RC24760 (W0903-030) March 10, 2009
Computer Science

IBM Research Report

SPADE Language Specification

Martin Hirzel, Henrique Andrade, Bugra Gedik, Vibhore Kumar,
Giuliano Losa, Robert Soulé, Kun-Lung Wu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 703
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zuric

March 11, 2009

SPADE Language Specification

Martin Hirzel, Henrique Andrade, Buğra Gedik, Vibhore Kumar, Giuliano Losa, Robert Soulé, and Kun-Lung Wu.

Contents

1 Introduction

This document describes the language design of the SPADE language, version 2. SPADE is the programming language for InfoSphere Streams, IBM's high-performance distributed stream processing system [1]. This document only focuses on the syntax and semantics of user-visible features; a description of the implementation design or the inner workings of the compiler is out of scope for this document. SPADE 2 is not backward compatible to SPADE 1, and instead takes the opportunity to clean up several features.

This document is sprinkled with paragraphs containing auxiliary information:

- Practical advice: Best practices and conventions for users.
- Implementation note: Note about how the compiler or runtime implements a feature.
- For SPADE 1 users: Comparisons between old and new language features.
- For language experts: Terminology from the programming language community.
- Language design rationale: Justification for decisions where we had to reconcile conflicting design goals.

In addition, there are numerous code examples. IBM's SPADE compiler is continuously tested on these examples. While some examples are semantically incomplete (e.g., using undefined identifiers), all examples are syntactically valid.

1.1 Language Overview

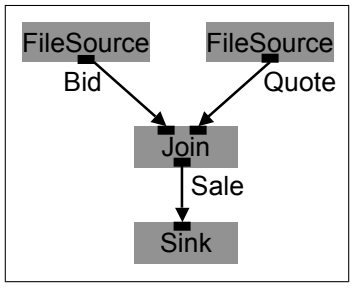

Figure 1: Stream graph for sale-join example.

SPADE is a stream programming language. Figure 1 shows an example stream graph. The following code shows how it could be implemented in SPADE.

```
composite SaleJoin {                                                          // 1
graph                                                                         // 2
  stream<string buyer, string item, decimal64 price> Bid = FileSource() {     // 3
    param fileName: "BidSource.dat"; format: csv;                             // 4
  }                                                                           // 5
  stream<string seller, string item, decimal64 price> Quote = FileSource() { // 6
    param fileName: "SaleSource.dat"; format: csv;                            // 7
  }                                                                           // 8
  stream<string buyer, string seller, string item> Sale = Join(Bid; Quote) { // 9
    window Bid   : sliding, time(30);                                         //10
           Quote : sliding, count(50);                                        //11
    param  match : Bid.item == Quote.item && Bid.price >= Quote.price;        //12
    output Sale  : item = Bid.item;                                           //13
```

```
  }                                                                          //14
  () = FileSink(Sale) { param fileName: "Result.dat"; format: csv; }         //15
}                                                                            //16
```

A SPADE program consists of one or more composite operators. A composite operator defines a stream graph. The vertices of the stream graph are operator invocations, and the edges are streams. An operator invocation defines output streams by invoking a stream operator on input streams. For example, the operator invocation in Line 9 invokes the `Join` operator on two input streams `Bid` and `Quote` to produce one output stream `Sale`. An operator invocation may also have zero input streams (e.g., `FileSource` in Lines 3 and 6) or zero output streams (e.g., `FileSink` in Line 15). The body of an operator invocation configures the way the operator works. For example, the `window`, `param`, and `output` clauses in Lines 10-13 configure how the `Join` operator is invoked.

This language specification is written bottom-up, starting from types such as `string` or `int32` (Section 2). Values of these types are manipulated by expressions, such as `Bid.item == Quote.item` (Section 3). But the core concept of SPADE is the operator invocation, such as `Sale = Join(Bid; Quote)` (Section 4). SPADE allows users to define their own primitive or composite operators, such as `SaleJoin` (Section 5). Finally, a SPADE program specifies functions and operators in namespaces (Section 6). The appendix contains an extended example (A), a grammar overview (B), acknowledgements (C), and an index.

1.2 Summary of Changes to Spade 1

For SPADE 1 users: This document refers to the original SPADE language as "SPADE 1". It is described at `http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.spade.html`. There is also a research paper about SPADE in SIGMOD'08 [4]. The SPADE 2 language specification is self-contained, so it should be accessible to people new to SPADE.

SPADE 2 adds some features missing from SPADE 1, removes others, and changes many features. Newly added features include composite operators, shared variables, and a richer data model (`tuple`, `map`, `decimal`, `timestamp`, etc.). Removed features include bundles, bulk functions, and the preprocessor. The most visible change in SPADE 2 is the syntax, which has become more readable, especially for programmers familiar with C or Java. For example, in SPADE 1, the operator invocation for `Sale` looks like this:

```
stream Sale(buyer: String, seller: String, item: String) # 1
  := Join(Bid <time(30)>; Quote <count(50)>)              # 2
     [ $1.item = $2.item & $1.price >= $2.price ]         # 3
     { item := $1.item }                                  # 4
```

Language design rationale: The design of SPADE 2 has the same spirit as SPADE 1: the primary goal is permitting efficient implementation on distributed hardware. Most changes aim to make the language simpler and more uniform. When given a choice between code readability and writeability, we usually opted for readability.

