype (, Prototype)* ;

Prototype::= ( BaseReference | {  AttributeList } )
```

A component thus consists of a set of prototypes, either a reference to a pre-defined prototype or an explicitly given list of attributes. It is optionally flagged as `DATA` – this is noted and the component description so flagged, but this is ignored for the rest of the language processing.

This process of type resolution is a depth-first pass over the hierarchy of component descriptions starting at root component description, in the order of definition of the attributes. The purpose is to identify all the attributes defined through `extends` and to turn them into a simple attribute set from all the prototypes provided.

As can be seen from the syntax, there are two forms of prototype: references and explicitly provided attribute sets. The semantics are in effect to start with an empty attribute set and to add each attribute from the first mentioned prototype, then those from the next, and so on through the list of prototypes.

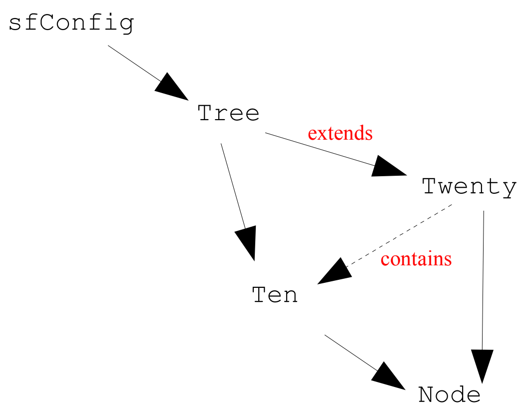
If the prototype reference indicates a component description that is not yet resolved, it resolves it first before copying: i.e. each type resolution is carried out with respect to the location where the prototype is defined.

Consider this simple example:

```
DataVal 40;

Node extends {
```

```
    data TBD;
    left TBD;
    right TBD;
};

Tree extends Twenty, {
    data DataVal;
    left:data 5;
};

Twenty extends Node, {
    data 20;
    left extends Ten;
    right extends Ten;
};

Ten extends Node;
    data 10;
};

sfConfig extends Tree;
```

The first aspect to consider are the dependencies, as shown in the following dependency graph of both the direct extensions of prototypes as well as the containment of an attribute that extends a prototype.



The dependency graph defines a partial order for type resolution – no cycles are allowed – and this is used to define the semantics of type resolution. The targets of the arrows must be resolved before the sources.

The leaves of the dependency graph are type resolved first, though these are by definition explicit component descriptions and so simple to resolve. In the above example, `Node` will be done first, followed by `Ten`, `Twenty`, `Tree` and finally `sfConfig`. The attribute `DataVal` doesn't need to be type resolved since it is not a component description defined through extends.

The process of type resolution of an attribute defined through extends is as follows:

- If the attribute is defined as extending a single explicit component description – that is the attribute value
- if the attribute is defined as extending a number of prototypes, either through reference or explicitly, the semantics is

  1. create a new "working" component description
  2. iterating through each prototype in turn, in the order given, copying the attributes into the working component description in the order defined in the prototype.

3. If the attribute name already exists in the working list, replace the value; if it does not exist add the attribute to the end of the working list.

- The value of the attribute is the resultant working component description.

So, if this is applied to the example above, we get:

```
DataVal 40;

Node extends {
    data TBD;
    left TBD;
    right TBD;
};

Tree extends {
    data DataVal;
    left extends {
        data 10;
        left TBD;
        right TBD;
    };
    right extends {
        data 10;
        left TBD;
        right TBD;
    };
    left:data 5;
};

Twenty extends {
    data 20;
    left extends {
        data 10;
        left TBD;
        right TBD;
    };
    right extends {
        data 10;
        left TBD;
        right TBD;
    };
};

Ten extends {
    data 10;
    left TBD;
    right TBD;
};

sfConfig extends {
    data DataVal;
    left extends {
        data 10;
        left TBD;
        right TBD;
    };
    right extends {
        data 10;
        left TBD;
        right TBD;
    };
    left:data 5;
};
```

Notice how every extends is now defined as an explicit component description. Consequently by the first bullet point above repeating type resolution has no further effect; type resolution is idempotent.

However if one or more prototype references has failed to resolve, or refers to an attribute whose value is not a component description, the whole resolution process ceases and an exception is thrown indicating the missing or erroneous prototypes and the locations of these errors. Dependency cycles are also an error and are reported.

Placement Resolution

Placement resolution is the process by which the attributes are placed into the correct location. Attributes are named, and this name may contain a reference to a component description as well as the name by which it is to be known in that component description. If the reference is not present, the attribute is assumed to be in the correct component description as defined.

Thus in the example attribute declaration:

```
foo:bar:baz 42;
```

The `foo:bar:` defines the target component description, and `baz` defines the name for the attribute in that component description.

Placement resolution is the transformation process that results in the attribute definitions being removed from their point of definition and placed in the target component descriptions. The process is a multi-pass process, for each pass:

- traverse the component description hierarchy
  - depth first
  - visiting the attributes in the order of definition (as determined by type resolution)
- each attribute visited is examined, if it should be placed elsewhere – try to do so, if it fails – leave as is.

The pass is repeated until one of the following occurs:

- there are no placements left to carry out
- in the pass, no placements have been successfully carried out but at least one placement was tried

In the first instance, the placement resolution has successfully completed, the second it has not and an error is generated for each remaining placement.

So for the example defined above, there are a number of placements to complete: one in `Tree` and the other in `sfConfig`. For brevity, consider only `Tree`; `sfConfig` would be identical.

```
Tree extends {
    data DataVal;
    left extends {
        data 10;
        left TBD;
        right TBD;
    };
    right extends {
        data 10;
        left TBD;
        right TBD;
    };
    left:data 5;
};
```

The definition of the placement *`left:data`* results in left's data attribute being set to 5, giving

```
Tree extends {
    data DataVal;
    left extends {
        data 5;
        left TBD;
        right TBD;
    };
    right extends {
        data 10;
        left TBD;
        right TBD;
    };
};
```

Note how the placed attribute has been removed, and the existing value in the target component description has been overridden.

This is a simple example carried out in a single pass. To see why multiple passes are necessary, consider the following:

```
foo extends {
     a 21;
}
foo:bar:a 42;
foo:bar extends { b 34; }
```

In the first pass, the attribute `foo:bar:a` is first to be placed, but it fails since `foo` does not yet contain `foo:bar` as a component description. Also in the first pass, but later since it is defined later, `foo:bar` is placed, giving

```
foo extends {
     a 21;
     bar extends { b 34; }
}
foo:bar:a 42;
```

This leaves a placement incomplete so a second pass is required. This time it succeeds, resulting in

```
foo extends {
     a 21;
     bar extends {
        b 34;
        a 42;
     }
}
```

This order dependency does not have much of an effect, except for when two identically named attributes are placed into the same component description. At this point understanding the order of resolution becomes important.

Since placement resolution is carried out after type resolution, the following consequences should be noted:

- As type resolution is carried out before placement, attributes placed into a prototype will not be inherited by those extending the prototype; the placement occurs too late to be taken into account.

- Again, as type resolution is carried out before placement, do not place an attribute that is itself to be used as a prototype; it will not be found during the type resolution.

- Wherever possible, placement should be restricted to referencing downwards into a structure from the point of attribute definition. Descriptions can be very hard to understand if `PARENT`, `ROOT` or `ATTRIB` are used in a placement reference; this particularly so within a component description to be used as a type. As a consequence, this release of SmartFrog does not permit these reference parts to be used in a placement; references are limited to "`HERE`" reference parts represented as a sequence of words.

The reason why type resolution is done before placement resolution is that the intended use for placement is to "fill-in" empty or defaulted "attribute slots" in a prototype. As each instance of the prototype will in general need differently filled slots, placement must be done after the type has been resolved for each instance.

Note that placement of attributes whose values are links do not modify the links to "correct" for the new location. Thus, links are resolved with respect to where they are placed, not where they are defined. Thus

```
data 10;
```

```
foo extends {
    data 20;
};
foo:x data;
```

results in

```
data 10;
foo extends {
    data 20;
    x data;
};
```

and hence `x` having the value 20, not 10.

Function Resolution

Function resolution is when the component description forms for functions and predicates are transformed into the canonical forms ready for evaluation. The required transformations are covered in REF and REF. Function resolution is in reality a pre-defined "user" phase as described in REF.

The function phase occurs after the type and place resolution phases, so that the definitions of the component descriptions for functions and predicates can take full advantage of these semantics.

Link Resolution

Link resolution is a slight misnomer coming from the time when the only "expressions" were basic values and link references. In the latest versions of the language there are operators, functions and various forms of predicate. These are all resolved (evaluated) during link resolution which should perhaps more accurately be known as expression resolution.

Link resolution is the most straightforward of the three forms of resolution; all links and other expressions are resolved in their location after type and place resolution, and the referenced value replaces the link as the value of the attribute. There are a number of points to note:

- Link resolution only occurs on `sfConfig` since this top-level attribute defines the meaning of the whole description. Other attributes not contained within `sfConfig` are link resolved only if required. This allows other top-level attributes to contain errors such as having `TBD` values – but this would be expected for component descriptions that are only to be used as prototypes for others.

- Functions and assertions are evaluated inside-out. Thus the attribute parameters of canonical form functions are first resolved, then the function evaluated. The same goes for assertions.

- Only links, functions, etc. that are **not** `LAZY` or `DATA` are resolved; those that are `LAZY` are left unresolved for run-time evaluation. `DATA` links, functions, etc, are not expressions to evaluate, they are values in their own right.

- If the result of dereferencing a link is a link, this is first resolved and the result of that resolution is used. This is known as link chaining. This is a special case of link-offsetting in which any link found during the dereference of a link offsets the remaining dereference to the target of that link.

- Links are always resolved in the contexts in which they are located after the type and placement resolution phases are over, not necessarily those in which they were defined.

- Links referring to an attribute whose value is a `LAZY` link will leave the `LAZY` link unchanged and itself become `LAZY` through propagation.

- Functions and predicates containing a `LAZY` parameter (link or nested function evaluation) will itself become `LAZY` through propagation.

- In resolving a link, the value of the attribute referenced is not copied, but shared, at the original point of definition if this is relevant (e.g. for component descriptions and their parent). Thus any operation that affects the value of this data has an impact on all parts of the tree that share this data.

Consider the following examples to illustrate some aspects of the link resolution (or run-time link resolution with `LAZY` links) semantics.

- Link Chaining:

The following works with either `LAZY` or "normal" links – it is just more obvious with the normal ones so is illustrated here with the `LAZY` links.

```
sfConfig extends {
    x extends {
        v 10;
    };
    y extends {
        w LAZY x:v;
    };
    z LAZY y:w;
};
```

The resolution of the value of attribute `z` will follow the link to `y:w`, then chain to the value of `x:v` resulting in `z` having the value `10`.

- Link Offset

Again, `LAZY` or normal links could be used with the same consequences

```
sfConfig extends {
    y extends {
        x extends {
            a extends {
               b 20;
            };
        };
        w LAZY x:a;
    };
    z LAZY y:w:b;
};
```

The resolution of the value of `z` causes the reference *`y:w`* to be dereferenced. However this first step refers to a reference `LAZY x` which is dereferenced before the remainder of the original reference is completed. Thus *`z`* resolves to *20*.

- Function Evaluation

```
sfConfig extends {
    x 10;
    y (x + 5);
    nested extends {
        x 20;
        z y;
    };
};
```

The value of `nested:z` will be the same as the value of `y`, not the definition of `y` re-evaluated in the context of `z`. Thus `z` will have the value `15`, not `25`.

- `DATA` links

As an illustration of the difference between DATA and LAZY, consider the following example.

```
sfConfig extends {
    x 10;
    y LAZY x;
    z DATA x;
};
```

The language processing will resolve neither y nor z. However, at run-time, if the value of y is requested, this will return the value 10; the value of z in contrast will be DATA x.

The Difference Between Types and Links

On the surface, there are many similarities between the definitions of *x* and *y* in:

```
Foo extends {
    a 10;
};

x extends Foo;
y Foo;
```

They both appear to end up by having the definition of a component description containing a.

One obvious difference is that since they occur each side of place resolution, a placement into Foo will affect y but not x. However there are more subtle differences to do with the sharing of data with links, rather than the copying of data with extends. Consider the following example:

```
data 1;

Foo extends {
    a data;
}

sfConfig extends {
    data 100;
    x extends Foo;
    y Foo;
}
```

In this definition, sfConfig:x:a has the value 100, whereas sfConfig:y:a has the value 1. The reason for this discrepancy is that the extends copies the definition of Foo and the subsequent link resolution for data is done relative to the copy's location. The link, on the other hand, simply links to the definition of Foo in its existing position, and there the value of data on resolution is 1.

The difference can also be highlighted using one of the functions, such as *next* that return a different value at each use (strictly speaking they are not functions as they have a side-effect). Consider the following description:

```
sfConfig extends {
    x extends next;
    y extends x;
    z x;
}
```

Assuming that this is the first use of next, *sfConfig:x* will have the value 0, example:y will have the value 1, but sfConfig:z will have the value 0. This is because it shares the result of the function bound to sfConfig:x.

Note that at the very end of the language processing as part of the conversion to the core data model, the sharing is eliminated and each attribute will have its own copy of the final value. This is explained in detail in REF.

19 The SmartFrog Grammar Rules

SmartFrog defines the default language's grammar using the Java Compiler Compiler system from Sun. This is a tool known as JavaCC. The SmartFrog grammar rules described here are part of the JavaCC input, the file `DefaultParser.jj`, which is available in the source distribution. The listing is derived from this file.

```
Entry Points
------------

Attributes     ::=    AttributesNoEOF <EOF>
AttributeList  ::=    AttributeListNoEOF <EOF>
Reference      ::=    ReferenceNoEOF <EOF>
AnyValue       ::=    ( Component | Expression ) <EOF>
PrimitiveValue ::=    Primitive <EOF>


Grammar
-------


AttributesNoEOF ::=   AttributeListNoEOF
AttributeListNoEOF ::= ( Attribute | Includes | ; )*


Includes ::= ((#codebase String)? (#include | #include?) String)

Attribute ::=  (([ Tags ])? Name Value)

Tags    ::=    ( <WORD> ,? Tags )?

Name    ::= --
          | <WORD>  (: Name)?

Value   ::= Component
          | Expression ;
          | ; )


Expression ::= ( ReferenceNoEOF | Basic | Operator | IfThenElse )


Operator ::=   (  ( ! Expression )
                | ( Expression (
                          (++ Expression (++ Expression)*)
                        | (<> Expression (++ Expression)*)
                        | (+  Expression (+  Expression)*)
                        | (*  Expression (*  Expression)*)
                        | (&& Expression (&& Expression)*)
                        | (|| Expression (|| Expression)*)
                        | (== Expression )
                        | (!= Expression )
                        | (>= Expression )
                        | (<= Expression )
                        | (>  Expression )
                        | (<  Expression )
                        | (-  Expression )
                        | (/  Expression )
                       )?
                  )
               )


IfThenElse ::= IF Expression THEN Expression ELSE Expression FI

Component      ::= (DATA|LAZY)? BaseComponent

BaseComponent ::= ;
                | (NULL ( BaseComponentRest | ; ))
                | (BaseComponentLinkType ( BaseComponentRest | ; ))
                | (BaseComponentAttributesType BaseComponentRest?)




BaseComponentRest ::=
          (, ( BaseComponentLinkType       ( BaseComponentRest | ; )
             | BaseComponentAttributesType ( BaseComponentRest | ; )
             | NULL                        ( BaseComponentRest | ; )
```

```
          )
          | BaseComponentAttributesType

BaseComponentLinkType ::= LinkReference

BaseComponentAttributesType ::= { AttributesNoEOF }

ReferenceNoEOF ::=     LAZY? BaseReference

BaseReference ::= ((OPTIONAL ( Primitive ))? LinkReference )
                | ( APPLY { AttributesNoEOF } )
                | ( ASSERT { AttributesNoEOF } ) )

LinkReference ::= ReferencePart ( : ReferencePart )*

ReferencePart ::= ROOT
                | PARENT
                | ATTRIB <WORD>
                | HERE <WORD>
                | THIS
                | CONSTANT (<WORD>|<STRING>)
                | PROPERTY <WORD>
                | IPROPERTY <WORD>
                | ENVPROPERTY <WORD>
                | IENVPROPERTY <WORD>
                | HOST (<WORD>|<STRING>)
                | PROCESS
                | <WORD>


Basic   ::= Primitive
         | [ (Expression ( , Expression )* )? ]

Primitive ::=  NULL
             | String
             | Number
             | Boolean
             | <BYTEARRAY>
             | [| ( Primitive ( <COMMA> Primitive )* )? |]
             | DATA BaseReference
             | TBD


Number ::= <DOUBLE>
         | <FLOAT>
         | <INTEGER>
         | <LONG>


String ::=  <STRING>
         |  <MULTILINESTRING>


Boolean ::= true | false
```

20 The SmartFrog Lexical Rules

The SmartFrog lexical rules described here are part of the JavaCC input, the file `DefaultParser.jj`, which is available in the source distribution. The listing here is a slight simplification of this file.

```
/* White Space */
SKIP : " "| "\t"| "\n"| "\r"| "\f"

/* Comments */
SINGLELINECOMMENT: "//"(~["\n","\r"])*
FORMALCOMMENT: "/**" ... "*/ "
MULTILINECOMMENT: "/*" ... "*/"

/* Reserved Tokens */
RESERVED:
   "#codebase" | "#include" | "#include?"
 | "APPLY" | "ASSERT" | "ATTRIB"
 | "DATA"
 | "ENVPROPERTY" | "extends"
 | "false"
 | "HERE"  | "HOST"
 | "IENVPROPERTY" | "IPROPERTY"
 | "CONSTANT"
 | "LAZY"
 | "NULL"
 | "OPTIONAL"
 | "PARENT" | "PROCESS" | "PROPERTY"
 | "ROOT"
 | "TBD"  | "THIS"  | "true"
 | ";" | "," | "{" | "}" | "[" | "]" | ":"
 | "[|" | "|]"  | "--"  | "(" | ")"
 | "==" | "!=" | ">=" | ">" | "<=" | "<" | "<>" | "!"
 | "+" | "-" | "*" | "/" | "++" | "&&" | "||"
 | "IF" | "THEN" | "ELSE" | "FI"

/* Tokens - using Unicode */
WORD: LETTER (LETTER|DIGIT|SPECIAL)*
SPECIAL: [".", "_", "-", "+", "@", "#", "~", "$", "%", "^", "&"]
LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]

/* Literals */
STRING: ("\"" (
          (~["\"","\\","\n","\r"])
          | ("\\"
              ( ["n","t","b","r","f","\\","'","\""]
              | ["0"-"3"] ["0"-"7"] ["0"-"7"]
              )
            )
          )
```

```
        )* "\"")

MULTILINESTRING: ("##" (
          (~["#","\\"])
          | ("\\"
              ( ["n","t","b","r","f","\\","'","#"]
              | ["0"-"3"] ["0"-"7"] ["0"-"7"]
              )
            )
        )* "#")

NUMBER: <INTEGER> | <FLOAT> | <LONG> | <DOUBLE>

INTEGER: (("-")? ["1"-"9"] (["0"-"9"])*)  |  "0"

FLOAT_BASE:  ("-")?
        (
          (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
        | "." (["0"-"9"])+ (<EXPONENT>)?
        | (["0"-"9"])+ <EXPONENT>
        | (["0"-"9"])+ (<EXPONENT>)?
        )

EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >

DOUBLE: <FLOAT_BASE> (["d", "D"])?