1.3 Grammar Notation

This language specification adopts a flavor of BNF (Backus Naur Form) to describe the syntax of language features, following the following conventions:

italics	Non-terminal
ALL_CAPS_ITALICS	Token, e.g., *ID* for identifiers
'`fixed-width font`'	Verbatim text, quoted to avoid confusion with meta-characters, e.g., '`(`'
(...)	Grouping, to disambiguate meta-syntax precedence
... `|` ...	Alternatives, match either syntax left or right of bar
...$^{?}$	The preceding syntax is optional
...*	The preceding syntax is repeated zero or more times
...$^{+}$	The preceding syntax is repeated one or more times
...*,	Comma-separated list of zero or more items
...$^{+,}$	Comma-separated list of one or more items
...$^{*;}$	Semicolon-separated list of zero or more items
...$^{+;}$	Semicolon-separated list of one or more items
nonTerminal ::= ...	Rule definition

1.4 Lexical Syntax

SPADE files are written using 8-bit ASCII characters (the subset 32-126 of the Latin-1 alphabet ISO 8859-1, which coincides exactly with the same code points in unicode). Identifiers start with a letter or underscore, followed by letters, digits, or underscores. The syntax for literal values of primitive types (numbers and strings) is similar to that of Java or C++, and is described in Section 2.1.

SPADE has two forms of comments: single-line comments (from `//` to the end of the line) and delimited comments (between `/*` and `*/`). Delimited comments can span multiple lines, and can be followed by regular code in the same line.

SPADE syntax is not sensitive to indentation or line breaks. SPADE syntax is case-sensitive, for example, `mud` and `Mud` are different identifiers.

SPADE uses lexical scoping for identifiers. In other words, a declaration in an inner scope shadows declarations of the same identifier in statically enclosing scopes. SPADE does not permit synonymous entities of different categories in the same scope. For example, it is a compiler error if a program declares both an operator named `f` and a function named `f` in the same scope. As another example, it is a compiler error if a program declares both a type named `t` and a variable named `t` in the same scope. In other words, SPADE does not segregate scopes by identifier categories, unlike for example Java. That also means that identifiers in inner scopes shadow identifiers in outer scopes even when they are of a different category. For example, a locally declared stream `s` hides any function `s` in an outer scope.

Practical advice: We encourage developers to imitate the indentation and line breaks style demonstrated in the examples in this document. We encourage starting identifiers for operators, aggregations, and user-defined types with an upper-case letter, and starting functions, variables, and built-in types with a lower-case letter. We encourage `CamelCase` instead of `underscore_style` to separate words in identifiers. We encourage starting shared variable identifiers with `s_`.

For SPADE 1 users: In SPADE 1, single-line comments started with `#` and delimited comments were surrounded by `#*...*#`.

2 Types

SPADE has a wide array of primitive types tailored for streaming, and a small set of composite types inspired by scripting languages but statically checked. Figure 2 shows all types arranged in a hierarchy. This section describes the types, how to define them, and how to convert values between different types.

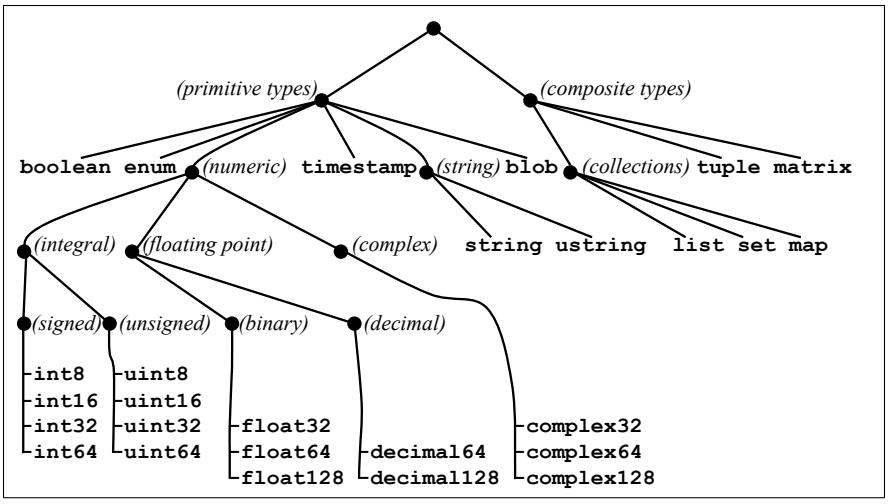

Figure 2: Hierarchy of SPADE types.