LONG: <INTEGER> (["l", "L"])?

FLOAT: <FLOAT_BASE> ["f", "F"]

Hex: "#HEX#" (["A"-"F", "a-f", "0"-"9", "\n", "\t", " "])* "#"
Oct: "#OCT#" (["0"-"7", "\n", "\t", " "])* "#"
Bin: "#BIN#" (["0"-"1", "\n", "\t", " "])* "#"
Dec: "#DEC#" (["0"-"9", "\n", "\t", " "])* "#"
B64: "#B64#" (["A"-"Z", "a-z","0"-"9", "/", "+", "\n", "\t", " "])* "#"
```

21 Predefined SmartFrog Functions

SmartFrog provides a number of functions. These functions are all available as templates that are defined in a file which must be included if they are to be used. However, some are also available as operators, using the SmartFrog operator syntax, and in this case the include file is not required.

The operators are all converted into an instance of the expanded template at time of parsing, so may in every respect be treated in the same way as a use of the template itself. Furthermore, it should be noted that any references that are used within an expression containing operators, will be resolved in the context of the templates – this means that use of reference parts such as `PARENT` are hard to use. `ATTRIB` reference parts are useable in the normal way.

Note also that since attribute names may contain many of the operator symbols, it is best to always surround the operators with space characters to ensure that they do not accidentally "stick" to the names.

The functions are defined by including the `components.sf` or `functions.sf` file as follows:

```
#include "org/smartfrog/functions.sf"
```

The functions defined as operators may be grouped into three main categories: unary, binary and nary.

Unary Operators

There is currently only one unary operator, the Boolean negation operator. The syntax for unary operators is

```
( opsymbol value )
```

The surrounding `( )` symbols must be present. All templates for unary operators have as their parameter the attribute "`data`". Other attributes are allowed, but are ignored for the purpose of evaluating the function. They may, of course, be used for the definition of the data attribute during earlier phases.

21.1 not

Operator symbol: !

The function not is defined as the negation of the boolean attribute "`data`". If the attribute is not present or of the wrong type, an exception is reported.

```
x   true;
foo   (! x);
bar extends not {
    data x;
}
```

Binary Operators

There are a number of binary operators covering primarily the arithmetic, comparison and logical operators. The syntax for binary operators is

```
( value opsymbol value )
```

The surrounding `( )` symbols must be present. All templates for binary operators have as their parameter attributes the names "`left`" and "`right`", to indicate which value it is. Other attributes may be present and are ignored whilst evaluating the operator.

21.1 minus

Operator symbol: `-`

The minus operator subtracts the right attribute from the left, resulting in a number which satisfies the Java rules for numbers. If either of the two attributes are not numbers, an exception is thrown. Other attributes that may be defined in the template are ignored.

```
minus10 extends minus {
        value;
        left value;
        right 10;
}

foo extends minus10 {
    value 34;
}

aFoo  ( 34 - 10 );
```

21.1 divide

Operator symbol: /

The divide operator divides the left attribute by the right, resulting in a number which satisfies the Java rules for numbers. If either of the two attributes are not numbers, an exception is thrown. Other attributes that may be defined in the template are ignored.

```
percent extends product {
    fraction extends divide {
           enum; denom;
           left enum;
           right denom;
       }
     -- 100;
}

foo extends percent {
    fraction:enum 34;
    fraction:denom 56;
}
```

21.1 EQ, NE

Operator symbols: `==`, `!=`

These operators are the comparator operators, equals and not equals respectively. The two attributes, left and right, are compared using the Java equals method (`left.equals(right)`). The result of the function is the boolean value that is returned by that test.

21.1 GE, GT, LE, LT

Operator symbols: `>=, >, <=, <`

These operators are the numeric value comparators, testing to see if the left attribute value is greater than or equal to (or whatever operator is used) the right attribute. The Java rules for numeric comparison are used.

N-ary Operators

N-ary operators are operators that are may have a arbitrary number of attribute parameters. All the attributes provided within the template are assumed to be part of the function, and the names used to provide these attributes are ignored. Thus the “new unique” name “--” is normally used for these operators.

The syntax for an nary operator is as follows:

```
( value opsymbol value opsymbol value opsymbol ... )
```

Each of the operator symbols must be identical, though other may be used by nesting the use of operators wherever a value is expected. The above form is converted to the expanded template form during parsing, so any references that are used when a value is expected is resolved relative to the template and not the operator expression.

21.1 concat

Operator symbol: `++`

The concatenate function takes each of its attribute parameters and concatenates them in the order of definition. These attributes are converted to strings using the `toString()` Java method. An example of the use of the concatenate function is:

```
myString extends concat {
    -- "the meaning of life is ";
    -- 42;
    -- extends concat {
        -- " by ";
        -- "Douglas Adams";
    }
}
```

which results in the string *"the meaning of life is 42 by Douglas Adams"*.

21.1 append

Operator symbol: `<>`

The append function is similar to the vector function, except that all parameters must be vectors and these are expanded in-line. The difference can be seen by considering the same example

```
myVector extends vector {
    -- ["the meaning of life is "];
    -- [42];
    -- extends vector {
        -- " by ";
        -- "Douglas Adams";
    }
}
```

which results in the vector

```
[ "the meaning of life is", 42, "by", "Douglas Adams"]
```

The operator form can be used for the same purpose. The following definition is equivalent to the definition of `myVector` above.

```
MyVector   ( ["the meaning of life is"]
                 <> [42]
                 <> ["by", "Douglas Adams"] );
```

21.1 sum

Operator symbol: `+`

The sum function sums each of its attributes which must be numbers, failure will result in an exception. The names of the attributes are, of course, irrelevant. An example of the use of the sum function is:

```
val1 34;
val2 45;
num extends sum {
    -- val1;
    -- 345;
    -- val2
}
```

This will result in *`num`* being set to *`424`*. An equivalent expression is

```
num (val1 + 345 + val2);
```

21.1 product

Operator symbol: `*`

The product function multiplies each of its attributes type-cast to integers, failure will result in an exception. The order is irrelevant. An example of the use of the product function is:

```
times10 extends product {
ten 10;
}

myNum extends times10 {
    val 34;
}
```

This will result in `myNum` being set to `340`.

21.1 and

Operator symbol: `&&`

The conjunction operator takes the logical "and" of all its attribute parameters. Each of the parameters must be a boolean and if they are not, an exception is thrown. As with all nary operators the names of the attributes are irrelevant.

21.1 or

Operator symbol: ||

The disjunction operator takes the logical "or" of all its attribute parameters. Each of the parameters must be a boolean and if they are not, an exception is thrown. As with all nary operators the names of the attributes are irrelevant.

Other Functions

In addition to the operators, there are a number of other functions provided. Some of these also have specialised syntactic forms – most notably the `ifThenElse` and the vector function. These syntactic forms are converted to the template form at time of parsing, and do not require the use of the include file "`functions.sf`". If the templates are used directly, then this file must be included.

21.1 IfThenElse

The `ifThenElse` function is provided to conditionally provide a value for an attribute. The template uses three well-known attributes: `if`, `then` and `else`. Other attributes may be present but are ignored. If one of the attributes is not provided, or is the if attribute is not a boolean value, an exception will be thrown. Since the `ifThenElse` template is defined through attributes, these may be changed by extension or placement.

```
aStepFunction  extends  ifThenElse {
    boundary 10;
    value;

    if (value > boundary);
    then 20;
    else 30;
}
```

This may now be used as follows:

```
aValue extends aStepFunciton {
        value 5;
}

anotherValue extends aStepFunction {
        value 15;
}
```

Which would give values for these two attributes as `20` and `30` respectively.

The special syntactic form for `ifThenElse` is as follows:

```
foo    IF (10 > 20) THEN 5 ELSE 15 FI;
```

defining foo to be 15. Note that as with all special forms, any references used within it are evaluated relative to the transformed syntax, and not that given. The definition for foo above is equivalent to:

```
foo extends ifThenElse {
        if (10>20);
        then 5;
        else 15;
}
```

21.1 vector

The vector function takes each of its attribute parameters in the order provided and creates a vector whose elements are the values of the attributes (names are ignored, hence unique naming is useful). An example is

```
myData extends vector {
    -- "the meaning of life is ";
    -- 42;
    -- extends vector {
        -- " by ";
        -- "Douglas Adams";
    }
}
```

which results in the vector with three elements, the third of which is itself a vector.

There is a special syntactic form for vector which is as follows:

```
myData ["the meaning of life is", "42", ["by", "Douglas Adams"]];
```

This is the equivalent of the earlier definition using the template. Again, any reference provided within the vector syntax is resolved relative to the transformation into the template form.

21.1 formatString

FormatString is a function that takes a format string and a set of parameters and creates a resultant string which has the values of the parameters embedded. The format string attribute itself should be named format and the various parameter strings should be named *sx* where x is a single digit. The format string should identify the places where the various parameter strings should be embedded using the characters “$x” for a single digit x. An example is

```
myString extends formatString {
    format "the meaning of $2 is $1";
    s1 42;
    s2 "life";
}
```

The attributes may of course be links to other values, but not LAZY links as these are not resolved in time for the function phase.

21.1 random

The random function, which in truth is not really a function since it returns a different value for each invocation, returns a random number as follows:

- if the attribute integer is set to true, an integer between attributes min and max is returned, otherwise a floating point value between 0 and 1. The default values for min and max are 0 and 10 respectively.

- if the attribute seed is provided, and the random number generator has not yet been initialized, that seed is used.

Examples of the use of the random are:

```
dice extends random {
    integer true;
    min 1;
    max 6;
}

myConfig extends Compound {
    throw1 extends dice;
    throw2 extends dice;
}
```

Each of `throw1` and `throw2` will be some random integer between 1 and 6. Note that each invocation in `myConfig` is independent. Each JVM contains a single random number generator for use during function resolution.

21.1 next

The `next` function is one that returns a monotonically increasing value, guaranteed never to return the same number twice within a single description. Again, it is not strictly a function since it never returns the same value for the same parameters. The only parameter attribute is the `base` attribute, setting a minimum value for the values. If the `base` is below the next value, it is ignored. If it is above, the next value will be the `base`. The default `base` is 0.

An example of the use of *next* is

```
unique extends concat {
    prefix "xxyyyqqq";
    postfix extends next;
}

myConfig extends Compound {
          name extends unique;
          otherAttr 42;
}
```

21.1 ref

The ref function converts a string to a reference, and then optionally resolves the reference in place or leaves it as a `LAZY` link. This allows links to be created by using functions over strings, such as concatenation, to generate a reference. The attributes for the ref function are `reference` which is the string to convert, and the optional attribute `lazy` which defaults to false and controls whether to leave the reference as a lazy link, or to resolve to replace it with the value obtained.

An example of the use of ref is:

```
x "ROOT";
y "a";
a 24;

myConfig extends ref {
          reference (x ++ ":" ++ y);
          lazy false; // default
}
```

This results in `myConfig` being set to 24.

21.1 date

The `date` function returns a string representation of the current date. There are no formatting parameters. Again, this is not strictly a function.

21.1 userinput

The `userinput` function asks the user for an input on the command line. It returns the value entered. The prompting message may be specified in the `prompt` attribute.

```
anything extends userinput {
    prompt "Enter any value";
}
```

This will result in *anything* being set to whatever has been entered on the command line. A default value may also be set using the attribute “default”. This function is not really meant for serious use, but more for experimentation and testing.

21.1 System

The System function replaces the component description which represents the function with the value of its system attribute.

```
someAttr 65;

something extends System {
    a 45;
    b ATTRIB someAttr;
    system (a + b);
}
```

This will result in something being replaced by the value of its system attribute; in this case 110. This can almost be treated as a “let” construct – allowing the definition of locally named data which is discarded when the final result is resolved.

22 Schemas

Schemas are descriptions that may be attached to other descriptions and cause them to be checked against the schema description. Schemas are evaluated as part of the standard predicate phase.

Schemas are defined by extending the predefined template Schema, defined in the file */org/smartfrog/predicates.sf*:

```
mySchema extends Schema {
     // schema entries
}
```

Each of the schema entries are attributes whose names are to be found in the template to be validated. Each of these entries must extend a description that defines certain properties about the attribute. The properties are

- `optional`: a Boolean that states whether the attribute is optional or compulsory
- `binding`: a string which defines whether the attribute must be lazy `"lazy")`, must be eager (`"eager"`), or may be either (`"anyBinding"`) – this controls whether a link may exist instead of a value of the correct class
- `class`: a string which defines the name of the class which should be found as the value of the attribute (e.g. `"java.lang.Integer"`), or any class (`"anyClass"`).

Thus entries in a schema for a web server component may be

```
webServerSchema extends Schema {
    port extends {
          optional false;
          binding "anyBinding";
          class "java.lang.Integer"; }
    directory extends {
          optional true;
          binding "anyBinding";
          class "java.lang.String"; }
}
```

However this is rather cumbersome, so some helper templates are defined in the include file. These are defined as follows, with the obvious meanings.

```
Compulsory extends {
    optional false;
    binding "anyBinding";
    class "anyClass";
}

Optional extends {
    optional true;
    binding "anyBinding";
    class "anyClass";
}

OptionalBoolean extends Optional {
    class "java.lang.Boolean";
}

Boolean extends Compulsory {
    class "java.lang.Boolean";
}

OptionalInteger extends Optional {
    class "java.lang.Integer";
}

Integer extends Compulsory {
    class "java.lang.Integer";
}
```

```
OptionalDouble extends Optional {
    class "java.lang.Double";
}

Double extends Compulsory {
    class "java.lang.Double";
}

OptionalLong extends Optional {
    class "java.lang.Long";
}

Long extends Compulsory {
    class "java.lang.Long";
}

OptionalFloat extends Optional {
    class "java.lang.Float";
}

Float extends Compulsory {
    class "java.lang.Float";
}

OptionalString extends Optional {
    class "java.lang.String";
}

String extends Compulsory {
    class "java.lang.String";
}

OptionalVector extends Optional {
    class "java.lang.Vector";
}

Vector extends Compulsory {
    class "java.lang.Vector";
}

OptionalReference extends Optional {
    class "org.smartfrog.sfcore.reference.Reference";
}

Reference extends Compulsory {
    class "org.smartfrog.sfcore.reference.Reference";
}

OptionalCD extends Optional {
class "org.smartfrog.sfcore.componentdescription.ComponentDescription";
}

CD extends Compulsory {
class "org.smartfrog.sfcore.componentdescription.ComponentDescription";
}
```

These templates allow for a neater definition of the schema given above;

```
WebServerSchema extends Schema {
        port extends Integer;
        directory extends OptionalString;
}
```

To attach a schema to a description, the schema need only be an attribute within the description to which it applies. Thus we can complete the above example as follows:

```
 // the definition of schemas
#include "/org/smartfrog/predicates.sf"

WebServerSchema extends Schema {
        port extends Integer;
        directory extends OptionalString;
}

WebServerTemplate extends Prim {
        schema extends WebServerSchema;
        // default value
        port 80;
}
```

Note that the name for the attribute linking the template to its schema need not be, as in this case, `schema`. Indeed, a template may have more than one schema attached as attributes, in which case the uses of the template are checked against all schemas attached.

Schemas may be extended in the same way as other templates, and their uses may easily be extended through placement as illustrated in the following examples.

```
#include "/org/smartfrog/predicates.sf" // the definition of schemas

ThreadedWebServerSchema extends WebServerSchema {
        minimumThreads extends Integer;
}

ThreadedWebServerTemplate extends WebServerTemplate {
        //overwrite with extended schema
        Schema extends ThreadedWebServerSchema;
        minimumThreads 7;
}

AlternativeThreadedWebServerTemplate extends WebServerTemplate {
        // add to existing schema
        schema:minimumThreads extends Integer;
        minimumThreads 7;
}
```

Note that schemas are entirely optional and need be used only if desired. They carry no overhead during deployment and at run-time, but they can be expensive at language processing time.

`Prim` and `Compound` templates both have schemas associated with them. This can be useful to locate errors, but they also have two other effects. Firstly, when the descriptions are printed in any expanded form the schemas occupy a rather large amount of the overall description and can hide the structure of a description. Secondly, they can make a large description very expensive to process. Consequently, an attribute can be set at the top level to control whether these schemas should be included. Thus the following would switch off the use of the Prim and Compound schemas:

```
sfSchema false;
```

Part1: The SmartFrog Data Model

23 Mapping to the Core Data Model

The SmartFrog run-time and deployment system accepts data structures that have a specific structure defined through the use of Java interfaces and classes. These interfaces and classes are described in this part of the reference manual.

The attribute sets produced by the language processing phases are mapped into the core data structures supported by the SmartFrog runtime as the last step of the processing. These data structures do not support sf language features such as extension, placement, functions or predicates – so all these are eliminated as part of the processing. Links are supported, but these are by definition all `LAZY` links.

Once processing is complete the translation into these core data structures is straight-forward apart from one additional point: the structures produced by the phases can share data, such as when links point to the same value, but this is eliminated by copying the data as part of the mapping to the core data model. If this copying involves component descriptions, these are also parented into the part of the tree into which they are being copied.

The reason for this sharing elimination is to do with the semantics of the distributed system. Whilst all the data is local it could make sense to share data as it is more efficient, although care has to be taken when data is changed behind the scenes with side-effects on other parts of the tree. However, when parts of the data get mapped to different Java processes during deployment, the data has to be copied and the sharing broken in any case. To ensure a common semantics between local and remote deployments, separate copies are taken at all times.

This sharing elimination is illustrated by the following diagram. Note that the parent link from back from the `foo` attribute's data only exists if the attribute is itself an attribute set (a component description).

24 The Core Data Model

This section describes the data structures produced after the complete cycle of language processing. These are the structures that are understood and accepted by the SmartFrog run-time system.

The primary data structures that are generated as the output of this process implement the interface `ComponentDescription`, and it is this interface that users must understand to be able to create interesting tools or components. These data structures define the concept of an ordered attribute set.

In addition, the classes that are used to represent the various attribute values need to be considered: both the basic values such as Integers and Booleans, and references (all `LAZY` by this time).

In all cases, these interfaces and classes are fully defined in the accompanying Javadoc. The description provided here is only partial and is to give an overall feeling for the overall structure of the Java representation. The details of exceptions should also be obtained from the Javadoc.

Basic Values

Each of the basic values that have a syntax in the SmartFrog notation are mapped to different classes in Java. Wherever possible, they are mapped directly to the most obvious class in Java.

- Numbers are mapped to the equivalent Java subclass on `java.lang.Number`.
- Booleans are mapped to the class `java.lang.Boolean`.
- Strings are mapped to the class `java.lang.String`.
- `NULL` is mapped to the class

  `org.smartfrog.sfcore.common.SFNull`

  There is exactly one instance of class `SFNull`.
- Vectors are mapped to `java.util.Vector`.
- Byte arrays are mapped to the class

  `org.smartfrog.sfcore.common.SFByteArray`

  From which the actual byte array may be obtained.

  Instances of this class are immutable.

Values of other types can be contained within component descriptions, but these may not be properly handled by some of the operations in SmartFrog. In particular, although user-defined SmartFrog functions could could in principle return values of any class, and these would be patched into the attribute tree, when the conversion to the core data structures these will be rejected. Values of arbitrary types can be serialized into byte arrays, and then extracted and de-serialized at the appropriate time.

References

The class `org.smartfrog.sfcore.reference.Reference` is the Java class for defining references. References are lists of elements of the abstract class `org.smartfrog.sfcore.reference.`*`ReferencePart`*, each indicating a single step in the resolution that must occur.

The only interesting methods are those for constructing references, namely the constructors and the static method *fromString*. The other methods are typical of those required for manipulating lists, such as adding and removing parts and enumerating over the elements.

There are a number of constructors for references – one for an empty reference (one with no parts) and one for constructing a reference with a single part. Other parts must either be added to these basic references, or the reference may be created by parsing a string.

ReferencePart is the parent class of all reference parts, there is one per syntactic reference part (ROOT, ATTRIB, etc.). Again, the main interest is in the constructors for these. There are a couple of static helper methods for their construction defined in ReferencePart.

The following examples shows how to construct references in different ways:

```
import org.smartfrog.sfcore.reference.Reference;
import org.smartfrog.sfcore.reference.ReferencePart;

//parse a string - expensive but simple
Reference ref = Reference.fromString("x:y");

//use constructor with single part, then add another,
//using static helper for attrib part, and constructor for here part
Reference ref1 = new Reference(ReferencePart.attrib("x"));
ref1.addElement(new HereReferencePart("y"));

//use constructor with no part, adding each in turn,
//using static helpers for attrib and here parts.
Reference ref2 = new Reference();
ref2.addElement(ReferencePart.attrib("x"));
ref2.addElement(ReferenePart.here("y"));
```

For a complete list, the Javadoc for Reference and ReferencePart should be consulted.

In addition to the basic reference construction methods there are methods for getting and setting the flags (DATA and LAZY), getting and setting optional values for failed reference resolutions, iterating over the reference parts, copying and equality checking, and so on. However user code rarely needs to use these methods. When required, the details can be found in the Javadoc for Reference.

Component Descriptions

A ComponentDescription is an interface, with default implementation ComponentDescriptionImpl, which represents the concept of an attribute set in the syntax. Consequently, it has a number of methods that enable the creation and traversal of the containment and extension hierarchies.

ComponentDescription, in addition to defining its own methods, extends an number of additional interfaces, two of which merit further description: ReferenceResolver and Copying.

A base implementation of the interface ComponentDescription is the class ComponentDescriptionImpl is provided by the framework. This class may be generated directly by the language processor, or users may produce a class which extends it in some way.

The interface and implementing class can be considered in four parts:

1. Construction – how to create an instance of the class (other than through parsing a text stream)

2. the core interface for construction and traversal.

3. a copying interface which provides a deep copy operator essential when handling descriptions.

4. a reference resolution interface, defining methods to look up attribute values given references that describe paths through a description hierarchy.

24.1 Construction

24.2 Core

TO BE DONE

cover add/remove/replace attribute, iterate over attributes, get the parent.

24.1 Copying

The interface defines two methods of note – a deep copy operator that returns an equivalent structure of data and a clone method that returns a shallow copy. The copy method is recursive, in that it clones the top level component description, then embeds within it all the data contained in the copied description - invoking the copy method first if this data implements the Copying interface.

```
public Object copy();
    Produce a deep copy of the component description
```

```
public Object clone();
    Produce a shallow copy of the component description
```

24.1 ReferenceResolution

The reference resolution interface contains a number of methods to locate attributes within the hierarchy of component descriptions. The main method provided is the following:

```
Object sfResolve(Reference r)
    Resolve a given reference in the ComponentDescription hierarchy
    starting from this component.
```

In addition to this method, there is a whole family of variants, such as methods which take strings containing the textual syntax for references rather then references themselves, or define the specific return type so that users can avoid the class-cast, and so on. These are fully documented in the Javadoc.

Part1: Working with the Parser

25 Background

Given an term of the SF language, the question is how this is converted to the components of the data model defined in REF. Normally, this will be done by one of the command-line programs supplied as part of the SmartFrog system – such as the daemon itself reading its initial configuration file `default.sf` . However occasionally it is useful to be able to invoke the parser and language processing programmatically, for example to parse a string generated dynamically as part of the behaviour of some system.

Also, it is sometimes necessary to write new functions and predicates, or to provide additional phases to process the descriptions, thus customizing the normal tools provided by SmartFrog. This part of the reference manual deals with these aspects.

To a limited degree, the parser interface is language neutral. The API certainly supports multiple languages, the problems lie in the interfaces required by the nodes of abstract syntax tree for the phase mechanism. This may mean that a language that is significantly different to the core sf language may need to skip the phase mechanism and carry out all its tasks during the basic black-box "parsing" operation itself and not expose additional phases.

Also, defining functions, predicates and specific phase-implementing code is clearly not language neutral – so the reference manual only covers this process for the core default sf language.

Immediately after parsing, an abstract syntax tree is created for the text - the top node of which must implement the `Phases` interface. This interface defines how the rest of the language processing progresses by providing a set of methods for applying one or more tree-processing phases and eventually converting to the form defined in `REF` – the core data model.

26 The Phases Interface

The purpose of the Phases interface is to provide an abstraction of the processing steps to convert the abstract syntax tree which is the result of parsing, to the core data model.

The interface is `org.smartfrog.sfcore.parser.Phases` and documentation for it can be found in the accompanying Javadoc, however for completeness here its definition is:

```
package org.smartfrog.sfcore.parser;

import java.util.Vector;

import org.smartfrog.sfcore.common.SmartFrogCompilationException;
import org.smartfrog.sfcore.common.SmartFrogException;
import org.smartfrog.sfcore.componentdescription.ComponentDescription;


/**
 * Defines the Phases interface.
 * Objects that implement this interface are created by the parser.
 * The phases may then be invoked and the finally the
 * resultant Phases instance may be converted to a simple class
 * implementing only ComponentDescritpion for handing to the
 * SmartFrog deployment engine.
 *
 */
public interface Phases extends ComponentDescription {
    /**
     * Apply all the phases required of the description implementing
     * the interface. The list of phases is defined as a default for
     * the language used, or defined somehow as an attribute.
     *
     * @return An instance of Phases that is the result of applying all
     * the defined phases
     *
     * @throws SmartFrogException error evaluating phases
     *
     */
    Phases sfResolvePhases() throws SmartFrogException;

    /**
     * Apply the phase given in the parameter.
     *
     * @param phase the phase to apply
     *
     * @return An instance of Phases that is the result of applying the
     * phase.
     *
     * @throws SmartFrogException error evaluating phases
     */
    Phases sfResolvePhase(String phase)
        throws SmartFrogException;

    /**
     * Apply the phases given in the parameter.
     *
     * @param phases the phases to apply
     *
     * @return An instance of Phases that is the result of applying the
     * phase.
     *
     * @throws SmartFrogException error evaluating phases
     */
    Phases sfResolvePhases(Vector phases)
        throws SmartFrogException;

    /**
     * Return the phases required to be evaluated.
     *
     * @return the standard phases to apply
     */
    Vector sfGetPhases();

    /**
     * Convert the Phases (resulting from applying the phases) to a
     * ComponentDescription ready for the SmartFrog deployment engine.
     *
     * @return the convertion to a component description
     *
```

```
     * @throws SmartFrogCompilationException error converting phases to
     * a componentdescription
     */
    ComponentDescription sfAsComponentDescription() throws
                SmartFrogCompilationException;
}
```

Notice that the `Phases` interface extends that of `ComponentDescription`, thus it is clear that the language processing assumes that the abstract tree is already close to the final form.

There are basically two classes of methods:

- those that deal with applying phases
- the single method to convert from the possibly processed abstract syntax tree to the final form implementing the core model.

Each language may define its own set of phases, and indeed this can vary on a per-description basis. For example, an sf language description may contain the set of phases that should be applied to the description as an attribute. For any specific language, the `sfGetPhases` method should take all of this into account.

27 Phases for the "sf" language

Phases are a way transforming the sf language parse tree into the final form ready for deployment (or other purpose). Each phase is a pass over the component description hierarchy carrying out an action controlled, in the case of user-defined phases, by attributes defined within the descriptions.

Under normal circumstances users will not need to know about phases or how to modify on adapt them, the default collection of phases is already correct for most purposes.

The predefined phases for the default language are as follows:

- `type` – carry out type resolution on the component description hierarchy; this is predefined and does not rely on attributes in the tree to trigger it.

- `place` – carry out place resolution on the component description hierarchy; this is predefined and does not rely on attributes in the tree to trigger it.

- `link` – carry out link resolution on the component description hierarchy; this is predefined and does not rely on attributes in the tree to trigger it.

- `sfConfig` – not really a phase, rather it controls where the phases are applied. Its effect is that for the remaining phases in the current phase list, they are only applied to the `sfConfig` attribute.

- `print` – again, not really a phase, but it triggers the printing of the tree to the standard output. This provides a debugging mechanism as it can be placed between any other phases to view the intermediate state of the tree.

- `function` – in reality a user-defined phase, but one which is provided by default. It causes all the uses of functions and predicates defined as component descriptions to be converted to canonical form. It is triggered in the same way as the other user-defined phases, by the occurrence of attributes with the name *`phase.function`*.

Phases are triggered in a specific order, as determined by the top-level attribute ***`phaseList`***. If the attribute is not present, for the default "sf" language it is as though the attribute were defined as follows:

```
phaseList ["type", "place", "function", "sfConfig", "link"];
```

This default definition provides the semantics described in the section .

28 User defined phases for the sf language

In addition to the pre-defined phases, a user may introduce their own. User phases are defined as follows:

- A class must be created which implements the interface `PhaseAction` in package `org.smartfrog.sfcore.languages.sf`. The interface is fully defined in the Javadoc, but in summary, it provides two methods:
    - `forComponent` – which initializes the instance of the action with the component description on which it is to operate
    - `doit` – which triggers the action of the phase,
- In whichever component description the action must take place, an attribute whose name starts with the string `phase.nnn` must be provided, set to the string containing the class name, where `nnn` is the desired name of the phase.
- The `phaseList` attribute must be set at the top level of the description, containing the phase name `nnn` at the appropriate point relative to the other phases. The `phaseList` attribute must be a primitive vector – see example below. An alternative is to invoke the parser in user code and apply a known set of phases without relying on the `phaseList` attribute.

There are a few points to notice. Firstly, the descriptions are traversed depth-first so the inner descriptions are visited before the outer. This makes sense for functions, for example, that are evaluated from the inside. The second point is that the action is independent of the phase, in that the attribute name determines the phase; the action is determined by the attribute value. Thus, it is possible for the same action to be used in two different phases, and for different actions to be invoked in the same phase – as is the case with functions and predicates where different actions are invoked for the different canonical forms. It is also possible to have more than one action for each phase in a component description since the attribute name merely needs to start with the `phase.nnn` string so several may be provided.

Note that both the `phaseList` attribute is removed, as are the `phase.nnn` attributes after each action is invoked.

Example

Consider the following example. A class is provided that adds the `sfProcessHost` attribute (used to determine on which host a component should be deployed - REF) to a component description, based on the value of an attribute `sfLogicalHost`. It maps the logical host to the physical host in some way not defined here – say by using the method `mapHost`.

The class might be defined as follows (ignoring possible errors):

```
package org.smartfrog.example;

class MapHost implements PhaseAction {
    ComponentDescription cmp = null;

    public void forComponent (ComponentDescription c) {
        cmp = c;
    }
```

```
    public void doit() {
        String logicalHost =
                  c.sfResolve(Reference.fromString("sfLogicalHost"));
        c.sfAddAttribute("sfProcessHost",  mapHost(logicalHost));
        c.sfRemoveAttribute("sfLogicalHost");
    }

    private String mapHost(String logical) { … }
}
```

This class may then be used in a description, to be acted on in the phase `mapHosts`, as follows

```
phaseList [|"type","place","function","sfConfig","link","mapHosts"|];

MappedCompound extends Compound {
  phase.mapHosts "org.smartfrog.example.MapHost";
}

sfConfig extends MappedCompound {
    sfLogicalHost "databaseHost";
    component1 extends Prim { …}
    component2 extends Prim { … }
}
```

The phase list adds the mapping phase to the end, providing for the host mapping. The `MappedCompound`, when used, carries its phase attribute with it. Consequently, it is now contained within `sfConfig`. Thus during that last phase, `sfConfig` will be mapped to the correct physical host.

29 Functions

Function usage defined through the use of component descriptions are converted to canonical form during a predefined phase, named `function`, with the effect that an attribute obtains the canonical form of the function usage. To make functions easier to write, a predefined abstract `PhaseAction`, called `BaseFunction` from package

```
org.smartfrog.sfcore.languages.sf.functions
```

is provided that makes writing new functions easier.

New functions should extend the class `BaseFunction` and provide the method `doFunction()`, returning the result of the function as an `Object`. Any attribute may be accessed during the evaluation process.

`BaseFunction` is documented in the Javadoc and predefined functions are documented in REF.

Predicates

Predicates usage defined through the use of component descriptions are converted to canonical form during a predefined phase, named `function`, with the effect that an attribute obtains the canonical form of the predicate usage. To make user-defined predicates easier to define, a class `BasePredicate` from package

```
org.smartfrog.sfcore.languages.sf.predicates
```

is provided that makes writing new predicates easier.

New predicates should extend the class `BasePredicate` and provide the method `doPredicate()`, throwing the exception

```
SmartFrogCompileResolutionException
```

if there is an error. Any attribute may be accessed during the predicate evaluation.

`BasePredicate` is documented in the Javadoc and the predefined predicate *Schema* is documented in REF.

30 Invoking the Parser

Background

The SmartFrog framework is designed to support a range of possible languages to define configurations for the deployment engine to instantiate. The languages are all required to follow a common model for their processing, and to eventually produce data structures that are suitable for the deployment system. The default language is the base SmartFrog language defined above, and which uses the file extension "`.sf`".

The first stage of language processing is the parser – a tool for turning text into data structures for further processing. The parser interface allows programmers to select the parser based either on the language type of the file (as defined by file extension), by direct selection, or simply using the default (sf) parser.

After parsing, the data structures produced must implement an interface for driving the remaining resolution phases. This interface is

```
org.smartfrog.parser.Phases
```

Following the invocation of the various phases, the data is converted into a hierarchy of data supporting the `ComponentDescription` interface, which may then be passed to the deployment system.

Using this model, it is reasonably easy to define a new language and integrate it into the system. The default SF language is the first such, but others such as XML based languages, or the more advanced SF2 language currently under development are also possible.

The remainder of this section describes how to invoke the parser, how to step the language data structures through the various processing phases, and finally the nature of the resultant `ComponentDescription` data structures.

Summary of Language Processing

All of the tools provided with the SmartFrog system handle a SmartFrog text in an identical way to produce a fully resolved deployable description. The process is basically:

- parse the text stream to produce hierarchical data structures
- carry out all the phases, which for the default primary language are
    - type resolve the root
    - place resolve the root
    - convert functions and predicates to canonical form in `sfConfig`
    - extract attribute `sfConfig` from the root
    - link resolve `sfConfig` thereby dereferencing links, evaluating functions and checking predicates
- convert to standard data model, creating simple normalised attribute tree

The Parser

The SmartFrog parser is implemented as a Java class with a method to parse an `InputStream` producing an instance of the class `ComponentDescription`, the Java class representing the parsed text allowing programmatic manipulation of the information. Any `InputStream` may be used, thus the parser may be invoked on a `String`, a `File`, a `URL`, or any indeed any object that provides a stream model.

During parsing, a number of include files or URLs may be specified indicating text that should be included into the current parse. It should be noted that unlike C, the text is not merely embedded into the source text, rather the files are parsed independently by the parser and the consequent data embedded into the resultant `ComponentDescription` data structure produced by the initial stream. Note that in principle, the parsers of include files may be different from the parser for the main stream, thus providing a means for including files in different notations. However, the mechanisms for doing so are not covered in this manual.

30.1 The Parser API

Under normal circumstances, users of SmartFrog will not be expected to use the parser directly. Rather the parser will be invoked on the users behalf by the tools and scripts provided to start and run the SmartFrog framework. However, just in case the need arises to invoke the parser within user code the parser API is now described.

Two aspects must be considered:

1. Ensuring that security properties are maintained: if security is required, the appropriate actions should be taken to ensure that only streams from signed and trusted sources are used.

2. Invoking the parser itself on the stream.

The security model is covered in section 43, and this should be read in detail before implementing any secure code, however enough of the security API is defined here for completeness.

30.1 Ensuring Security

Two important steps must be carried out to ensure that the security of the SmartFrog framework is not compromised. The first is to initialize the SmartFrog security infrastructure, if this is not already done, and the second is to ensure that every resource (test file, URL, etc.) is loaded through the secure mechanisms provided.

Under normal circumstances, users will be using the parser from within the SmartFrog system itself; writing components that use the parser. However, just in case this is not so and security is still required, initializing the security mechanisms is carried out by invoking the `initSecurity()` static method on the `SFSecurity` class as follows:

```
import org.smartfrog.sfcore.security.SFSecurity;
…
SFSecurity.initSecurity();
```

Once the security has been initialized, streams may be created on strings or files as required. However, to ensure that security is maintained, it is important that the correct class loaders are used for accessing any external resources. This is achieved by using the following invocation to create a stream from the required resource:

```
import org.smartfrog.sfcore.security.SFClassLoader;
…
InputStream stream = SFClassLoader.getResourceAsStream(url);
```

30.1 Invoking The Parser

Once a stream is created, a parser instance may be created and the input stream parsed to generate the data model. This is done through the following code

```
import org.smartfrog.sfcore.parser.SFParser;
import org.smartfrog.sfcore.parser.SFPhases;

…
Phases component = new SFParser().sfParse(stream);
```

The `getParser()` method returns a parser for the currently selected language (by default the sf language) and this parser supports the `sfParse(InputStream s)` method to parse the input stream.

If the parser for a different language is required, say for the csf language, the following code is required

```
import org.smartfrog.sfcore.parser.SFParser;
import org.smartfrog.sfcore.parser.Phases;

…
Phases component = new SFParser("csf").sfParse(stream);
```

The parser is built for the correct language, then asked to parse a stream.

All the SmartFrog command-line tools that use URLs to identify descriptions to load examine the URL to determine from the extension which parser should be used.

30.1 Evaluating The Phases

Once the parser has completed, the resultant data structures must implement the `Phases` interface. Through the use of this interface the various phases of the language processing are carried out – either as a single step or by carrying them out one at a time. After each phase, date structures that implement the `Phases` interface must be returned.

The complete description of the API is given in the Javadoc, but the following examples are probably sufficient to illustrate the process.

To evaluate all phases in one go:

```
Phases phases = new SFParser().sfParse(stream);
phases = phases.sfResolvePhases();
```

To extract the phases, then apply them one at a time:

```
Phases phases = new SFParser().sfParse(is);
Vector thePhases = phases.sfGetPhases();

for (Enumeration e = thePhases.elements(); e.hasMoreElements();) {
     phases = phases.sfResolvePhase((String) e.nextElement());
}
```

Both of these mechanisms will use the phases given in the `phaseList` attribute defined at the top level of the stream if one is given, otherwise the default phase list for the specific language will be used.

It is possible to apply phases one at a time, or with a specific phase list generated in some other way, and thus not be dependant on the phase list given in the description itself.

All the command-line tools provided by the SmartFrog system use the mechanisms that read the `phaseList` or apply the default phase list if one is not supplied.

30.1 Converting to `ComponentDescription`

Before handing the data to the deployment system, the languages own data structures must be converted to those expected by the deployment system – namely the standard data model implementing the `ComponentDescription` interface (normally, but not necessarily) an extension of `ComponentDescriptionImpl`.

This is done using the `sfAsComponentDescription` method defined in `Phases`. The full code for parsing and processing a stream in the default language is

```
Phases phases = new SFParser().sfParse(stream);
phases = phases.sfResolvePhases();
ComponentDescription component.sfAsComponentDescription();
```

Part1: The Component Model

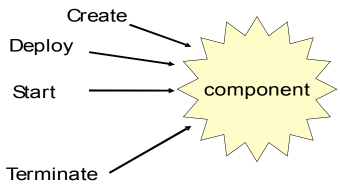
31 Introduction

Components

Having written the description of a service in a notation, and having had this processed by the appropriate language processor (done in most instances by the SmartFrog command-line tools), the simplified models are passed to SmartFrog to carry out the deployment process. This process consists of the hierarchical deployment of components as defined by the description. At this point, ignoring how the description defines which components should run where, it is worth defining the very notion of a component.

A component is an instance of a Java class that implements a specific Java interface: the `Prim` interface. The classes for components are normally implemented by extending one of the predefined classes that already provide most of the required semantics, adding to or modifying this behaviour as appropriate. This interface is quite complex and extensive, however most of it is for SmartFrog's internal operation. Programmers really only need to use a subset of these.

The most important aspect, and that which is core to the notion of a component, is that of the lifecycle and the methods associated with it. A component has a simple lifecycle to bring it into being, to start its activity, and eventually to terminate it. This lifecycle is implemented as a set of methods provided by the programmer (or through inheritance, by one of the classes provided as part of SmartFrog).



As illustrated by the diagram above, the lifecycle of a component is generally thought of as the phased start-up of first component creation, then component initialization (also know as deployment) and finally component initiation: starting the active parts of the component. Termination and clean-up can happen asynchronously at any time after the initial creation, triggered internally by the component, externally by the environment, for example in response to some form of failure.

In addition to considering a single component and its lifecycle, it is necessary to consider the lifecycles of collections of components. These collections might define the set of components in a specific application or service. As such, the sum of the lifecycles of these components effectively determine the lifecycle of the service itself. As an example, consider a service that manages web servers and creates or terminates web servers according to loading . Any single component has a simple start/stop lifecycle as determined by the lifecycle model given above, but the collection has a much more complex lifecycle involving the dynamic modification of the collection of web server components that are included in the service – sometimes adding and sometimes removing these web server components.

Now the lifecycle of services, or more generally the service of some collection of components, are mediated by other components – known as compound components – that make use of SmartFrog's APIs to interact with the SmartFrog framework to instantiate and terminate the components for which it is responsible.

Most such compound components are very specific to a service – for example the component that monitors the response-time of the web servers to decide whether to deploy or terminate a web server is a specific component to that service. However there are some generic types of collection that can also be defined and which can usefully be implemented as part of a core framework such as SmartFrog.

The most important such specific collection is the Compound (as opposed to the more general concept of a compound component). The Compound component is one where every member of the collection shares fate with the others in the group. Thus they are all created or none are, they are all initialized or none are, they all start together or none start, and they all terminate or fail together.

An example of where this Compound behaviour is appropriate might be where the collection consists of the component to collect service response data and the component that uses this data to decide how many web servers to be running. It would not be appropriate to tie the different web servers together in such a group so that terminating one web server would terminate them all.

All of these group behaviours are mediated by components that can be defined and then made available for use by other services wherever this is appropriate. In addition to the pre-defined compound components, such as Compound, users may define their own specific to their service.

32 The SmartFrog Component Model

This section describes how a set of attributes is interpreted by the SmartFrog framework as a collection of components distributed across a number of hosts in a network. There are several aspects to consider:

- how attributes indicate type of component.
- how attributes are used to indicate the location of components.
- how the system carries out the creation of components.
- how components are started and stopped.
- how components may use the framework APIs to access configuration values.
- how components may dynamically interact with the framework to dynamically modify the sets of components that are created.

First, however, it is worth a general discussion of the concepts behind the SmartFrog application model.

Applications As Component Collections

SmartFrog considers a whole system to be a collection of applications running over a distributed collection of compute resources. This collection of applications may be dynamic, generated on demand by a variety of external and internal events, such as a user request or a new resource being started.

Each application is, in turn, a collection of components defined statically via an application description or generated dynamically at run-time according to the requirements determined at that time. The components of an application may be dynamic, changing over time to adjust for circumstances.

These terms, namely *system*, *application* and *component*, deserve better definition to highlight their respective roles and the ways in which SmartFrog manages them. It is easier to consider them in reverse order.

Component

A component is defined as a single Java object which implements a specific API (defined in the `Prim` interface) and which consequently implements the specific lifecycle as defined by the SmartFrog component model.

Since the component is implemented as a single object, it resides entirely within one JVM on a single host. The component may, behind the scenes, create and manage other objects including other processes and programs written in other languages. However, for the purpose of SmartFrog, the management view of component is entirely defined by the `Prim` object.

Application

An application consists of a collection of components, and consequently it is not an atomic object as seen by SmartFrog. This means that the lifecycle is not viewed through a single interface, complicating the handling of the lifecycle of the whole application. An application has two characteristics that characterize the notion:

- Each component is tightly bound to the others via a parent-child relationship, each may be a parent to others and each (unless it is the root component) has a parent. This transitive closure of the parent-child relationship is the scope of the application.

- The lifecycle of each component in an application is tied to the lifecycle of the others via this parent-child relationship – parents are notified of child death, and vice-versa, and the parents are entirely responsible for the lifecycle of their children. The order of component start-up and termination is well defined and may be relied on for the simplification of component coding.

  Components of an application may locate each other, thereby enabling communication, using the built-in SmartFrog naming capabilities by following the parent-child relationships. These links between components are specified via `LAZY` links.

System

A system is simply a collection of applications, loosely grouped over the distributed resources. Applications within a system do not have direct links between their components, nor any direct responsibility for, nor notification of, their respective lifecycle (though these may be implemented within specific component behaviour).

Typically, applications locate each other through naming or discovery services, and they must be able to cope appropriately with the non-existence of applications on which they depend, both at start-up and during operation.

Applications and Component Descriptions

As described above, each application is a collection of components connected via a parent-child hierarchy. Thus an application is a tree structure similar to the tree structure present in a component description hierarchy, as described in section 24. Indeed, it is the role of the SmartFrog system to convert a description into the equivalent running application according to an appropriate interpretation of attributes present in the description.

The process of taking a description and converting it into a running application is defined roughly as follows:

- A description in SmartFrog notation is parsed and resolved.

- The specific *interesting* component description is selected, namely the `sfConfig` component description.

- The result is a hierarchy of component descriptions (each extending `NULL`, the empty description).

- The role of SmartFrog is to create an equivalent hierarchy of objects, as determined by the attributes present in the component description hierarchy.

Consider the following small example:

```
StatusMonitor extends Prim {
    sfClass "org.smartfrog.examples.StatusMonitor";
}

LogMonitor extends Prim {
    sfClass "org.smartfrog.examples.LogMonitor";
}

sfConfig extends Compound {
    log extends LogMonitor;
    status extends StatusMonitor;
}
```

`Prim` and *`Compound`* are described in detail in section 32.1, but in effect define collections of attributes that describe leaves (*`Prim`*) and nodes (*`Compound`*) of a component hierarchy. `Prim` is a collection that defines some basic attributes

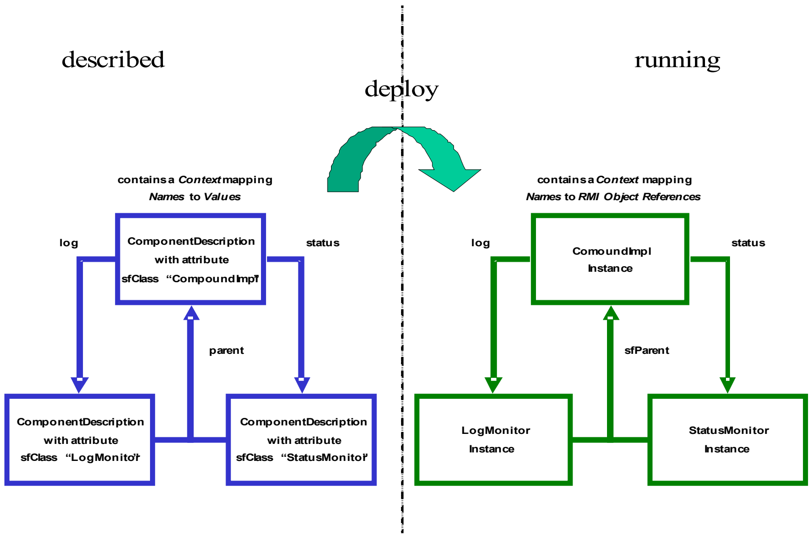
relevant to all components (such as default code base), and Compound contains at least the additional attribute:

```
sfClass "org.smartfrog.sfcore.compound.CompoundImpl";
```

After all the resolution steps, and after selection of the sfConfig attribute, the following description is obtained:

```
sfConfig extends {
    sfClass "org.smartfrog.sfcore.compound.CompoundImpl";
         // plus Prim attributes
    log extends {
        sfClass "org.smartfrog.examples.LogMonitor";
                          // plus Prim attributes
    }
    status extends {
         sfClass "org.smartfrog.examples.StatusMonitor";
                           // plus Prim attributes
    }
}
```

This is a hierarchy of component descriptions. The SmartFrog framework takes this description and turns it into the following running application: a component parent-child hierarchy using the sfClass attribute to determine the class of object to instantiate.



In both cases, the described and running, a *Context* (section 24) is used to store the attributes. In the described case, the values are the component descriptions of the next level in the tree, whereas in the running case the same attributes hold the RMI references to the running components represented by the equivalent component description.

Consequently, the application forms a naming structure mirroring the one in the description and indeed, references may be used to traverse the structure in exactly the same way as with the descriptions. The difference is that once the application is deployed, references to components return the RMI object reference to the component rather than a copy of the attributes in the referenced description.

Representing Components With Attributes

Applications are described using the SmartFrog notation. To create an application, its description is parsed, resolved and the sfConfig attribute extracted. The ComponentDescription obtained is given to the framework to create and manage the components associated with that description. The means by which the SmartFrog framework does so depends on the attributes that are present within the ComponentDescriptions.

Each description that is intended to represent a running component has two types of attributes, though these are indistinguishable within the notation. These are

- the template attributes that define the Java class of the component, where it should be created, and certain other management aspects; these are all identified by their "well-known" names. All template attributes start with the letters *sf*.

- the component configuration attributes containing configuration information to control the component behaviour; these are all the attributes other than the template attributes, and their interpretation is component specific.

This section is concerned only with the template attributes. The template attributes may be split into several categories, the most important of which are described in the following sub-sections.

32.1 Defining the Component Class

Each component created by SmartFrog is an instance of a class that implements the `Prim` interface (and usually extends the class `PrimImpl`). The key attribute of any component is therefore defining the identity of that component class. This is done by providing the well-known attribute `sfClass` holding a string representing the full package and class name of the component. Thus the description:

```
sfConfig extends {
    sfClass "org.smartfrog.example.AClass";
}
```

defines an application with a single component, an instance of the class `AClass` in the package `org.smartfrog.examples`.

The code for the component class is normally loaded from the codebase defined at the point of launching the daemon. However it is also possible to define a codebase for a specific component, or a whole sub-tree of a description, by setting the `sfCodeBase` attribute within the definition. This also affects all sub-components, which may reset the codebase to the default by setting the `sfCodeBase` attribute to the string `"default":`

```
sfConfig extends {
    sfCodeBase "default";
}
```

Note that the `sfCodeBase` defines an additional location where a class may be found – the default codebase is still searched for the definition.

32.1 Controlling Deployment

In addition to specifying the class of the component, it is necessary to define where the component is to be created. Components are created in SmartFrog processes known as *daemons*. Daemons have a variety of flavours, however all provide the ability for the SmartFrog infrastructure to request the creation of new components. The two primary flavours are the **root** daemon that must run on each host, started manually or automatically at boot time, and named sub-daemons that may be created on demand by the SmartFrog framework.

SmartFrog provides a mechanism by which daemons may be identified via a combination of host and process names, however it also provides the means by which other location identification mechanisms may be implemented; an example of this being the use of SLP.

The two key attributes for identifying locations are the `sfProcessHost` and the `sfProcessName` attributes, both strings identifying the host and process respectively. If neither attribute is present, the current process is assumed; the current process being whichever is currently carrying out the deployment. If the `sfProcessHost` attribute is present, but the `sfProcessName` is not, the root daemon is assumed. Clearly any such use of the `sfProcessHost` attribute assumes that a root SmartFrog daemon is running on the identified host, and that security settings of the remote host permit the local system to deploy to it.

Thus, the following example will create the example component in the root process on the specified host.

```
sfConfig extends {
    sfClass "org.smartfrog.example.AClass";
    sfProcessHost "15.144.56.243";
}
```

and the following will place the component into a sub-process named `ExampleProcess`, creating it if it does not already exist.

```
sfConfig extends {
    sfClass "org.smartfrog.example.AClass";
    sfProcessHost "15.144.56.243";
    sfProcessName "ExampleProcess";
}
```

32.1 RMI

SmartFrog relies on Java/RMI to provide the transport layer for carrying out management tasks, such as initiating the creation of a component. SmartFrog also makes it easy to use RMI to provide inter-component communication for application purposes. In particular, the RMI reference of a component is returned whenever a `LAZY` link is dereferenced to point to a component. However, it is an overhead for all components to be full-blown RMI servers; it is only necessary that the first component of any hierarchy of components in a daemon be a server, though others may be so if required.

If it is unnecessary for a component to be *exported* as an RMI server, the *`sfExport`* attribute may be set:

```
sfConfig extends {
    sfClass "org.smartfrog.example.AClass";
    sfExport "false";
}
```

It is equally possible to set it to true to ensure that the RMI export is done, although this is also the default if the attribute is not explicitly set. If the export is set to `"false"`, only components in the same JVM will be able to access the component.

32.1 Prim and Compound

In the notation, the root of all extensions is the `NULL` attribute set; this is normally indicated by extending nothing. However, when components are being described (as opposed to arbitrary structured data) it is conventional to use the description `Prim` as the root of all descriptions. This provides a single place where default-valued template attributes may be placed, thus guaranteeing that all components inherit them.

Prim is defined in the file `org/smartfrog/components.sf`. Thus, this must effectively be included in every description, as in the following example.

```
#include "/org/smartfrog/components.sf"
sfConfig extends Prim {
    sfClass "org.smartfrog.example.AClass";
}
```

In addition to `Prim`, there is a standard component called `Compound` defined in the same `components.sf` file. This provides the definition of the most commonly used application-grouping component, which creates a set of components as part of an atomic collection. Thus, most applications are similar to the following example in their use of `Prim` and `Compound`:

```
#include "/org/smartforg/components.sf"
Service extends Prim {
     sfClass "org.smartfrog.example.Service";
     // and other attributes
}
ServicePair extends Compound {
    service1 extends Service{ … }
    service2 extends Service { … }
}
sfConfig extends ServicePair {
   sfProcessHost "localhost";
}
```

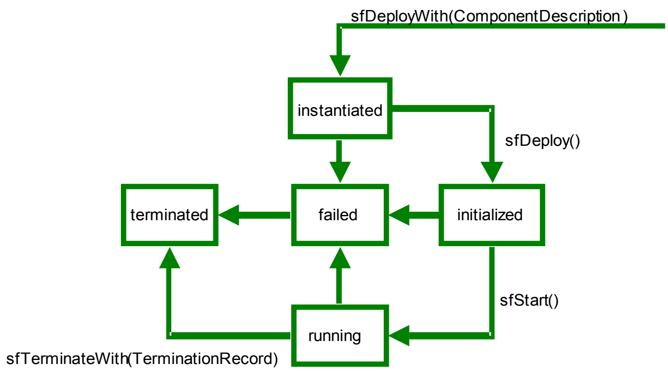
The `Compound` description is unlike `Prim`, in that it contains an `sfClass` attribute defining the compound implementation class, which means that it is directly instantiatable in a deployment. Otherwise, it is empty but may be used to define default template attributes for all compounds.