2.1 Primitive Types

A primitive type, such as `int32` or `string`, is one that is not composed of other types. SPADE supports the following primitive types:

`boolean`	`true` or `false`
`enum`	user-defined enumeration of identifiers
`int`*b*	signed *b*-bit integer, one of:
`int8`	(-128 to 127)
`int16`	(-32,768 to 32,767)
`int32`	(-2,147,483,648 to 2,147,483,647)
`int64`	(-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
`uint`*b*	unsigned *b*-bit integer, one of:
`uint8`	(0 to 255)
`uint16`	(0 to 65,535)
`uint32`	(0 to 4,294,967,295)
`uint64`	(0 to 18,446,744,073,709,551,615)
`float`*b*	IEEE 754 binary *b*-bit floating point number, one of:
`float32`	(single-precision, equivalent to `float` in Java)
`float64`	(double-precision, equivalent to `double` in Java)
`float128`	(same as `binary128` in the IEEE 754-2008 standard)
`decimal`*b*	IEEE 754 decimal *b*-bit floating point number, one of:
`decimal64`	(significand 16 decimal digits, exponents 10^{-383} to 10^{384})
`decimal128`	(significand 34 decimal digits, exponents $10^{-6,143}$ to $10^{6,144}$)
`complex`*b*	*2b*-bit complex number, one of:
`complex32`	(both real and imaginary part are `float32`)
`complex64`	(both real and imaginary part are `float64`)
`complex128`	(both real and imaginary part are `float128`)
`timestamp`	point in time, with nanosecond precision
`string`	string of 8-bit characters
`ustring`	string of UTF-16 unicode characters, based on ICU library
`blob`	sequence of raw bytes
`string[`*n*`]`	bounded-length string of at most *n* 8-bit characters

Besides these primitive types, SPADE also defines meta-types for compile-time entities in operator parameters, see Section 5.2.4.

An example for an enumeration is

```
type DebugLevel = enum { error, info, debug, trace };
```

Any of the identifiers `trace`, ..., `fatal` can be used where a value of enumeration `DebugLevel` is expected.

Like in C/Java, literals for `int`, `uint`, `float`, and `decimal` can have optional type suffixes. For example, `123` is signed (`int32`) whereas `123u` is unsigned (`uint32`). One suffix indicates the kind of number:

`s ->` signed integer (default for integer literals)
`u ->` unsigned integer
`f ->` binary floating-point (default for floating point literals)
`d ->` decimal floating-point

Another suffix indicates the number of bits:

`b (byte)` -> 8-bit
`h (half-word)` -> 16-bit
`w (word)` -> 32-bit (default for integer literals)
`l (long)` -> 64-bit (default for floating point literals)
`q (quad-word)` -> 128-bit

Some more examples for literals with type suffix: `0.0005` (`float64`), `0.5e-3` (`float64`), `3.5d` (`decimal64`), `3.5w` (`float32`), `123d` (`decimal64`), `123dq` (`decimal128`),

Strings may contain internal null characters, which, unlike in C, are not considered terminating. In other words, the length of a string is independent from whether or not it contains characters whose encoding is zero.

Unicode string literals can use escape sequences of the form \u*hhhh*, where the four hexadecimal digits *hhhh* specify a character. For example, `ustring("Bu\u01ffra")` uses the escape `\u01ff` to specify a ğ with a u-shaped accent on top. Recall from Section 1.4 that SPADE files are written in the ASCII character set, so letters such as ğ can only be written by escape sequence and can not appear directly in the source code.

Literals for complex numbers are written by using the type as a function name, as in `complex32(1.0, 2.0)`.

A timestamp stores TAI (International Atomic Time) with nanosecond precision. It uses a 128 bit representation, where 64 bits store the seconds since the epoch as a signed integer, 32 bits store the nanoseconds, and 32 bits store an optional identifier of the machine where the measurement was taken, which can be useful for after-the-fact drift compensation. The SPADE library provides various functions for manipulating timestamps:

- Getting TAI on the local machine. Usually, this involves obtaining UTC from the operating system, then adding the correct leap second offset.
- Converting back and forth between a TAI timestamp value and an ISO time string. An example for an ISO time string is `"1960-01-01T23:03:20"`.
- Converting back and forth between a TAI timestamp value and its second, nanosecond, and machine identifier components represented as integers.
- Converting back and forth between TAI, UTC, and UT time values, by consulting the appropriate conversion tables.

Many operators and functions are overloaded to work with different types. For example, the operator `+` can add various types of numbers, but it can also concatenate strings. Likewise, the function `length(x)` is overloaded to accept `x` of type `string` or `ustring`.

SPADE offers a bounded-length variant of some types. For example, `string[5]` can store any strings with at most 5 characters. The compiler prohibits implicit conversions from unbounded to bounded types, but the user can override that by explicit casts. Type bounds, whether in variable declarations or in casts, must be compile-time constants. A cast from any string to a bounded string truncates the value if it is too long. SPADE limits all strings, bounded or unbounded, 8-bit or unicode, to at most 2^{32} characters. SPADE does not offer bounded unicode strings.

Blobs are sequences of at most 2^{64} raw bytes.