```
Compound extends Prim {
     sfClass "org.smartfrog.sfcore.compound.CompoundImpl" ;
}
```

Compounds are described further in section 34.

Lifecycles

An important aspect of the SmartFrog component model is the lifecycle. The lifecycle is implemented as a simple state machine, shown in the following diagram.



The transitions in the state machine are associated with actions implemented (if required) by the components. The transition actions are implemented by the invocation of methods on the component, during which the component may take any appropriate action. These transition methods, also known as the template methods, are indicated along side the transition in the diagram.

These methods are described more fully in section 33. Default minimal actions are provided if the component has no specific action to carry out during the transition.

In an application with multiple components, the lifecycle of the whole system is defined by the combination of component lifecycles in some order. It is not completely defined within SmartFrog as to how these lifecycles are composed as it depends on the specific components used, and specifically those that are compound components.

The root parent of the parent-child hierarchy is controlled by the framework, and it is responsible for triggering the lifecycle transitions. It is the responsibility of that component to transition each of its children. The most common component used as root, and as intermediate nodes of the hierarchy is the `Compound` and this defines a simple combined lifecycle aimed at providing the notion of a single atomic composition with a shared lifecycle:

- on transition to the *instantiated* state, all children are similarly instantiated;
- on transition to the *initialized* state, all children are initialized;
- on transition to *running*, all children are started;
- on transition of any component to *terminated*, all children and the parent are terminated, and thereby the whole hierarchy is terminated.

This specific semantics has some important properties. The entire application is stepped through the lifecycle in a synchronous way: all components are created before any are initialized and all components are initialized before any are started. Termination is rather different as its occurrence is asynchronous with respect to the other transitions and any component in the hierarchy may be the first to transition to the terminated state (i.e. unlike the other transitions, it is not only the root which may initiate it). Nevertheless, the semantics are such that the whole application will terminate if any component terminates.

There are other possible semantics for parent-child lifecycle combination other than `Compound`, and some are provided by the framework. In particular, the workflow package provides a number of combinations such as parallel composition and sequential composition. The workflow package, documented in an accompanying manual, is designed to provide a simple lightweight workflow-style capability to the SmartFrog framework.

Only `Compound`, and its close relative `DetachingCompound`, will be considered within this manual, as it is the core implementation of component composition and is used by all others.

The SmartFrog API

The SmartFrog API is in essence the interface that each component must implement. This is the `Prim` interface (not to be confused with the `Prim` description) in the package `org.smartfrog.sfcore.prim`. All the SmartFrog capabilities are defined through the methods defined in this interface. To provide default implementations of these methods, all components should extend the class `PrimImpl` to define their default behaviour and enable access to the framework services.

The API is divided into two parts:

- The template methods that a component may wish to redefine; these are primarily the lifecycle methods.
- The utility methods that provide the component with the ability to access the SmartFrog framework capabilities, which includes access to the component’s configuration attributes. Equally they provide the SmartFrog system or other components access to the same capabilities on that component.

Both of these parts are defined in the `Prim` interface and are fully documented in the accompanying *Javadoc*; a partial description of the more important aspects is given in section 33.

33 Primitives

Primitives are the core of the SmartFrog component model, they are the basis for all the components that are created to implement application behaviour. Implementing a prim component in Java is the essence of SmartFrog programming.

Creating a prim class involves the following steps:

- Inheriting from the `PrimImpl` class and implementing the `Prim` interface (note that although `PrimImpl` itself implements `Prim`, it is necessary for RMI that the component directly declares that it implements it or another `Remote` interface.
- Providing an empty default constructor throwing the RMI remote exception, this constructor is used by SmartFrog to create the component instance.
- Providing the appropriate lifecycle template methods if they are required – default methods are provided by the `PrimImpl` class should no action be required at a specific point in the lifecycle.
- Implementing any component-specific interface required by the component.

The component-specific behaviour defined in the template methods or in the component-specific interfaces may use the utility methods.

Template Methods

The template methods are defined to be the ones shown in the lifecycle diagram, namely:

- `sfDeployWith(…)` is the method that is used alongside the default constructor for the basic creation of a component. It should not be overridden with application specific behaviour.
- `sfDeploy()` is the method that the system invokes to transition the component from the created state to the initialized state. Users may override this method to provide component specific initialization code, though care should be taken to invoke the superclass implementation to maintain correct behaviour. After this method, components should be ready to receive requests from other components in the application. Thus, listener threads must be started but threads that invoke other components may not. The component may assume that all other components in the same application have been created, but may not assume that they have been initialized.
- `sfStart()` is the method that the system invokes to transition the component from the initialized state to the running state. Users may override this method to provide component specific initialization code, though care should be taken to invoke the superclass implementation to maintain correct behaviour. The method will start any required active threads. The component may assume that all other components in the application have been initialized, but may not assume that they have been started.
- `sfTerminateWith(TerminationRecord tr)` is the method that may be called at any time to transition from any state to the terminated state. The termination record contains details of the reason for termination. All threads must be stopped, all resources must be

released, etc. No assumptions may be made as to the state of any other component in the application. The superclass termination method must be invoked to ensure correct behaviour.

Thus, a typical SmartFrog primitive has the following shape:

```
import org.smartfrog.sfcore.prim.Prim;
import org.smartfrog.sfcore.prim.PrimImpl;
import org.smartfrog.sfcore.prim.common.SmartFrogException;

import java.rmi.RemoteException;

public class MyPrim extends PrimImpl implements Prim, … {
  /* any component specific declarations */

  public MyPrim() throws RemoteException {
  }

  public synchronized void sfDeploy()
    throws RemoteException, SmartFrogException {
    super.sfDeploy();
    /* any component specific initialization code */
  }

  public synchronized void sfStart()
    throws RemoteException, SmartFrogException {
    super.sfStart();
    /* any component specific start-up code */
  }

  public synchronized void sfTerminateWith(TerminationRecored tr)  {
    /* any component specific termination code */
    super.sfTerminateWith(tr);
  }

  /* any component specific methods */
}
```

This template is absolutely the key to programming SmartFrog components. The three template methods may be left out if they are not required. Note the fact that all the three methods call the super-class method, and note that in `sfDeploy` and `sfStart` this is done before the component specific part, whilst with `sfTerminateWith` this is done at the end. The `sfTerminateMethod` must also be written so as not to fail if called before the component was actually started.

Utility Methods

The utility methods are defined in `PrimImpl` designed to provide a component programmer with the key utilities to enable interaction with the SmartFrog system. These are the ability to find and manipulate attributes, locate application components and to terminate itself and hence (under normal circumstances) the whole application.

The utility methods are primarily defined in `Prim`, however other methods are defined in the `RemoteReferenceResolver` interface that `Prim` extends. A brief overview of these methods is now given, however a more complete definition is provided with the *Javadoc*. The precise signature of these methods should also be checked in the *Javadoc*,

```
Object sfResolve(Reference r)
```

Locate an attribute when given a reference. This method resolves from the point in the component hierarchy that is the Prim on which it is called. References may be constructed using the class constructor (recommended) or the *fromString("…")* method defined on Reference to provide a simple, though inefficient, way of creating complex references. The class *Reference* is documented in section .

```
void sfTerminate(TerminationRecord tr)
```

Terminate the component and hence (under normal circumstances) the application. This will cause the termination template method to be called at some point in the future, though not necessarily immediately.

`void sfDetach()`

Split the application tree into two independent components; the parent-child link becomes severed and thus the lifecycles of the parent and the child components become separated. The child component becomes a root component, i.e. one with no parent.

`sfAddAttribute/sfReplaceAttribute/sfRemoveAttribute`

These are a collection of methods that allow the modification or removal of the component's attributes.

`Prim sfParent()`

Returns the parent of the component, or null if it is a root component.

`Context sfContext()`

Each component contains the context that was originally given to the component as its description. This method returns that context.

`Reference sfCompleteName() throws RemoteException`

Returns the complete name of the component from the root of the application. Throws exception if unable to trace the component hierarchy starting from the root.

`Reference sfCompleteNameSafe()`

Returns the complete name of the component from the root of the application and does not throw any exception.

These methods form the basis of the programming API for components providing interactions with the SmartFrog system. Other methods, to do with dynamically modifying the structure of the application hierarchy, such as deploying new branches of the tree, are covered in the section on Compounds, section 34.

34 Compounds

SmartFrog provides the ability to create collections of components with certain semantic guarantees. These collections – whatever their semantics – are known as compounds. SmartFrog provides a small number of these collections; others may be defined as required. The primary collection component is simply known as "Compound" and is the root of all other collections.

There are three aspects to consider with Compound

- The component description for compound – contained in the file "/org/smartfrog/components.sf".
- The interface Compound.java.
- the implementation of the base class of all compounds: CompoundImpl.java.

Compound Component Descriptions

The component description is an extension of the component description Prim and adds attributes relevant to the liveness mechanisms and, importantly, it adds the sfClass attribute to ensure that the compound component description generates an instance of the CompoundImpl class when deployed.

In order to define a collection of components, the description must extend Compound and add any Prims or Compounds that are required. For example

```
#include "org/smartfrog/components.sf"
sfConfig extends Compound {
        component1 extends Comopound {
            foo extends Prim { sfClass "…"; }
            bar extends Prim { sfClass "…"; }
        }
        baz extends Prim {sfClass "…"; }
}
```

defines the system to be a collection containing a sub-collection and a primitive. The sub-collection component1 contains two primitives. Note that the extension of the Compound does not contain an sfClass attribute as this is supplied by the definition of Compound in the include file.

The Compound Interface

The Compound interface, in the package org.smartfrog.sfcore.compound, is an extension of the Prim interface, hence any compound is by definition also a primitive component. It also defines a number of additional features:

- Compound extends the interface ChildMinder that provides methods for registering and removing child components. These methods will rarely be used by SmartFrog programmers as they are automatically invoked as required by the underlying system. These methods are documented in the Javadoc for the interface.
- Compound directly defines a number of methods to allow applications, should they desire it, to deploy additional application descriptions. Although in the normal course of events this will not be used directly by SmartFrog programmers, it is sufficiently frequently used to merit some further explanation here. There are three methods:

```
public Prim sfDeployComponentDescription(Object name,
```

```
                                          Compound parent,
                                          ComponentDescription cmp,
                                          Context parms)
               throws RemoteException, SmartFrogException;
public Prim sfCreateNewChild(Object name,
                             ComponentDescription cmp,
                             Context parms)
               throws RemoteException, SmartFrogException;
public Prim sfCreateNewApp(ComponentDescription cmp,
                           Context parms)
               throws RemoteException, SmartFrogException;
```

The first of these is primarily intended to be an internal method for use by the framework and should be used cautiously by programmers. The method is given a component description that has already been parsed and resolved and deploys it (i.e., creates and calls `sfDeployWith(...)`) with the following additional aspects:

- A parent, an existing `Compound` including the one implementing the called method, may be provided, or `null` if no parent is required. If no parent is provided, a wholly new application is created.
- The name by which this component is to be known within that parent compound – again may be null if a parent is not required
- A set of additional attributes that should be placed into the top-level component description before deployment.

This method is used by the SmartFrog system itself to build the complete component hierarchy. Note that since the remainder of the lifecycle is not invoked, this is up to the programmer to do, ensuring that no part-deployed components are left polluting the system. Users must also be careful of synchronization issues and ensure that lifecycle methods are not invoked twice by error (easy to do without care)..

The next two methods are designed for component programmers, as they encapsulate a higher-level of functionality. In particular, they both involve the whole of the start-up lifecycle (`sfDeployWith`, `sfDeploy` and `sfStart`), and have standard error handling behaviour, and the handling of the synchronization and multiple lifecycle aspects.

The first of the two methods, `sfCreateNewChild`, is used to create a new application part as a child of an existing compound. The way to achieve this is by invoking the method on that compound with (in addition to the description):

- The name by which this component is to be known within that parent compound – again may be null if a parent is not required
- A set of additional attributes that should be placed into the top-level component description before deployment.

The error handling during the lifecycle is defined to be to detach and terminate the new child, if necessary, and to throw an appropriate exception in the calling thread.

The second of these methods, `sfCreateNewApp`, is used to create a new application which is its own root – it has not parent. Consequently, no parent is required, and no name for it to be associated with within a parent. Consequently, in addition to the description, the only additional data required is

- A set of additional attributes that should be placed into the top-level component description before deployment.

The error handling during the lifecycle is defined to be to terminate the application (there is no need to detach – it is a new root), and to throw an appropriate exception in the calling thread.

CompoundImpl

The class `CompoundImpl` in package `org.smartfrog.sfcore.compound` is the implementation of the core capabilities of SmartFrog collections. In addition to being the class which is extended whenever a new type of Compound is defined, it provides all the semantics for one of the most common type of grouping – the shared-lifecycle component collection.

The purpose of the `CompoundImpl` is to provide a collection with the following semantics:

- The `Compound` provides a notion of child component – one for whose lifecycle it is responsible. A child is an attribute of the `Compound`, but also has a specific relationship relative to lifecycle management and failure monitoring.

- The *`Compound`* and its sub-components share an identical lifecycle, i.e.

    - whenever the compound is created, so are the sub-components in order of definition

    - whenever the compound is initialized, so are the sub-components in order of definition

    - whenever the compound is started, so are the sub-components in order of definition

    - whenever the compound fails or is terminated, so are the sub-components; this may be done asynchronously or synchronously in order of definition

1. This phasing of the lifecycle is of extreme importance to the SmartFrog system. The phasing carries through the hierarchy of the compounds and effectively provides a top-down, depth-first traversal of the compound parent-child containment tree. Thus, each child is guaranteed that each of the other children (and indeed the entire tree) has been stepped through the previous phase of the lifecycle before it will step through the next. Thus, all components will be initialized before any are started, and so on. The only exception to this is termination, which may occur at any time and may be completely asynchronous.

    - The `Compound` monitors its children for their status using the child's `sfPing` template method. If there is any failure of this method (either an exception or the method returning false), the child is assumed to have died and the compound propagates this as defined by the shared lifecycle principle. Note that other semantics for failure recovery, or other semantics for lifecycles, may be implemented and many are provided within the accompanying workflow package

35 Component Template

The following text is the typical outline for a primtive component in SmartFrog

```
import org.smartfrog.sfcore.prim.Prim;
import org.smartfrog.sfcore.prim.PrimImpl;
import org.smartfrog.sfcore.prim.common.SmartFrogException;

import java.rmi.RemoteException;

public class MyPrim extends PrimImpl implements Prim {

    /* any component specific declarations */

    public MyPrim() throws RemoteException {
    }

    public synchronized void sfDeploy()
            throws RemoteException, SmartFrogException {
        super.sfDeploy();
        /* any component specific initialization code */
    }

    public synchronized void sfStart()
         throws RemoteException, SmartFrogException {
        super.sfStart();
        /* any component specific start-up code */
    }

    public synchronized void sfTerminateWith(TerminationRecored tr)  {
        /* any component specific termination code */
        super.sfTerminateWith(tr);
    }

    /* any component-specific helper or public methods go below here */

}
```

The template to write a basic modified compound is similar, but using `Compound` and `CompoundImpl` in place of `Prim` and `PrimImpl`.

36 Well-Known Attributes

SmartFrog defines a number of predefined component templates, such as the ones for `ProcessCompound`, which require specific attributes to be defined. These attributes are captured in the interface:

```
org.smartfrog.sfcore.common.SmartFrogCoreKeys
```

The special attributes used in the framework are:

- `sfProcessHost`: Attribute used to determine the host to use to locate the root process compound on that host.
- `sfProcess`: Attribute used to determine the process/subprocess name where a component runs.
- `sfProcessName`: Attribute used to name a process/subprocess
- `sfProcessComponentName`: Attribute used to name a component
- `sfRootLocatorPort`: Registry port used by the `rootProcess` daemon
- `sfSubprocessGCTimeout`: Attribute with garbage collection time out for subprocesses
- `sfHost`: attribute used to determine the host address where a component runs
- `sfProcessAllow`: Attribute used to define if subprocesses can be used
- `sfProcessTimeout`: Attribute with subprocess deployment timeout
- `sfProcessJava`: Attribute that holds the process java start command
- `sfProcessClass`: Attribute that holds the class name for subprocesses
- `sfDeployerClass`: Attribute that holds the class name for deployer
- `sfSyncTerminate`: Attribute that determines asynchronous or synchronous termination of compound
- `sfClass`: Attribute that holds the class that implements a component
- `sfConfig`: Attribute that determines the resolution root of a SmartFrog description
- `sfSchemaDescription`: Attribute that determines the definition of a schema
- `sfCodeBase`: Attribute that defines the codebase for a component
- `sfLivenessDelay`: Attribute that defines how often to send liveness in seconds.
- `sfLivenessFactor`: Attribute that defines how many multiples of the liveness delay to wait till a liveness failure of the parent is declared
- `sfExport`: Attribute that defines if a component has to accept remote method calls
- `sfRootLocatorClass`: Attribute that defines the root locator class

- `sfBootDate`: Attribute that hold the boot time of the root process daemon
  -
  - Some special names used in the framework:
- `rootProcess`: Name used to name root process
- `ROOT`: Name used to refer to the root reference in a particular hierarchy of components or description
- `sfRunProcess`: Name used to name a root process deployed without registry
- `unnamed_`: Prefix use to name unnamed deployments
- `sfDeployFailure`: Name used to name a deploy phase failure
- `sfStartFailure`: Name used to name a start phase failure

Part1: The SmartFrog Runtime

37 Deployment In Detail

In SmartFrog, the term deployment is used to indicate the creation of the components in response to the system being given a suitable parsed and resolved description. A parsed component description may be deployed in two primary ways:

- by invoking the `sfDeployComponentDescription` method on a compound, and passing the parsed component to it – this would on the whole be restricted to those cases when the compound is to be the parent, or when it is simply desired to have the initial deployment occur near the compound (i.e. in the same JVM).

- by invoking the same method on the `ProcessCompound` of the SmartFrog JVM in which the top-level component should be created, though any use of deployment control attributes, such as `sfProcessHost`, may still cause the component to be deployed elsewhere.

The SmartFrog system uses the second of these two mechanisms for deploying the descriptions that are given to it on the command-line.

In either case, the deployment semantics are identical and proceeds as follows:

- The compound passes the top description to the deployer.

- The deployer, if it is the right one, examines the description for the attribute `sfClass`, extracts the class-name and creates an instance of the component class using the default constructor. If it is not the correct deployer, it locates and passes on the request – see section for a description of how this is done.

- The instance is initialized using the `sfDeployWith` method, being passed its component description as a parameter.

  - If the component is a prim, it carries out some basic initialization and returns.

  - If the component is a compound, it examines the attributes of the description and extracts those that are non-lazy component descriptions. It passes these descriptions to the deployer to create an instance of its subcomponents.

- The object reference (possibly remote) is returned as the result of the `sfDeployComponentDescription` method call. The receiving object is then responsible for invoking the remaining lifecycle methods in the correct order.

A number of facts should be noted. Firstly, that any component tagged `LAZY` is not deployed by its parent compound – this leaves the description to be used as structured data or for later programmatic deployment. Secondly, the semantics of what is considered a component in a description is not defined by the framework; rather it is defined by the compound component. This alternation of responsibility between the deployer to create a single instance of a component and the component to decide the next steps is an important feature of the deployment mechanisms within the SmartFrog framework. The last fact is that the follow-up initialize/start lifecycle is entirely up to the components that control the deployment – they are responsible for invoking these methods. It is true that the framework will invoke a specific lifecycle on the initial descriptions that are passed on the command-line – however, any other component that initiates a deployment may chose to impose a different lifecycle if so desired.

Selecting Deployers

A deployer is a Java object that is capable of creating an instance of a component in a specific place, as described by attributes in the component description. SmartFrog comes with three deployers, each providing a little more capability than the previous. Some of the services supplied with SmartFrog may add additional deployers. There are two aspects to cover

- how a deployer is chosen
- how each deployer interprets the attributes to select a location for deployment.

The first of these is either the class defined by the `sfDeployerClass` attribute of the component description to be deployed or the default deployer if this is not provided. The default is an instance of the `PrimProcessDeployerImpl` class. Thus in the example

```
sfConfig extends Compound {
    sfDeployerClass "org.smartfrog.example.NewDeployer";
    …
}
```

The component will be deployed with the `NewDeployer` class. If this attribute had not been present, the default would be used. Each component in a hierarchy may be deployed with a different deployer.

The three deployers provided by the core SmartFrog system are as follows:

- `PrimDeployerImpl` – deploys the component in the current JVM, ignoring any other attributes
- `PrimHostDeployerImpl` - deploys the component in the root process compound of the host whose name is provided in by the `sfProcessHost` attribute. If no name is provided, the current JVM is used.
- `PrimProcessDeployerImpl` – deploys the component in the named process (or the root process if not provided) of the specified host. If neither is provided, the deployer uses the current JVM. The process name is defined using the `sfProcessName` attribute, the host using the `sfProcessHost` attribute. This is the default deployer.

Thus, the following description will be deployed on the appropriate host and process, given that the default deployer will be selected.