Implementation note: The language specification purposely does not specify a byte order, because users are oblivious to these details within a SPADE application. Compilers are expected to provide flags to choose, for example, network byte order or native byte order. The on-the-wire format for the streaming middleware will be specified in a separate forthcoming document. The internal representation of both bounded and unbounded strings and blobs stores a separate length field. As with all types, the exact layout is implementation dependent and not exposed at the language level. Typically, bounded types would be padded in case the length is lower than the bound, thus allowing subsequent attributes to be stored at a fixed offset in a network packet. The implementation may also reduce the number of bits in the length field according to the bound to save space.

Practical advice: Use the `decimal` floating point types in financial, commercial, and user-centric programs to avoid losing decimal digits to binary rounding. Use TAI instead of UTC or UT wherever possible. For unstructured data of bounded size, use a `list<uint8>[`n`]` instead of a `blob`, see Section 2.2.

Language design rationale: SPADE has an unusually large zoo of primitive types, because in network applications, users need a lot of control over the data representation to achieve performance. Tight representation is important both to keep the data on the wire small, and to reduce serialization and deserialization overheads. The `timestamp` type is designed to allow a high degree of precision as well as avoid overflow conditions (it can represent values ranging over billions of years) by following widely accepted standards. The bounded types permit efficient marshaling and unmarshaling of network packets: if all parts have a fixed size, then parts can be found at a fixed offset without decoding the whole. We considered using slicing instead of casts to convert unbounded to bounded values, but decided against it because unlike type bounds, slicing parameters are not necessarily compile-time constants, and slicing does not alter the type.

For SPADE 1 users: Several primitive types are new in SPADE 2, e.g., `uint`, `ustring`, `decimal`, or `timestamp`.

2.2 Composite Types

A composite type is the result of applying a type constructor to one or more type arguments. For example, the composite type `list<int32>` is the result of applying the type constructor `list<`T`>` to the type argument T=`int32`. For SPADE 1 users: The only composite types in SPADE 1 were matrix and list; SPADE 2 introduces additional composite types tuple, set, and map. Also, in SPADE 1, a list could only contain primitive values; in SPADE 2, composite types (except for matrix) nest arbitrarily.

2.2.1 Collections

SPADE has three built-in collections (list, set, and map) with the following type constructors:

`list<`T`>`	list (random-access zero-indexed sequence)
`set<`T`>`	set (unordered collection without duplicates)
`map<`K`,`V`>`	map (unordered mapping from key type K to value type V)
`list<`T`>[`n`]`	like `list<`T`>`, but bounded-length
`set<`T`>[`n`]`	like `set<`T`>`, but bounded-length
`map<`K`,`V`>[`n`]`	like `map<`K`,`V`>`, but bounded-length

Like primitive strings, collection types have bounded-length variants, which follow the same rules. For example, a `list<int32>[2]` can store at most two integers. It is legal to declare bounded-size types with

unbounded elements, such as `list<string>[3]`, though that obviously does not offer the same marshaling optimization opportunities. SPADE limits all containers, bounded or unbounded, to at most 2^{32} elements.

The literal syntax for list, set, and map values is as follows:

listLiteral	::=	'[' *expr**, ']'	# e.g. `[1, 2, 3]`
setLiteral	::=	'{' *expr**, '}'	# e.g. { "IBM", "GOOG" }
mapLiteral	::=	'{' (*expr* ':' *expr*)*, '}'	# e.g. { "Mon": -1, "Fri": 1 }

List, set, and map types can be arbitrarily nested. In other words, their elements, and in the case of maps even their keys, can be of any type, including other composite types.

As we will see in Section 3, composite types have their own operators. You can subscript a list or map with `l[i]`, check membership in a collection with "`x in s`", iterate over a collection with "`for(T x in s)` ...", and so on. There are also various functions to work on collections; for example, `union(set1, set2)` or `intersection(set1, set2)`. For lists and maps, the left operand of the `in` operator is the key. In other words, `"Oz" in m` checks whether `"Oz"` is a key of the map `m`. Value (as opposed to key) membership tests use functions, not the `in` operator.

Implementation note: Lists are implemented via arrays, but unlike C, the language uses static or dynamic checks to protect against out-of-bounds accesses. Sets and maps are implemented via hash tables. They are unordered and support constant-time lookup.

2.2.2 Tuples

A tuple is a set of attributes, and an attribute is a named value. For example, the tuple {`sym="Fe", no=26`} consists of two attributes `sym="Fe"` and `no=26`. The type for this tuple is `tuple<string sym, int32 no>`. Tuples are similar to database rows in that they have an unordered set of named and typed attributes. Tuples differ from objects in Java or C++ in that they do not have methods.

Tuple types can extend other tuple types, for example:

```
type Loc2d = tuple<int32 x, int32 y>; // 1
     Loc3d = Loc2d, tuple<int32 z>;   // 2
```

The resulting type `Loc3d` is equivalent to `tuple<int32 x, int32 y, int32 z>`. In tuple extension, the identifier may also be the name of a stream as a shorthand for the type of the tuples in that stream. For example:

```
stream<int32 x, int32 y> LocStream = Source() {/*...*/} // 1
type LocWithId = LocStream, tuple<string id>;           // 2
```

The resulting type `Loc3d` is equivalent to `tuple<int32 x, int32 y, string id>`. The syntax for tuple types is:

tupleType	::=	'tuple' '<' *tupleBody* '>'	
tupleBody	::=	*attributeDecl*$^{+,}$	# attributes
	|	(*ID* | *tupleType*)$^{+,}$	# tuple type extension
attributeDecl	::=	*type ID*	

In other words, the body of the tuple type either consists of a comma-separated list of attribute declarations, or a comma-separated list of tuple types, either by name or written in-place. Tuple types can be arbitrarily nested. In other words, their attributes can be of any type, including other composite types, even other tuple types. Here is an example:

```
type Loc2d  = tuple<int32 x, int32 y>;        // 1
     Sensor = tuple<Loc2d loc, string color>; // 2
```

The resulting type `Sensor` is equivalent to `tuple<tuple<int32 x, int32 y> loc, string color>`. For example, given a variable `Sensor s`, the expression `s.loc.x` accesses the x-coordinate.

As we will see in Section 5.1.2, the explicit `tuple<...>` type constructor can be omitted at the top level of a type definition. That means that the previous example can also be written like this:

```
type Loc2d  = int32 x, int32 y;        // 1
     Sensor = Loc2d loc, string color; // 2
```

The literal syntax for tuple values is as follows:

tupleLiteral ::= '{' (*ID* '=' *expr)*, '}' # e.g. { x=1, y=2 }

You can access an attribute of a tuple with dot notation, e.g., `myLocation.x`.

Language design rationale: Attributes within a tuple are unordered for semantic consistency with relational databases. Users should treat tuples as a high-level concept, and not order fields in a misguided effort to optimize the representation. Such an effort would be futile, because the runtime implementation uses a canonical ordering of attributes anyway, and the representation is invisible to the user.

2.2.3 Matrices

A matrix is a two-dimensional array of numbers for use in linear algebra. The type constructor is:

matrixType ::= 'matrix' '<' *type* '>' '[' *expr* ',' *expr* ']'

For example, `matrix<int32>[3,2]` is the type for a matrix with 3 rows and 2 columns. To create a matrix value, use the matrix type as a function, and pass a nested list literal to it, like this:

```
matrix<int32>[3,2]([[0,1], [2,3], [4,5]])
```

This would produce a matrix with three rows, where row 0 is `[0,1]`, row 1 is `[2,3]`, and row 2 is `[4,5]`. Unlike other composite types, the matrix type does not permit arbitrary nesting. The elements of a matrix must be either `boolean` or numeric (`int`, `uint`, `float`, `decimal`, or `complex`). As we shall see in Section 3, you can subscript and even slice a matrix with `[...]`, and you can use various arithmetic operators on a matrix (addition, multiplication, etc.).

Implementation note: 2D matrix types are implemented efficiently with the uBlas libraries. Other dimensionalities (such as 3D tensors) are not currently supported.

Language design rationale: We only support 2D matrices, since they suffice for most applications. We use nested list literals for matrix literals, because that syntax is general enough, and using a different syntax was not warranted since users rarely write matrix literals in their code. Library functions support creating custom matrix values, such as a 40x40 identity matrix.

2.3 Value Semantics

All primitive and composite SPADE types have value semantics, not reference semantics. That means that:

- An assignment ("=") copies the contents, instead of aliasing the location.
- An equality comparison ("==") compares the contents, instead of comparing the locations.

One consequence of value semantics is that modifying the original after an assignment does not change any copies. Consider this example:

```
mutable map<string, tuple<int32 x, int32 y>> places = { }; // 1
mutable tuple<int32 x, int32 y> here = { x=1, y=2 };       // 2
places["Hawthorne"] = here;                                // 3
here.y = 3;                                                // 4
```

Line 3 initializes variable `here` with the value {`x=1,y=2`}. Line 4 assigns a copy of that value into the map at key `"Hawthorne"`. Line 5 modifies the version of the value in variable `here`, so variable `here` now contains {`x=1,y=3`}. However, this does not affect the copy at `places["Hawthorne"]`, which is still {`x=1,y=2`}. Another consequence of the value semantics is that since there is no notion of a "reference", there is no notion of "null", nor can there be any cyclic data structures in SPADE.

Language design rationale: We picked value semantics for SPADE because it is more natural, and more efficient, to treat streaming data as pure copies rather than having an identity in an address space. Disallowing references also prevents null pointer errors, simplifies memory management, and prevents unintended side effects of mutating a value stored in a container or used as a map key.

Practical advice: If you need a more powerful type system than that provided by SPADE, use SPADE's extension mechanisms and implement your functionality in a low-level language such as C++ or Java.

2.4 Type Conversions

SPADE is statically typed and strongly typed. The static typing is enforced by explicit type declarations and compile-time type checking. The strong typing is enforced by providing almost no implicit conversions between types. SPADE does offer explicit conversions where it makes sense. Explicit conversions look like function calls, e.g., `int32(2.5)` returns 2.