```
sfConfig extends Compound {
    sfProcessHost "sfhost.smartfrog.org";
    sfProcessName "example";
    …
}
```

A description of processes, root processes and named sub-processes, is given in section 42.

Termination

Termination may be initiated on a component in one of three ways:

- By a call of the API method `sfTerminate(…)` by a thread started by the component, or from an external source such as a management system.
- From the parent of the component.
- From a child of the component, indicating that the child has terminated – although depending on the semantics of the compound, this may trigger an action other than the propagation of termination. The standard `CompoundImpl`, though, will propagate this termination to itself, the other children and its parent.

Whereas for most of the component lifecycle stages, the compounds and primitives are stepped synchronously through the cycle – each is created before any are initialized, and each is initialized before any is started – this is not necessarily a good model for termination. In many cases, termination may be carried out asynchronously. Indeed, in the case of failure of a component or the network, this is an absolute necessity. In other cases, it may be desirable to terminate in a specified order whenever possible.

SmartFrog allows a degree of control over the termination process. The default model is to terminate asynchronously, notifying parent component and child components in arbitrary order, and calling the termination template method on the component – the *`sfTerminateWith(…)`* method – at some point in that process.

If a component must be terminated synchronously, the attribute *`sfSyncTerminate`* must be set to the boolean value *`true`*. If not set, or set to *false*, the component will terminate asynchronously.

37.1 Synchronous Termination

Synchronous termination causes a component to carry out the following steps in order:

- Inform the children to terminate, waiting for each to be complete until the next is invoked. This is done in the order of definition in the component description.
- Terminate self by calling the `sfTerminateWith` template method.
- Notify parent that termination has occurred, if not initially told by the parent to terminate.

Note that a child component need not terminate itself synchronously, this is set by the attributes within the child. If it is desired that all components terminate synchronously, the definition of `Prim` should be modified in the file `components.sf` to include the appropriate attribute.

37.1 Asynchronous Termination

This is the default mode of termination. The same three steps are undertaken, though the children are all informed of the need to terminate in separate threads that are created for this purpose. The implications of this are that there may be some delay in the termination process, that resources may not be freed immediately. However, it also means that long termination sequences are scheduled independently of notifying its termination to its parent. This fast notification may be important for fault tolerance purposes, for example.

37.1 Terminator Thread

As mentioned in section 9.2.2, asynchronous termination is done using a separate thread. Terminator thread provides a standard way of asynchronously terminating `SmartFrog` components. This class is defined in `org.smartfrog.sfcore.common` package. For example:

```
// create an instance of terminator thread by passing component
reference and termination record
TerminatorThread terminator = new TerminatorThread(compRef,
                                            terminationRec);
// Start the thread
terminator.start();

// Terminator thread object can also be constucted using another
constuctor
public TerminatorThread(Prim target, Throwable t, Reference compId);
```

Some utility methods are available in the `TerminatorThread` class:

```
// returns a TerminatorThread object which does not notify
// components parent while terminating the component
public TerminatorThread quietly();

// returns a TerminatorThread object which
// detaches the component from parent before terminating
// the component
public TerminatorThread detach();

// returns a TerminatorThread object which
//  does not terminate the underlying component
public TerminatorThread dontTerminate();
```

38 Attributes, LAZY Links and RMI Object References

Accessing Attributes At Runtime

Attributes of a component are stored in a `Context`, as defined by the `ComponentDescription` used to create the component. These attributes may be accessed using the `sfResolve` method of the `Prim` interface. This method takes a reference and, relative to the component on which it the method is initially invoked, de-references it to obtain the desired attribute. This method may equally be used by any component, one that has an RMI reference to the object or by the component itself. If called within the body of component, the reference is de-referenced starting with the component itself.

The `sfResolve` method is defined as follows.

```
Object sfResolve(Reference r)
        Locate an attribute when given a reference. This method resolves
        from the point in the component hierarchy that is the Prim on
        which it is called. References may be constructed using the
        class constructor (recommended) or the fromString("…") method
        defined on Reference to provide a simple, though inefficient,
        way of creating complex references.
```

This method will return an *Object* that may be inspected using reflection or caste to an appropriate class. For most attributes it is clear from the context what the value returned should be.

Although the above method is the primary one for accessing attributes, there are a number of other methods with a number of semantics – they are all documented in the Javadoc.

LAZY links And RMI

If the description contained a `LAZY` link, the link resolution phase would not have resolved it in advance – rather it would have left the reference to be the value of the attribute. There are now two possibilities when `sfResolve` is invoked:

- That the reference is de-referenced at this point, returning the appropriate value.
- That the reference itself is returned as the attribute value.

The first of these semantics is that implemented by `sfResolve` (though not by all methods in the `Prim` interface). Consequently, if an attribute is a `LAZY` link, it is silently de-referenced at the time it is accessed, following the parent-child component hierarchy in exactly the same way as is done with `ComponentDescription` declarations during link resolution.

Note that the value returned by the resolution is not cached locally. It will be fetched each time the attribute is accessed. The value should be cached by the component itself if this is required.

There are several reasons why a `LAZY` link might be used instead of a normal link, in spite of the fact that delaying access in this way is much less efficient – particularly so when the attribute referenced may be remote. These reasons are all related to the property that the value of the attribute is not available at time of link resolution.

- The attribute referenced may be set at run-time by the component, or may continually change its value and must be read many times. This variability is the reason for not automatically caching values locally.

- The link may reference a component with the intention of obtaining the RMI object reference of the component

Consider the following example:

```
sfConfig extends {
    comp1 extends Prim { … }
    comp2 extends Prim { …
        otherComp LAZY comp1;
    }
}
```

The link to `comp1` from within `comp2` is defined as `LAZY`. This is because the link, if it de-referenced at the time of link resolution, results in a copy of the `ComponentDescription` that is `comp1` being placed within that of `comp2` under the name `otherComp`. Thus, if it were not `LAZY`, after resolution the result would be the resolved equivalent of

```
sfConfig extends {
    comp1 extends Prim { … }
    comp2 extends Prim { …
        otherComp extends Prim { … };
    }
}
```

which is not the intended behaviour. If the link is made lazy, on the other hand, the resolution is carried out at run-time by the `Compound`/`Prim` hierarchy and instead of the `ComponentDescription` being copied, the RMI object reference of the component referenced is returned. Under these circumstances, `comp2` would be able to call the exported methods of `comp1`.

The Moving ROOT

The root reference part appears simple, but it can have some unexpected side effects. The meaning of the `ROOT` reference part is that, at the time of resolution, the `ComponentDescription` or `Compound`/`Prim` hierarchy is traversed upwards from the point of initial resolution until there is no parent. This top `ComponentDescription` or `Component` is the root.

Therefore, when a link is defined using the ROOT reference part, the ROOT refers to the file itself, the virtual `ComponentDescription` containing all the attributes at the top level.

At run-time, however, this `ComponentDescription` no longer exists – the component called `sfConfig` from that file is normally the new run-time root. Consequently, `LAZY` links with `ROOT` resolve differently from the way they would have done if they had not been lazy and had been resolved at the time of link resolution.

The situation is even more confused, though, since the structure of the tree can change during the life of an application. For example, the use of `sfDetach` changes the structure of the tree during its execution. This difficulty in pinning the meaning of `ROOT` can cause great confusion and therefore the use of `ROOT` should be limited to occasions where the semantics are clear.

Modifying Attributes Values

Given the use of `LAZY` links, with the ability to read attribute values many times, they may be used as a way of passing information between components. To provide the ability to add, remove and replace attribute values from the context, three methods are provided as part of the `Prim` interface. Note that these descriptions of these methods do not accurately define the signature. These should be found from the Javadoc.

```
public Object sfAddAttribute(Object name, Object value)
      Add an attribute to the component's context. Values should be
      marshallable types if they are to be referenced remotely at run-
```

time. If an attribute with this name already exists it is not replaced.

```
public Object sfRemoveAttribute(Object name)
```

Remove named attribute from component context. Non present attribute names are ignored.

```
public Object sfReplaceAttribute(Object name, Object value)
```

Replace named attribute in component context. If attribute is not present it is added to the context.

Trapping Accesses And Reference Adaptors

Attributes of a component are stored in a `Context`, as defined by the `ComponentDescription`, and made available through the `Prim` interface using the `sfResolve` method. However, it can be useful for a component to provide an attribute that is not stored within its context but is evaluated on demand, for example an attribute representing the current load on the component.

For this, the request for an attribute value must be trapped, diverted from the lookup in the context, and the correct value returned. This is possible by understanding how a reference is de-referenced.

The `sfResolve` method is almost immediately converted into a request to resolve the reference from a specific index of the reference parts using the method

```
Object sfResolve(Reference r, int index)
```

This method resolves one part to get to another ComponentDescription, and forwards the request adding one to the index. In this way the reference is eventually fully de-referenced and the attribute value returned.

This method is implemented by `PrimImpl`, and its default implementation is to look up in the context and forward if the reference is not yet completely de-referenced.

The accepted way of trapping a request is to override the default implementation in the appropriate component class, checking to see if the attribute is one that needs special handling and if not, invoking the super-class method to continue as normal. If it has to be handled specially, carry out the appropriate processing and return the evaluated attribute value.

This mechanism has some interesting possibilities. For example, it is possible to intercept a reference early on in its de-referencing process, within a component that may then use the rest of the reference parts to be a parameter to the component to evaluate its attribute value. This may be used, for example, to provide an adaptor into other naming and discovery technologies.

Consider a component that uses a database to extract a value for an attribute given a table name, a key and the column name required. Assume further that this component has been deployed on a specific host, say `db.smartfrog.org`, with the `sfProcessComponentName` set to `dbAccess`. Descriptions may now use references of the form

```
foo LAZY HOST db.smartfrog.org:dbAccess:ttt:kkk:ccc
```

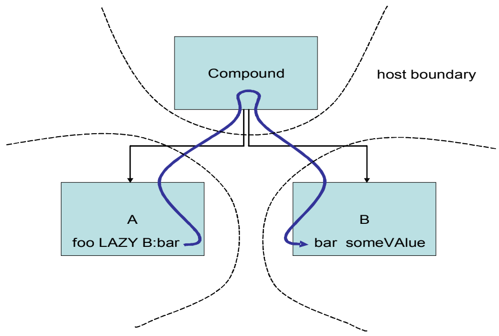
where *`ttt`*, *`kkk`*, *`ccc`* are the table name, key and column respectively. This is again an example of how a component which is accessing the attribute *foo* need know little of how the attribute's value is obtained. This is an issue for the configuration not for the component code.

sfHost and sfProcess Attributes

Every component, when created, adds two attributes to its collection of attributes. These are the hostname and process name in which the component is running. These are defined in the very earliest phase of the lifecycle, during construction. Consequently they are present during all other phases.

39 Attribute Serialization

At run time, when components are scattered across a number of hosts, components exchange the values of attributes through the use of link resolution. As these values are passed around, they traverse a number of hosts. For example, consider the diagram below:



In this diagram, component A requests the attribute "foo" from component B whenever the attribute "bar" is accessed. This is passed back in response to the request to resolve the bar attribute. The attribute has to be of a class that is known to both hosts containing the components A and B, otherwise some form of exception will occur. However, what is not so obvious is that the attribute value is also passed through the compound component containing A and B.

It should be noted that although in the language a very limited number of classes can be used for attribute values, and these are known everywhere, at run-time attributes can have other values than these simple ones, indeed any serializable class, and in particular this will include all the RMI stub classes for the various components.

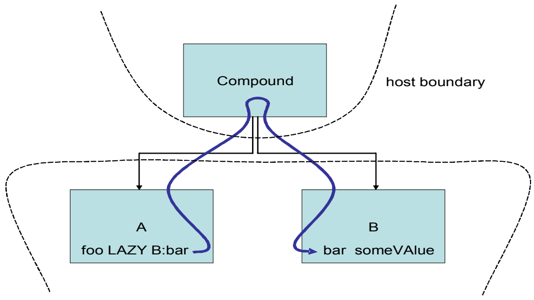
Early versions of SmartFrog simply insisted that the classes of attributes that were passed through intermediate nodes in response to a resolution request had to be known to those intermediate nodes. This was not an adequate solution, so an alternative model had to be found.

An attempt at solving this automatically wrapped all attribute values within a well-known wrapper class (`SFMarshalledObject`) for passing around. This wrapper held the actual value as a byte array containing the serialized value. In this way only the wrapper class needed to be known at every host and the serialization and deserialization simply occurred transparently at each end of a link resolution (in this case at the hosts containing A and B). At the intermediate node, only the wrapper would be handled and the real attribute value would be held as bytes within it.

This solved many of the problems associated with the need for attribute classes to be known everywhere, but it introduced others. Consider if two components are always together on the same host. They locate each other using a link in the normal way, and use local interfaces to communicate (this is often done for security reasons, not to expose to remote calls sensitive interfaces). As the objects are serialized and deserialized within any link resolution, the result is always a remote object reference even though the object is local. This cannot be caste to a local-only interface, so it became impossible to ever call a local interface on a component located through link resolution. As it is impossible to extract the real object from an RMI Object Reference, so there is no easy way out of this problem.

The final solution, and that in the current version, is for the wrapper to be a little more intelligent about the serialization of the value it wraps. It only does so if it is being serialized itself – so if the wrapper is never passed between hosts, the value it wraps is not serialized either.

This fixes most of the problems associated with local interfaces, as well as solving the issue of classes having to be known everywhere. However it is not perfect. If a value is ever passed remotely, even if it ends up on the original host, it will have become serialized. A classic example of this is shown in the following diagram.



Although A and B are on the same host, the compound is not and the attribute referenced will therefore pass through the remote host. So if A and B need to communicate using local interfaces, this is not a suitable description. An additional container compound will need to be added on the same host as A and B to act as the local container and so ensure that the link resolution never traverses a host boundary. This is also more efficient for deployment and for liveness, so is anyway to be encouraged.

Note that in the discussion above, for host one should really read process. Crossing process boundaries, even on the same host, introduces the same problems.

40 Liveness

A feature of SmartFrog is its all-or-nothing semantics for Compounds, its shared lifecycle for all components. This must be done in a context where any host within the network or, indeed, the network itself, may fail at any time. These failures may result in applications losing some components, or perhaps being partitioned into two or more parts if the network itself fails.

The guarantees that SmartFrog attempts to provide, such as all-or-nothing deployment, cannot be achieved by purely passive means. The system needs to monitor the various components to ensure that they are still active, can be accessed and are in good shape. Detection of the failure of an application component, or of the network between such components, must be notified to the application in an appropriate way. Note that the difference between failure of a node and the failure of the network between nodes cannot normally be diagnosed from the perspective of another node.

The hierarchical nature of an application provides a natural chain of responsibility – a parent is responsible for the checking and monitoring of its children, and vice-versa. Notification of failures will be made to these components and it is the responsibility of these components to take appropriate action.

SmartFrog provides a default liveness checking mechanism based on a parent regularly heart-beating its children. Each component provides a method `sfPing` as part of the `Prim` interface. This method either returns or throws an exception, and it may be overwritten by the component implementer to carry out any appropriate checks.

There are two possibilities when this method is called:

- The method succeeds, and returns normally – this is considered as confirmation that the component is alive.

- The method fails by throwing an exception, implying one of the following:

    - that the remote host has failed.

    - that the network has failed and the two components are out of communication.

    - that the component is indicating that an error has occurred and it should be considered as dead.

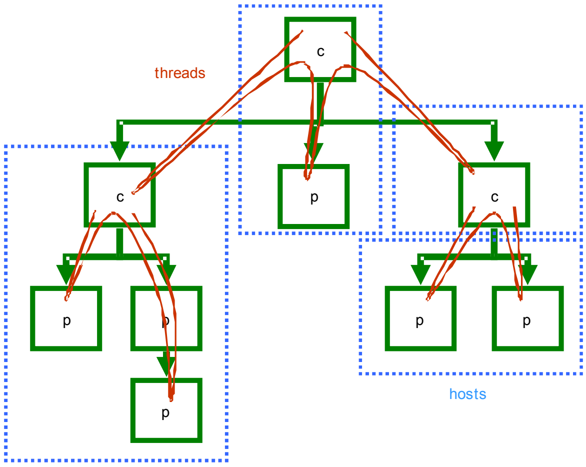
  The default `Compound` semantics is to terminate itself under any of these circumstances and hence the whole application follows suit – again alternative semantics may be defined.

On the child side, it monitors how frequently the parent is checking its state. If the parent misses more than a specified number of heart-beats, the child assumes that the parent has failed, or there is a network failure, and handles the failure – again the default is to terminate itself (and hence any part of the application below it in the hierarchy).

Should the parent or network recover, the child will respond with an exception to any request for status, and the parent and hence the rest of the application will terminate by default.

If the default behaviour needs to be changed, the report to the component is done via a call to the method `sfLivenessFailure`. This method, in the default implementation provided by `PrimImpl`, simply terminates. This method may be over-ridden if required.

To try to be a little more efficient, the SmartFrog system does not create a liveness thread for each component. Rather, it creates one for every component that is in a different process. This thread is responsible for checking all the components in the hierarchy downwards until it has checked a single level of remote components. Equally, it is only the “first” component that monitors whether it is being checked on a regular basis. Consequently, overheads for large numbers of components on one host are minimal. The following diagram makes this clear.



The frequency of liveness checking can be modified to balance responsiveness requirements and overheads in any particular system. Two attributes may be set:

- `sfLivenessDelay`: (default 15 seconds) how frequently in seconds should the liveness be checked. Zero indicates no checking.

- `sfLivenessFactor`: (default 2 misses) how many missed heart-beats should be considered a parent failure.

These may be modified for the entire system by setting them in `prim.sf`, or for all components rooted in a process by setting the attribute definitions in the process compound for that process, or it can be set for the sub-tree of a component by setting it in that component.

Each component searches for the current value of these attributes up the containment tree up to the root (using an `ATTRIB` reference part). If it does not find them anywhere up the tree, including in the root component, the default is taken from the local process compound (this defines the defaults of 15 seconds and 2 misses given above).

41 Hooks

At times, it is necessary to carry out an action for some or all of the phases of the lifecycle of every object – a typical use being to trace or log the components that come and go. To enable this, a set of lifecycle hooks have been provided, such that at every component lifecycle phase the appropriate hook is called, parameterized by the component.

The hooks are called in the `PrimImpl` base class, as part of the default implementation of the template lifecycle methods. For the hooks to be called, therefore, it is essential that the superclass template methods are called in any derived class, as indicated in the primitive template in Appendix E.

To create a new hook class, the class should implement the interface `PrimHook` in package `org.smartfrog.sfcore.prim`. This interface defines a single method, `sfHookAction`, which is called when the hook is invoked. This method is documented in the Javadoc.

To register a hook, an instance of the hook class should be added to one of the four `HookSet` members available in `Prim` (using the `addHook` method), namely `sfDeployWithHooks`, `sfDeployHooks`, `sfStartHooks`, and `sfTerminateWithHooks`; one for each of the template methods. Note that the effect of the hook is local to a specific process.

Removal of hooks is also possible, so creating a component that adds a hook on start, and removes it on termination, is feasible – so for example a management component could, on demand, capture all the lifecycles for a period of time at a process and send the information back to some central console.

The mechanism is primarily intended for debugging and management.

42 Processes and Java Virtual Machines

The SmartFrog system is designed to form the basis for a fully distributed configuration and programming environment. As such, the system must be able to deal with deploying components into many processes (Java Virtual Machines) on many different hosts. This section covers the subject of these processes, how they may be identified and located, and how they may be started and managed. It also deals to a degree with the APIs provided to allow applications to interact with them, however, as usual, the full API is described in the accompanying Javadoc.

There are two concepts to understand as part of the underlying control of the SmartFrog system. The first of these is the SmartFrog Resource Reference (`SFREF`). This is a URL to a description file to deploy or otherwise use. The second is an SmartFrog Action Descriptor (`SFACT`), which is used to indicate to SmartFrog an action to take. These are now described in more detail.

SmartFrog Resource References

Throughout the SmartFrog system, including on the command line, references to SmartFrog resources (i.e. files) may be given in a number of ways:

- as a URL to the file;
- as a relative or absolute path name to a file;
- as a path to a resource in a jar file on the classpath or code base.

In this last case, the reference should be given as a path relative to the root of the package structure within the jar file, i.e. without the leading "`/`". In most cases this leading "`/`" is removed by the code, but there may be some instances where this is not so.

In the following descriptions of the scripts, a reference to such a resource is referred to as an `SFREF`.

SmartFrog Action Descriptor

An action descriptor is used on the command line to describe a certain type of action that will be carried on by the daemon. An action has a number of "`:`" separated fields, most of which may be left blank in many cases.

The format is:

```
NAME:ACTION:SFREF:SUBREF:HOST:PROCESS
```

The semantics of the fields are defined as follows:

42.1 NAME

The name is a single word, or a SmartFrog reference in which case it must be surrounded by quotes. The name has one of two interpretations depending on the action to be taken (see next field).

In `TERMINATE`, `DETACH`, `DETaTERM`, `PING`, `DIAGNOSTICS` the name is a reference to the component on which to apply the action.

In `DEPLOY`, the name is treated like a placement and the name is split into two: all but the last part is a reference to another component and the last (or only) part is the name which will be given to the deployed component within that referenced component. If the component is not a `ProcessCompound`, the component is also made the parent of the deployment.

In all cases, the `NAME` is resolved relative to the process compound of the `HOST` and `PROCESS` specified by the appropriate fields.

When a name is not provided, it indicates the process compound of the host and process defined in the `HOST` and `PROCESS` fields. Also, in `DEPLOY` no name means use the `sfProcessComponentName` from the description if available or generate a random name to name the deployed description.

Examples:

```
foo
"HOST localhost:foo"
```

42.1 ACTION

This field defines the action to be taken on the named component

- `DEPLOY` a component or application.
- `TERMINATE` a component or application.
- `DETACH` a component from its parent.
- `DETaTERM` detach and terminate a component from its parent.
- `PING` a component.
- `PARSE` a description and generates report.
- `DIAGNOSTICS` a component and generate report.

42.1 SFREF

The SmartFrog description (if needed) to be used by ACTION. It is a SmartFrog Resource Reference see 3.1. It needs to use quotes (" or ') when the reference is using "`:`". Currently this is only required for a `DEPLOY` and `PARSE` actions and is ignored otherwise.

Examples:

```
/home/sf/foo.sf
"c:\sf\foo.sf"
'c:\sf\foo.sf'
```

42.1 SUBREF

When the SFREF is parsed and resolved, the result is a component description containing a number of attributes. In the "sf" language, this is the contents of the `sfConfig` definition. Under normal circumstances, it is this whole definition that is used for the deployment, but occasionally, for testing purposes perhaps, it is useful to specify some single subcomponent. Under these circumstances, the name of this attribute, or a reference to a deeply nested application, may be provided. This is the SUBREF.

Examples:

```
foo
"first:foo"
'first:foo'
```

42.1 HOST

host name or IP from where to resolve the name. If `HOST` is not present, the process name is ignored and the process executing is used. If you want to refer to another process, other than the executing one, on the local host, "localhost" should be used and the appropriate PROCESS name used.

Examples:

```
foo.hpl.hp.com

127.0.0.1
```

42.1 PROCESS

process name from where to resolve the name. When empty it defaults to “rootProcess”.

42.1 Examples

These examples show the use of the action descriptors for different purposes.

Example 1: Deploy a description in the local daemon

```
Ex1:DEPLOY:org/smartfrog/examples/counter/example.sf::localhost:
```

Example 2. Terminate the local sfDaemon

```
rootProcess:TERMINATE:::localhost:
```

or

```
:TERMINATE:::localhost:
```

Example 3: Deploy "counterToSucceed" from counter/example2.sf

```
counterEx3:DEPLOY:org/smartfrog/examples/counter/example2.sf:"testLevel
1:counterToSucceed":localhost:
```

Example 4: Get diagnostics report for "sfDefault" component running in localhost

```
sfDefault:DIAGNOSTICS:::localhost:
```

SFSystem And Command-Line Parameters

The main loop of a SmartFrog process is provided by the class *SFSystem*. When this class is invoked, it reads various command line parameters and system properties to generate the appropriate type of SmartFrog process and to trigger the desired configuration actions. The general form of the command line for SmartFrog is

```
java [-D properties] org.smartfrog.sfcore.SFSystem [parameters]
```

In general the command line will be triggered by a script setting up the properties as defined and described below, and arranging for the command line parameters to be correctly defined. The parameters are:

- -a SFACT: SmartFrog Action Descriptors (SFACT). There can be more than one of this.
- -p port:port where to locate/start the daemon
- -d : environment diagnostics report.
- -f SFREF: file that contains a set of file that contains a set of SmartFrog Action Descriptors (SFACT). There can be more than one of this.

- -e : exit after deployment of the configurations is complete.

- -headless : run SmartFrog in headless mode (equivalent to running the JVM with -Djava.awt.headless=true). All GUI components will be disabled in the started process.

- -? : usage and help information.

The system properties are rather more complex in their effect. To understand their effect on SmartFrog behaviour, the concept of a process compound must be explained. Of course, additional properties may be defined to parameterize specific applications that require it (this is particularly useful in conjunction with the `PROPERTY` and `IPROPERTY` links).

Process Compounds

Every process contains a specialized component known as a `ProcessCompound` created by `SFSystem` at start-up. It is a modified `Compound` providing the full component creation interface offered by all compounds. As such, it is the means by which all top-level components within the process are created. As it is a compound, it also contains attributes and sub-components:

- It contains attributes that affect the behaviour of the process, such as security settings, classpath information, and so on.
- It contains an attribute referencing each application that is running in the process. These applications are all children of the `ProcessCompound`, in the sense that the `ProcessCompound` does monitor the children for status. This is so they may be removed from the `ProcessCompound` on termination. The applications, however, do not consider the `ProcessCompound` as their parent.
- It contains, as a child, the `ProcessCompound` of any sub-process that may be created during the deployment process: it is possible to have a simple two-level tree of processes controlled by attributes within the `ComponentDescription` to be deployed. A sub-process does consider the root process as its parent and will terminate if the root dies.

However, to interact with the `ProcessCompound` in a process, for example to initiate a deployment, it is necessary to obtain the RMI object reference to that `ProcessCompound`. SmartFrog defines a core mechanism using the RMI registry for this purpose, though other mechanisms may additionally be defined. One such is that provided with the SLP discovery infrastructure described in a separate document. Only the registry-based mechanism is described here.

Types Of Processes

Processes come in three main types:

- Root Processes that create an instance of an RMI registry and register themselves within it so that they may be located. Note that a host may have several root processes so long as they use different ports for the registry. However there are restrictions in the current release that limit the degree of interactions of processes on different ports and hence this should only be done if the two belong to different SmartFrog systems. A SmartFrog system is designed so that a single root process exists on each host, and that these are on the same port number.
- Sub-processes are processes that register with the root process and become children of the root process, and hence are named within the `ProcessCompound` of the root process. This is the mechanism by which the sub-processes are located and their RMI references obtained. Subprocesses may be created in two ways – by explicit launching from a command line or by defining that an application should be deployed in a sub-process that is created if it does not already exist.

- Basic processes are processes that do not use the core SmartFrog mechanisms to advertise their existence – either because they are not required to be accessed or because they use other mechanisms for this purpose.

The normal model for a SmartFrog system is that a root process compound is created as a service or daemon on each host, possibly at boot time, and that applications are created (in dynamically created sub-processes if so desired) as defined by applications that are launched from basic processes that exit when the application has been successfully deployed.

Process Attributes

Each process, when it is started, customizes its behaviour dependent on attributes that are given it in two distinct ways:

- As attributes defined in the `processcompound.sf` file that is read at the start-up of any process.

- System properties set on the command line that override the defaults provided in the above start-up file.

These attributes control aspects of process behaviour such as type of process it should be, what port is used for the registry within the system, whether security should be enabled and where remote code is available for remote downloading. Most of the attributes may be defined in either or both the above ways, however a few of the attributes must be passed as system properties. In particular, the security properties need to be set before the `processcompound.sf` file is read.

When the `processcompound.sf` file is parsed, all system properties with the prefix `org.smartfrog.sfcore.processCompound` are added to the context before deployment, thereby creating the process compound itself. Consequently, using the command line option

```
-Dorg.smartfrog.sfcore.processCompound.sfRootLocatorPort=2000
```

when starting Java overrides the default value defined in `processcompound.sf.` The attributes from `processcompound.sf` that are considered user-modifiable are as follows:

- `sfProcessName`
  (no default value preset in `processcompound.sf`)

- `sfLivenessDelay 15;`

  how frequently (in seconds) to check children processes and applications to status

- `sfLivenessFactor 5;`

  how many missed checks from parent process before assuming that parent process is dead

- `sfProcessAllow true;`

  allow sub-processes – only used with the root process compound

- `sfProcessTimeout 60;`

  how long to wait (in seconds) for a child process to be created before failure is assumed

- `sfRootLocatorPort 3800;`

  the registry port for the root process compound

Advanced attributes to control the classpath used to bootstrap the process compound are:

- `sfProcessReplaceClassPath false;`
  It true, it then replace process compound class path with the classpath provided by `sfProcessClassPath` attribute. If false, the classpath of the root process compound is added at the end of the one provided by the previously mentioned attribute. Note that replacing the classpath can prevent the process compound from starting if the SmartFrog core libraries are not part of the new classpath.

- `sfProcessClassPath`
  Classpath to be used when bootstrapping a process compound for the first time. Valid values for this attribute are:

  - String – string path. Needs to use the right OS platform separator (ex: ';' in Windows and ':' in Unix).

  - Vector – with a list of String paths. The right platform separator will be added automatically when creating the String classpath.

  - ComponentDescription with “Files” definition where some of the attributes are (see example bellow and documentation of component: .../services/filesystem/files/):

    - `dir` where to find the list of files that match a pattern.

    - `pattern` Optional pattern expressed using Java Regular Expressions notation.

    - `caseSensitive` Optional Boolean

    - `includeHiddenFiles` Optional Boolean

```
//-----------------------------------------------------------------
//Example about how to change the classpath of a ProcessCompound
//-----------------------------------------------------------------
#include "org/smartfrog/components.sf"
#include "/org/smartfrog/services/filesystem/files/components.sf"
// Files definition for creating a fileset.
classPath extends Files {
    pattern "\\w+*.jar";
    dir ( LAZY PROPERTY "user.dir" ++ "/lib" );
}
sfConfig extends Compound {
       sfProcessName "TestProcess";
       sfProcessClassPath extends LAZY classPath;
       sfProcessReplaceClassPath true;
}
```

Accessing Process Compounds And Attributes

Attributes may be accessed via the process compound itself. Indeed, the attributes should never be accessed by requesting the system properties - they may not have been defined through that mechanism. To access the attributes, first obtain the process compound of that process then simply lookup the attribute using the normal SmartFrog supplied methods (it is, after all, a Prim) as in:

```
SFProcess.getProcessCompound()
    .sfResolve(Reference.fromString("sfRootLocatorPort"));
```

Note that for efficiency, using the string reference parser should be limited. Overall, it is better to build and use a reference to avoid the expensive step of parsing the string as a reference.

If the process compound required is on a different host, or is in a different sub-process, the mechanism is to first locate the root process compound on that host, then to find the process of that name as an attribute within the process compound. This can be done using the following invocation

```
SFProcess.getRootLocator().getRootProcessCompound("hostname");
```

Or by making use of the host reference

```
 sfResolve(Reference.fromString("HOST hostname"));
```

Attributes may then be queried as before. Once the root process compound is obtained, sub-processes may be obtained by looking up the process name in the compound as with any other attribute.

Creating And Naming Sub-Processes

Sub-processes are created in two ways. A user may create a sub-process of an existing root process by setting the system property `sfProcessName` as follows on the command line to start the daemon:

```
-Dorg.smartfrog.sfcore.ProcessComopund.sfProcessName=name
```

By doing so, the process registers with the root under the supplied name. If the root does not exist, the process will fail and terminate. Note that if the reserved name “`rootProcess`” is used, a root process is created.

The second mechanism is via the attribute “`sfProcessName`” being given in a component to be deployed as in the following snippet

```
Foo extends Compound {
  sfProcessName "fooProcess";
  // ...
}
```

On deployment, the compound `Foo` will be deployed in a sub-process `fooProcess`, the host depending on the provision of the attribute `sfProcessHost` and local if not provided. If `fooProcess` already exists, this will be used, otherwise a new process will be created using that name.

It should be noted that in automatically creating this subprocess, the command line is defined to pass on all system properties of the root, apart from the `org.smartfrog.sfcore.processCompound.sfProcessName` property that set to the appropriate name for the process.

Naming Applications

As applications are deployed into the various processes, the process compounds are involved in the creation of the first component of an application tree in that process (this may not be the root component of the application tree, just the first in that process). As it does so, the process compound will keep an attribute referencing this component, the name of the component being either obtained from the component’s `sfProcessComponentName` attribute if it exists, or a random unique name generated on demand.

This component is not a child of the process compound, though the process compound does monitor it for status using the liveness mechanism, removing it if liveness shows that the component has terminated. However, the provision of this attribute does enable a management system to locate processes (using the registry and named sub-processes) and then to find all applications that are resident completely or partially within that process. By following the hierarchy of parent/child relationships, every component across all processes of every application can be located.

Using the –n command-line parameter of `SFSystem` command ensures that the root component of the associated configuration is given the name provided on the command line. It is exactly as though application were being given a name by using the `sfProcessComponentName` in the application itself.

HOST and PROCESS Links

Note that the naming of components in this way also provides a convenient naming service across a SmartFrog system that could be used by components to locate each other – for example, a component may wish to log events at the logging service on a specific host. The normal way in which components locate each other is, of course, by using a link. To support the use of the naming capability, a `HOST` link is provided which will, as it is resolved, use the root location mechanism to access remote process compounds, allowing the remainder of the link to be de-referenced in that context, to a subprocess compound or an application component. The syntax for `HOST` references is given in section 11.

Another link that may be used in a similar way is the `PROCESS` link that refers to the process compound of process in which the component is deployed.

43 The SmartFrog Security Model

Introduction

This section describes the SmartFrog security model. Its design deliberately had a very simple usage model in mind, so it may not suit all uses.

Several aspects are covered in this section. Firstly, the threats that are considered in the model are described. Secondly, the policies that must be enforced in order to protect the system from these threats are defined. Thirdly, the specific mechanisms that have been implemented, and the assumptions regarding the rest of the system are listed. Finally, a discussion about the current limitations that will motivate enhancements in future releases.

Threat Model

SmartFrog could be used to run malicious code on machines hosting a SmartFrog Daemon. It is critical that the system controls who can deploy code to machines.

In particular, the communication channels between SmartFrog daemons are not necessarily secure, -there could be malicious machines on the network. An attacker could modify deployment configurations sent from legitimate daemons, or just pretend that they are a valid participant and send their own. In addition, they can obtain critical configuration information, such as passwords, by snooping on the communication, information that can later be used to attack the system.

An important feature of SmartFrog is to dynamically load resources from web servers while deploying. These resources could be additional configuration descriptions, Java classes, scripts, executable files and so on. On many networks, remote sites (or the DNS entries used to locate them, and proxy servers used to mediate access) could be subverted, and malicious code downloaded.

Making a node "SmartFrog aware" implies installing a permanent service in this node, opening up a port for incoming requests, creating a special account to run the service, and so on. This can make that platform more vulnerable if someone can compromise the service itself, e.g., by exploiting a buffer overflow.

Security Policy

The ultimate goal is that a distributed application configured, deployed and managed by SmartFrog is not more vulnerable than the same application configured, deployed and managed manually using a local secure procedure. This means that SmartFrog is not trying to "fix" the security problems of the application itself by constraining what the application can do. However, the application could indirectly benefit by having a more flexible mechanism to bootstrap its security, or it could even use directly SmartFrog security services.

Another desired goal is not to introduce new vulnerabilities in the hosting node because SmartFrog has been activated. The problem is that the interactions of SmartFrog with the rest of the system make this goal platform dependent. Again, SmartFrog cannot "fix" problems in the underlying OS security; it is simply that SmartFrog must not make them more "visible".

To help clarify the target security model in this release, the concept of a SmartFrog Trusted Community (SFTC) has been introduced. A SFTC is a set of principals, typically composed of SmartFrog daemons and administrators that fully trust each other, i.e., they will do **anything** that another valid member requests, and they do not trust anybody else. There is a single authority that defines who is initially in the community, but a current member could later on add new members based on its own criteria. A member of an SFTC should only use resources that are trusted, where trust in this context means that they were created or authorized by another member of the community, and nobody has modified them since. In some cases, we want to enforce confidentiality as well as integrity on these resources, always within the scope of an SFTC.

As mentioned in the discussion on threats, SmartFrog uses web servers to dynamically load resources. Web servers are **not** part of the SFTC, although they host resources that are trusted by members of the SFTC. Mechanisms described in the next section will enforce the integrity (but currently not the confidentiality) of these resources.

There is a strong requirement for knowing what is coming from inside or outside the community, but it is less valuable to know from whom within the community the original request came from. This means that accountability of individual members in the community is very limited, something not surprising when members fully delegate each other actions, and there are no members with "special" privileges. Nevertheless, SmartFrog still give different identities to each principal in an SFTC to make future enhancements easier.

Special care should be taken before authorizing that a particular application might be deployed in the platform. Deployed components share the same environment and privileges of the SmartFrog infrastructure and they could take control of it. The problem is not only deploying a virus, but also deploying an application with an exploitable vulnerability, that will allow an attacker to get control of the application, then the SmartFrog local daemon, and finally all the nodes in the SFTC. In general, ensure that daemons have the minimal privileges required to carry out the tasks required by the applications.

Security Mechanisms

In this section, the security mechanisms that have been implemented for SmartFrog to support SFTCs are described. This implies knowing when a principal is a valid member of the community, ensuring that interactions over an insecure network of valid members are safe, and satisfying the integrity (and authentication) requirements of trusted resources, possibly hosted by non-members of the SFTC. In addition, other mechanisms that have not been implemented as part of SmartFrog, but are assumed to be available to make the model work, are discussed.

43.1 Built-in security mechanisms

SmartFrog uses PKI (Public Key Infrastructure) based on X509 certificates to provide principals of the SFTC with credentials that justify they are valid members of the community. The centralized CA (Certificate Authority) is based on openSSL and fully integrated with the rest of the release installation process (by using Ant). Keys for individual members are actually generated using Sun's `keytool`, and Sun's implementation of a keystore (with random passwords) is used to keep node credentials safe (we assume though that the centralized CA is in a safe, isolated environment).

Java 2 security mechanisms are leveraged to create a SFTC. In particular, the Java security policy is set so that, when enforced by a `SecurityManager`, gives full privileges to classes loaded from signed JAR files (using a trusted key for the SFTC), and none otherwise. These mechanisms are extended to other resources apart from classes, such as configuration descriptions, so that they are only loaded if they are in a signed jar (same signing key as before). This is particularly useful when the jars are hosted by web servers that are not part of the SFTC. Jars containing classes and configurations of this release are signed as part of the installation process. JAR filess with your classes/configurations could be easily signed with a similar process.

RMI calls are tunnelled over SSL using the JSSE API and Sun's reference implementation. SmartFrog forces **mutual** authentication in the SSL sessions based on the 1024 bits RSA public/private keys that are discussed above. Only the SFTC CA keys are part of our trust assumptions so, by validating the other partner's X509 certificate chain, SmartFrog knows that it belongs to the community. However, the authenticated credentials of the other session peer are currently propagated to the application layer, but this will only be useful in future releases. Current SSL session settings include triple DES encryption, with HMAC SHA-1 for message authentication. In addition, a similar mechanism is used to protect access to the RMI registry.

SmartFrog supports dynamic loading of stubs during RMI calls, or arbitrary resources in signed jar files using an enhanced RMI class loader. In both cases the loading sources is restricted to a configured codebase, and use the Java 2 mechanisms for signed JAR files described above.

All SmartFrog core classes use the security hooks described above for loading resources and communicating with other peers. In addition, the set-up of the security mechanisms is done as early as possible in the initialization of daemons to minimize exposure.

43.1 Assumptions

The process of setting up the critical components of SmartFrog is safe. For example, the Zip file containing the release has not been tampered with or, if keys are centrally created, their confidentiality has not been compromised when they were ship to the target host.

Similarly, SmartFrog critical components are protected by the underlying OS after they have been set-up. For example, everything under the directory "`private/`" can only be read/modified by the authorized user account that is using SmartFrog (the security credentials are in that directory). Moreover, an attacker cannot modify the SF core classes, basic scripts, or the JVM itself. In general, SmartFrog is not trying to solve vulnerabilities of the platform, just hoping that things do not get worse.

The current SmartFrog infrastructure is only as secure as the applications that it deploys, unless significant application customization is done (e.g., spawn processes under a different user with less privileges).

The implementation of the security mechanisms in SmartFrog uses many third-party packages to quickly implement the security mechanisms. This implies that if vulnerabilities are found in one of these components, e.g., Sun's secure random number generator, this will have a devastating effect on the implementation.

43.1 Known limitations and future enhancements

Some of the security assumptions will be relaxed in future releases of SmartFrog. The next logical enhancement is to isolate the SmartFrog infrastructure from the applications that it deploys and manages. Since restrictions should not be placed on what SmartFrog can deploy, OS support will be required for that.

Once isolation can be guaranteed, a common platform can be used to deploy applications for non-mutually trusted clients. This will require virtualizing the deployment service so that clients just can view and interact with their own applications.

The next step will be deploying applications that spawn multiple domains of trust, i.e., multiple SFTC, in which a non-hierarchical trust model and more accountability is required. This federated model, together with a new revocation mechanism, will limit the exposure when the security credentials of a valid member are compromised.

Finally, denial of service attacks are not prevented, and some of third party packages that are used are particularly exposed to these attacks. Re-engineering SmartFrog to be more robust against these attacks is a significant effort.

44 Properties

SmartFrog may have a number of properties defined that alter the behaviour of the daemon. Some of these are required to be defined on the command line (typically those that affect the way Java itself works), others may be defined in a file (often `default.ini`) and passed to the daemon on the command line.

These properties are captured in the class:

```
org.smartfrog.sfcore.common.SmartFrogCoreProperty
```

Some interesting properties are:

- `org.smartfrog.sfcore.common.Logger.logStackTrace`: Optional boolean property to include stack trace with error message. Default=false.
- org.smartfrog.sfcore.processcompound.`sfRootLocatorPort`: SmartFrog daemon connection port. Default=3800.
- org.smartfrog.sfcore.processcompound.`sfLivenessDelay`: Liveness check period (in seconds). Default=15.
- org.smartfrog.sfcore.processcompound.`sfLivenessFactor`. Liveness check retries. Default = 5.
- org.smartfrog.sfcore.processcompound.`sfProcessAllow`: Allow spawning of subprocess. Default=true.
- org.smartfrog.sfcore.processcompound.`sfProcessTimeout`: Subprocess creation/failure timeout. Default=60. Slower machines might need longer periods to start a new subprocess.
-

Properties that can only be defined using Java `-D` command line:

- `org.smartfrog.sfcore.processcompound.sfProcessName`: A user may create a sub-process of an existing root process by setting this system property. To start a SmartFrog daemon this property has to be set to “`rootProcess`”.
- `org.smartfrog.iniFile`: to load a file with properties to define JVM system properties. This file may not contain properties that affect security or code loading – it is read after these are already initialized.
- `org.smartfrog.sfcore.processcompound.sfDefault.REGISTRATION_NAME`: This property is used to describe some descriptions that should be deployed in all daemons and sub-processes. Each description will be registered in the process using `REGISTRATION_NAME` as `name`. It is possible to have many of this properties but the `REGISTRATION_NAME` cannot be repeated.
- `org.smartfrog.codebase`: Property used to define the codebase, a list of space separated URLs of JAR files used by the daemon.
- `org.smartfrog.sfcore.security.keyStoreName`: to load the private keys for the daemon.
- `org.smartfrog.sfcore.security.propFile=` to load security properties.

- `org.smartfrog.sfcore.security.debug:` to enable security debug information. Default=`false`;
- `java.security.policy`: Property to define Java security properties.
- `java.security.debug`: Property to define debug level in Java security.
- `java.rmi.server.logCalls`: Property to enable RMI logging. Default: `false`;
- `sun.rmi.loader.logLevel`: Property to define debug RMI level. Example: `VERBOSE`
- `java.security.manager`: Property to define the Java security manager.

Part1: A SmartFrog Example

45 Example

This section creates a simple example, covering all aspects including the creation of the SmartFrog description and the necessary Java classes.

The example is a variant on the normal "hello world" that is so favoured by books on programming. There will be two components – a printer and a message generator. The message generator will locate the printer, send a series of messages to it, and terminate. The printer, on the other hand, will wait until it is given a message, and then print it. It will terminate when told to do so.

The example is supplied with the SmartFrog distribution and is in package `org.smartfrog.examples.helloworld`.

For each of the two components, two aspects must be considered:

- How the description for the component is defined, what attributes must be provided and what the default attribute values are

- How the implementation is to be written, what is to be done in the various lifecycle phases and what threading is required

Following the definition of the components, consideration must be given as to how they are to be combined into a single application.

The Printer

The printer is a very simple component. It offers a single method that must be accessible via RMI, namely `printIt`, which must take a string and print it on the standard output. It has no specific initialization requirements, no threading and no specific need to clean up on termination.

To make it a little more interesting, the printer will prefix the output message with an identifier - either the `name` attribute if it is provided, or the name the component has in the application tree if it is not.

The Description – file "`printer.sf`"

The description needs only to link the component type to the implementing class. It does not need to provide a default value for the name of the printer, since the use of no name will cause a default to the name within the component tree.

```
Printer extends Prim {
    sfClass "org.smartfrog.examples.helloworld.PrinterImpl";
    name "";   //name - if empty, uses sfCompleteName
}
```

This fully describes the basic printer component. Host bindings will be done as part of the complete application description.

The Interface – file "`Printer.java`"

Since the printer has an RMI accessible method, that method must be described in an interface that extends `Remote`, and the method must throw the appropriate `RemoteException`. The interface is very simple:

```
package org.smartfrog.examples.helloworld;
import java.rmi.*;

public interface Printer extends Remote {
    public void printIt(String message) throws RemoteException;
}
```

This interface will be used by RMIC to generate the appropriate stub for the `PrinterImpl` class enabling other components to remotely call this method.

There is no requirement that every SmartFrog component declare new RMI interfaces; that is only a requirement if multiple components wish to talk directly to each other *and* it is not somehow required that these components must exist within the same JVM. Many components indirectly communicate with each other by way of the model, writing attributes during their lifecycle, and reading them when started up.

The Implementation – file "`PrinterImpl.java`"

The `PrinterImpl` class is a primitive component and so must follow the outline template provided in section (REF). In addition, it must implement the *Printer* interface.

Components which do not implement a custom Remote interface must still declare that they override a Remote RMI interface; this is a requirement of the Java RMI compiler. Here the `PrinterImpl` class declares its implementation of `Prim` which is one such interface.

Note that since there are no specific start or termination actions to carry out, there is no need to override the `sfStart` or `sfTerminateWith` methods.

```
package org.smartfrog.examples.helloworld;
import org.smartfrog.sfcore.prim.*;
import java.rmi.*;
import org.smartfrog.sfcore.Reference.*;
import org.smartfrog.sfcore.prim.*;

public class PrinterImpl extends PrimImpl implements Prim, Printer {

  /* any component specific declarations */

  // my name
  String name = "";

  // reference to my name attribute
  Reference nameRef = new Reference(ReferencePart.here("name"));

  // standard remotable constructor - must be provided
  public PrinterImpl() throws RemoteException {
  }

  // initialization template method
  public synchronized void sfDeploy()
        throws RemoteException, SmartFrogException {
    super.sfDeploy();
    /* any component specific initialization code */
    name = sfResolve(nameRef, "", false);
    if (name.equals("") name = sfCompleteName().toString();
  }

   /* component specific methods */
   public void printIt(String message) throws RemoteException {
     System.out.println(name + ": " + message);
   }
}
```

The Generator

The generator is a slightly more complex component. In particular, it must locate the printer, and call it with a sequence set of messages. This must be done in a separate thread started during the `sfStart` lifecycle method. The thread must be terminated during the termination template method if it has not already done so.

In addition to locating the printer, the component must obtain its set of messages and a frequency with which it should print these messages (defaulting to, say, one every 10 seconds).

The Description – file "`generator.sf`"

The description needs to link to the component type to the implementing class and define the other attributes with their default values. The link to the printer component, which will be LAZY because the RMI reference is required, cannot be provided until the whole application is created.

```
Generator extends Prim {
    sfClass "org.smartfrog.examples.helloworld.GeneratorImpl";
    frequency 10;   // default value set to 10 seconds
    messasges [];   // a vector of messages
    printer;        // link to the printer component
}
```

The Implementation – file "`GeneratorImpl.java`"

Unlike the printer, the generator has no externally accessed methods apart from those related to being a primitive component. Consequently, no additional interface needs to be defined. The generator, though, requires a thread – this may be done in many ways but extending `Runnable` and creating a thread from itself is the easiest.

```
package org.smartfrog.examples.helloworld;
import org.smartfrog.sfcore.Reference.*;
import org.smartfrog.sfcore.prim.*;
import java.util.*;
import java.rmi.*;


public class GeneratorImpl extends PrimImpl implements Prim,Runnable {

  /* any component specific declarations */
  Reference messagesRef = new Reference(
                         ReferencePart.here("messages"));
  Reference printerRef = new Reference(
                         ReferencePart.here("printer"));
  Reference frequencyRef = new Reference(
                         ReferencePart.here("frequency"));

  Printer printer;
  Vector messages;
  int frequency;

  Thread sender;
  boolean terminated = false; // notify the thread to terminate

  public GeneratorImpl() throws RemoteException {
  }

  public synchronized void sfDeploy()
       throws RemoteException, SmartFrogException {
    super.sfDeploy();

    printer = (Printer) sfResolve(printerRef, true);
    messages = sfResolve(messagesRef), [], true);
    frequency = sfResolve(frequencyRef,0,true)* 1000;
  }

  public synchronized void sfStart()
        throws RemoteException, SmartFrogException {
    super.sfStart();

    // create and start the thread
    sender = new Thread(this);
    sender.start();
  }

  public synchronized void sfTerminateWith(TerminationRecord tr) {
    // terminate the thread nicely if needed, can wait for it to
    // awake to actually do so

    terminated = true;
    super.sfTerminateWith(tr);
  }

  public void run() {
    // the body of the thread
    try {
      for (Enumeration en = messages.elements();
           en.hasMoreElements() & !terminated; ) {
        printer.printIt(en.nextElement().toString());
        if (frequency > 0) {
           Thread.sleep (frequency)
        }
      }
    } finally {
      // it doesn't matter calling this many times
```

```
        sfTerminate(TerminationRecord.normal(null));
      }
    }
}
```

Compiling the Components

The two classes and the interface all need to be compiled using the `javac` compiler. This has been done in the distribution and the classes are in the `sfExamples.jar` file.

Further, the two classes must be prepared for RMI by creating and compiling the stubs and skeletons. This is done using the *rmic* compiler that is invoked on the class:

```
rmic org.smartfrog.examples.helloworld.PrinterImpl
rmic org.smartfrog.examples.helloworld.GeneratorImpl
```

Both the classes must be compiled with `rmic` because although the generator has no interface as part of its own specialized behaviour, it does implement the *Prim* interface which may be access remotely – for example in the case of remote deployment.

If either of the classes will never be remotely accessed – for example if the entire application will always be run in a single JVM – the `rmic` steps may be skipped. In this case, the attribute `sfExport` must be set to `false` in the appropriate component descriptions. If this is not done, the SmartFrog system will attempt to load the stubs and skeletons for the classes and make the objects accessible remotely.

Again, the stubs and skeletons have been pre-compiled and are in the jar file.

The Combined Application

Now that the various parts are defined and compiled it is possible to define a few applications, consisting of one or more generators and printers, deployed on one or more hosts, as required.

Starting with the simplest, one of each on the same host:

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/examples/helloworld/printer.sf"
#include "org/smartfrog/examples/helloworld/generator.sf"

// the application must be called sfConfig
// a compound is a collection of components
sfConfig extends Compound {
  g extends Generator {
    messages ["hello", "world"];
    printer LAZY ATTRIB p;    // link to the instance of the printer
  }
  p extends Printer {
    name "myPrinter";
  }
}
```

Note that the printer is given a name rather than using the name derived from the tree (which would be “`p`” as the name is from, but not including, “`sfConfig`”). The generator uses the default frequency of 10.

The next example shows that it’s possible to have more than one generator using the same printer:

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/examples/helloworld/printer.sf"
#include "org/smartfrog/examples/helloworld/generator.sf"

// the application must be called sfConfig
// a compound is a collection of components
sfConfig extends Compound {
  g1 extends Generator {
    messages ["hello", "world"];
    // link to the instance of the printer
```

```
    printer LAZY ATTRIB p;
  }
  g2 extends Generator {
    messages ["hello", "world", "again"];
    frequency 5;
    // link to the instance of the printer
    printer LAZY ATTRIB p;
  }
  p extends Printer {
    name "myPrinter";
  }
}
```

(In this case, there is a problem with termination that will be explained later.)

A more complex example is one where we define a new component that consists of a printer and generator pair. This pair will print to each other, and the application consists of two of these pairs.

```
#include "org/smartfrog/components.sf"

#include "org/smartfrog/examples/helloworld/printer.sf"
#include "org/smartfrog/examples/helloworld/generator.sf"

  // a compound is a collection of components
Pair extends Compound {

  g extends Generator {
    messages ["hello", "world"];
    // link to the instance of the printer
    printer LAZY ATTRIB p;
  }
  p extends Printer;
}
sfConfig extends Compound {
  pair1 extends Pair;
  pair2 extends Pair;
}
```

In this case the printer component has not been given a name, so will use the one derived from the tree – `pair1:p` and `pair2:p` respectively. Each generator will find the associated printer, even though they are using the same link, because these links are resolved in different contexts.

There is, however, a problem with this example and the previous one. Since the lifecycle of the whole application is connected, the first of the two generators to finish will cause the whole application to terminate even though the other may not have finished yet. This may be solved by the use of the workflow components, but this is not covered in this reference manual.

Enriching the example a little further, the pair should optionally be parameterizable with the set of messages and the frequency, but without requiring the user to know the structure of the components contained within it.

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/examples/helloworld/printer.sf"
#include "org/smartfrog/examples/helloworld/generator.sf"

Pair extends Compound {
  // default value for param
  messages ["this is a", "boring", "set of strings"];

  // ditto
  frequency 10;

  g extends Generator {
    // link to container's value
    messages PARENT:messages;
    // link to container's value
    frequency PARENT:frequency;
    // link to the instance of the printer
    printer LAZY p;
  }

  p extends Printer;
}


sfConfig extends Compound {
```

```
  pair1 extends Pair {
    messages ["hello", "world", "again"];
  }
  pair2 extends Pair {
    frequency 5;
  }
}
```

This example shows the use of the parameterization pattern described in section .

The final example is one where the hosts on which various components are to be deployed may be modified. Taking the last example, we will enrich the notion of *pair* with two hosts – the printer host and the generator host, both defaulting to the local host.

In the combination, we will put the printer of one pair and the generator of the other pair on one host, and the reverse on a second host.

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/examples/helloworld/printer.sf"
#include "org/smartfrog/examples/helloworld/generator.sf"

Pair extends Compound { // a compound is a collection of components
  printerHost "localhost";
  generatorHost "localhost";


  // default value for parameter messages
  messages ["this is a", "boring", "set of strings"];

  // ditto
  frequency 10;

  g extends Generator {
    sfProcessHost generatorHost;
    messages PARENT:messages;    // link to container's value
    frequency PARENT:frequency;  // ditto
    printer LAZY p;    // link to the instance of the printer
  }

  p extends Printer {
sfProcessHost printerHost;
  }
}


sfConfig extends Compound {
  hostA "foo.smartfrog.org";
  hostB "bar.smartfrog.org";

  pair1 extends Pair {
    printerHost hostA;
    generatorHost hostB;
    messages ["hello", "world", "again"];
  }
  pair2 extends Pair {
    printerHost hostB;
    generatorHost hostA;
    frequency 5;
  }
}
```

By modifying the `hostA` and `hostB` attributes, different deployments may be obtained..

Part1: Appendices

46 Exit Codes for SmartFrog Scripts

```
0 - Success
1 - General Errors
69 - Bad Arguments input to Script
130 - Script terminated by Control-C
```

47 Tree Factories

There is a very low-level feature in the SmartFrog parser which allows custom classes to generate the trees created during parsing. This is a feature which users of the framework are not encouraged to use -it is unstable and normally delivers little value, however it is a feature which exists and which is documented herein.

Consider it the SmartFrog equivalent of the XML parser factory: it is possible to define a new source of classes which will be created to represent the parsed definitions. This custom factory can generate different classes as desired.

The specification of a factory is defined in the system property

```
sfcore.languages.sf.factoryClass
```

The default value of this is

```
org.smartfrog.sfcore.languages.sf.DefaultFactory
```

A new factory allows the implementor to do two things

1. Provide a new base class for all component descriptions created by way of the `extends` keyword.

2. Provide custom classes whenever a specific component description subclass is explictly requested by way of the sequence `extends:name` in a SmartFrog specification.

Whenever a ComponentDescription element is required, the factory will be given the the extends name (or `default` when no explicit name is provided, and `root` for the root node). It must then create and return an implementation of `SFComponentDescription`, which is usually done by returning an instance or subclass of `SFComponentDescriptionImpl`. These component descriptions are expected to generate the live ComponentDescriptions on deployment, so can be used to ensure that even simple ComponentDescriptions are always handled by a custom class, which can do advanced operations such as push its state into a shared tuple-space.

This feature is very low level and cannot be considered stable. However, it exists and has been used to integrate SmartFrog models with distributed tuple-space platforms. In the open source philosophy, the SmartFrog team encourage people to explore and use this feature if it meets a need, however they must be aware of the risk: this is a low-level feature whose stability is uncertain.