SPADE allows explicit conversion from a tuple type `WideT` to a tuple type `NarrowT` if `WideT` has a superset of the attributes of `NarrowT`. The conversion discards excess attributes and cannot be reverted.

Types in SPADE are equivalent if they are the same primitive types, or if they are composed from equivalent types using the same type constructor. For example, after the following:

```
type LocT1 = int32 x, int32 y;                                          // 1
     LocT2 = int32 y, int32 x; // different order, but same attributes // 2
```

variables of these two types can be assigned to each other:

```
LocT1 loc1 = { x=1, y=2 };          // 1
LocT2 loc2 = loc1; // this is legal // 2
```

As far as SPADE is concerned, `LocT1` and `LocT2` are the same type; the fact that they have different names and attribute orders is irrelevant, because a tuple is an unordered set of attributes. Therefore, the assignment between variables `loc1` and `loc2` is legal, and does not constitute a type conversion. (For language experts: SPADE uses structural equivalence.)

SPADE permits implicit conversions in variable initializers. For example, instead of `int64 x = int64(1);`, you can simply say `int64 x = 1;`. Likewise, instead of

```
matrix<float64>[2,2] m = matrix<float64>[2,2]([[0.0,0.0], [0.0,0.0]]);
```

you can directly say

```
matrix<float64>[2,2] m = [[0,0], [0,0]];
```

(For language experts: This implicit conversion prevents what is known as "type stuttering", the unnecessary repetition of the type type in the initializer.)

Unlike some other languages, SPADE has no implicit conversion from `int` to `string` (no `"num" + 1`), from `int` to `float` (no `1+2.0`), or from `int` to `boolean` (no `while(1)...`). To perform these conversions, you must make them explicit: `string(1)` or `float64(1)`. But not all combinations are supported; for example, `boolean(myInt)` is illegal, because it is shorter and less ambiguous to use integer comparison such as `myInt!=0`.

Language design rationale: SPADE adopts a strong static type system, because it saves time for runtime checks, and errors that are not prevented statically are difficult to track down in a distributed system. SPADE

uses structural equivalence, because that facilitates converting data to and from external applications, files, or databases that do not share the same type system. We acknowledge that structural equivalence can lead to types being considered equivalent even when they were intended to differ, but we decided for structural equivalence anyway to avoid adapter bloat, which can slow down programs and clutter up code.

For SPADE 1 users: Both SPADE 1 and SPADE 2 are statically typed and strongly typed, with few implicit conversions. The explicit conversion syntax of SPADE 1 is more verbose than that of SPADE 2.

3 Expression Language

Though SPADE is a streaming language, there are many places where it uses expressions found in traditional imperative or functional languages. Some expressions are evaluated at runtime: in operator output assignments or operator parameters. Some expressions are evaluated purely at compile time, during code generation: these include operator pragmas and window clauses. And finally, there are places where SPADE accepts not only expressions, but even statements like those in traditional languages: this is the case in operator logic, and in function definitions.

3.1 Expression Operators

Expression operators are used to compute values, e.g., with `+` or `-`. They are not to be confused with stream operators, used to connect streams into a data flow graph. The SPADE operator table looks more or less like that of C or Java:

e`(...)`	N	Function call or type cast
e`[...]`	N	Map lookup, subscript or slice into string / blob / list / matrix
e`.`*id*	1	Tuple attribute access
`++ -- ! - ~`	1	Increment, decrement, prefix logical / arithmetic / bitwise negation
`* / %`	2	Multiplication, division, remainder
`+ -`	2	Addition, subtraction
`<< >>`	2	Bitwise shift left, right
`< <= > >= != ==`	2	Comparison (value semantics)
`&`	2	Bitwise and
`^`	2	Bitwise xor
`|`	2	Bitwise or
`&&`	2	Logical and (short-circuit)
`||`	2	Logical or (short-circuit)
`?:`	3	Ternary conditional
`in`	2	Membership in list / map / set
`= += -= *= ...`	2	Assignment (value semantics)

In this table, precedence goes from high (at the top) to low (at the bottom). Literals, such as strings, maps, tuples, lists, etc., have higher precedence than the highest-precedence operator. The middle column of the table indicates arity (the number of operands). All the binary operators (arity 2) are left-associative.

Besides the simple assignment operator (`=`), the following operators first perform a regular binary operation, and then an assignment: `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, and `|=`.

As mentioned in Section 2.3, SPADE uniformly uses value semantics. Hence, even in the case of composite values, assignments always copy the contents, and comparisons always compare the contents, never the location, of a value.

Side-effecting expressions are only allowed as top-level statements, not nested in other expressions, e.g.:

```
mutable int32 x, y;                                                    // 1
y = x++     + x++    ; //illegal, since "++" has side effect          // 2
x++;                   //legal as top-level statement                  // 3
```

```
y = (x=x+1) + (x=x+1); //illegal, since "=" has side effect                // 4
x = x + 1;             //legal as top-level statement                     // 5
y = inc(x)  + inc(x) ; //may be legal, if "inc" does not have side effects // 6
inc(x);                //legal as top-level statement                     // 7
```

Restricting side-effects to top-level statements makes code more readable, prevents common programming mistakes, and may lead to more optimization opportunities. The compiler checks whether a function such as `inc` is side-effect free by checking that it is not stateful and none of its parameters are mutable (see Section 3.4).

The right-shift operator (`>>`) behaves differently for signed and unsigned integers. For signed integers, it fills in with the sign bit, whereas for unsigned integers, it fills in with zero. Use casts if you want to override this behavior.

String, blob, list, and matrix subscripts can either retrieve a single element, or can perform slicing. Map subscripts can only refer to one element, not a slice, since maps are unordered. All string, blob, list, and matrix indexing is zero-based, and slices include their lower bound but exclude their upper bound. For example, if "`a`" is a list, then `a[2:5]` is the same as the list `[a[2],a[3],a[4]]`. If the lower bound is omitted, the slice starts at element zero; if the upper bound is omitted, the slice continues until the last element. For example, if the last element of `a` has index `9`, then `a[7:]` is the same as the list `[a[7],a[8],a[9]]`. A matrix subscript is a comma-separated list of indices or slices, one for each dimension. For example, if `m` is a matrix, then `m[1,:]` retrieves the entire row `1`, because the slice for the column dimension (`:`) omits both lower and upper bounds, and thus defaults to `0:C`, where `C` is the number of columns. Here is the syntax:

subscriptExpr ::= *expr* '`[`' *subscript*$^{+,}$ '`]`'
subscript ::= *expr* | (*expr*$^{?}$ '`:`' *expr*$^{?}$)

Practical advice: Invalid subscripts cause runtime errors, see Section 4.6. If you are not certain whether a subscript is valid, guard it with a defensive membership test, for example:

```
if ("Oz" in places)            // 1
  munchkinland = places["Oz"]; // 2
```

3.2 Mapped Operators

Spade takes inspiration from Matlab and supports auto-vectorization of expression operators to work on containers and matrices.

For a binary operator such as `+` or `*`, if one operand is a scalar and the other a container or matrix, the operator applies on each element. For example, `2 * [1,2,3] == [2*1, 2*2, 2*3] == [2,4,6]`. If both operands are matrices, `*` denotes matrix multiplication.

Spade also has mapped versions of some binary operators, written by "dotting" the operator, such as `.+` or `.*`. If both operands are lists, maps, or matrices of the same size, the dotted operators work on corresponding pairs of elements at a time. For example, `[3,2] .* [5,4] == [3*5, 2*4] == [15,8]`. Mapped operators have the same precedence as their non-dotted counterparts:

`.* ./ .%`	Mapped multiplication, division, remainder
`.+ .-`	Mapped addition, subtraction
`.<< .>>`	Mapped bitwise shift left, right
`.< .<= .> .>= .!= .==`	Mapped comparison
`.&`	Mapped and
`.^`	Mapped xor
`.|`	Mapped or

For Spade 1 users: Spade 1 implicitly mapped both expression operators and functions over lists. Spade 2 does not provide implicitly mapped functions, because the user can achieve the same effect with an

explicit function and loops, and because doing it implicitly would cause confusion with side effects and with SPADE 2's richer type system.

3.3 Statements

Statements are the traditional building blocks of imperative languages. As a streaming language, SPADE does not need statements most of the time, but they are permitted inside operator logic and function definitions. SPADE's assortment of statements is deliberately simple; if you want to write sophisticated imperative code, you should use SPADE's extension features and put it in a low-level language such as C++ or Java.

A local variable declaration consists of an optional `mutable` modifier and a type, followed by a comma-separated list of declared identifiers with optional initializer expressions, followed by a semicolon:

$$\textit{localDecl} \quad ::= \quad \text{`}\texttt{mutable}\text{'}^{?} \;\; \textit{type} \;\; (\;\; \textit{ID} \;\; (\;\; \text{`}\texttt{=}\text{'} \;\; \textit{expr} \;\;)^{?} \;\;)^{+,} \;\; \text{`}\texttt{;}\text{'}$$

An example local variable declaration is `mutable int32 i=2, j=3;`. The syntax is similar to C or Java, except that the type does not get tangled up with the variable. For example, SPADE does not have declarations like `int32 x,y[],z;`. Immutable local variables (without `mutable` modifier) must be initialized.

A block consists of zero or more statements or variable declarations, surrounded by curly braces:

$$\textit{blockStmt} \quad ::= \quad \text{`}\{\text{'} \;\; \textit{stmt}^{*} \;\; \text{`}\}\text{'}$$

An example block is { `int32 i=0; foo(i, 2);` }. Local variables declared in a block remain in scope until the end of the block. Blocks are often used as bodies for control statements like `if`/`while`/`for`, or as function bodies.

An expression statement consists of an expression followed by a semicolon:

$$\textit{exprStmt} \quad ::= \quad \textit{expr} \;\; \text{`}\texttt{;}\text{'}$$

An example expression statement is `foo(i, 2);`. Obviously, the purpose of an expression statement is its side-effect. The only operator that have non-error side-effects are assignments, certain function calls, and increment/decrement (`++`/`--`) operators.

An `if` statement can have an optional `else` clause:

$$\textit{ifStmt} \quad ::= \quad \text{`}\texttt{if}\text{'} \;\; \text{`}\texttt{(}\text{'} \;\; \textit{expr} \;\; \text{`}\texttt{)}\text{'} \;\; \textit{stmt} \;\; (\;\; \text{`}\texttt{else}\text{'} \;\; \textit{stmt} \;\;)^{?}$$

Dangling `else` is resolved to the innermost `if`; you can override this with blocks. SPADE does not have a C-style `switch` statement.

A `for` statement loops over the elements of a list, set, or map:

$$\textit{forStmt} \quad ::= \quad \text{`}\texttt{for}\text{'} \;\; \text{`}\texttt{(}\text{'} \;\; \textit{type} \;\; \textit{ID} \;\; \text{`}\texttt{in}\text{'} \;\; \textit{expr} \;\; \text{`}\texttt{)}\text{'} \;\; \textit{stmt}$$

SPADE's `for` loops are similar to "`for (`*type ID* `:` *expr*`)`" loops in Java 5, but use the Python-style `in` instead of the colon (`:`). SPADE does not have a C-style 3-part `for` loop.

A `while` statement looks just like in C or Java:

$$\textit{whileStmt} \quad ::= \quad \text{`}\texttt{while}\text{'} \;\; \text{`}\texttt{(}\text{'} \;\; \textit{expr} \;\; \text{`}\texttt{)}\text{'} \;\; \textit{stmt}$$

A `break` statement abruptly exists a `while` or `for` loop:

$$\textit{breakStmt} \quad ::= \quad \text{`}\texttt{break}\text{'} \;\; \text{`}\texttt{;}\text{'}$$

A `continue` statement abruptly jumps to the next iteration of a `while` or `for` loop:

continueStmt ::= '`continue`' '`;`'

A `return` statement abruptly exits a function, optionally returning a value:

returnStmt ::= '`return`' *expr*$^{?}$ '`;`'

To summarize, SPADE supports the following assortment of statements:

stmt ::= *localDecl* | *blockStmt* | *exprStmt*
| *ifStmt* | *forStmt* | *whileStmt*
| *breakStmt* | *continueStmt* | *returnStmt*

For SPADE 1 users: SPADE 1 supported a similar assortment of statements in custom logic in `Functor` operators.

Language design rationale: Strictly speaking, SPADE does not really need statements, because the user can put such logic in native (C++ or Java) code. We support statements anyway, because it is often easier to write SPADE statements than to go to a native language. Also, code written in pure SPADE is more portable across back-end languages, and offers more opportunities for front-end optimizations.

3.4 Functions

SPADE functions are similar to C functions: they can take parameters, return a value or return `void`, and are defined at the top level, nested in a namespace and in nothing else. Functions are called from expressions, which can occur in many places in a SPADE program.

Function definitions in SPADE look similar to those in C, for example:

```
int32 twice(int32 x) { return x + x; }
```

A function definition consists of a head and a body:

functionDef ::= *functionHead* *functionBody*
functionHead ::= *functionModifier** *type* *ID* '`(`' *functionFormal**, '`)`'
functionModifier ::= '`public`' | '`native`' | '`stateful`'
functionFormal ::= '`mutable`'$^{?}$ *type* *ID*
functionBody ::= '`;`' | *blockStmt*

The function head contains optional modifiers, the return type, identifier, and list of parameter definitions. A native function is a function implemented in C/C++ or Java. Native functions are denoted with the `native` modifier and have no body in SPADE code; instead, there are tools for generating skeleton code for their C/C++ or Java implementations. All non-native functions have a body. The caller of a function is oblivious to its implementation language (SPADE or native code).

Functions can be overloaded. In other words, there can be multiple functions with the same name in the same scope, as long as they have different parameter signatures. For example, there could be two function definitions for `print(int32)` and `print(string)`, such that `print(42)` would call the former and `print("the answer")` would call the latter. While SPADE permits overloading on parameter types, it forbids overloading on return value types. For example, declaring both `int32 f()` and `string f()` in the same scope is a compiler error.

By default, all parameters are deeply immutable. In other words, the function body can not modify the parameter itself or any part of it. For example:

```
void f(list<string> s) {                                    // 1
  s = ["hi", "there"]; //error: parameter s is not mutable // 2
```

```
  s[0] = "hello";      //error: parameter s is not mutable // 3
}                                                         // 4
```

The previous example would cause compiler errors. Parameters can only be modified if they are declared with the `mutable` modifier. The following example is legal:

```
void f(mutable list<string> s) {             // 1
  s = ["hi", "there"]; //legal: s is mutable // 2
  s[0] = "hello";      //legal: s is mutable // 3
}                                            // 4
```

A stateful function is a function that is non-deterministic or has side-effects. A function is non-deterministic if it does not consistently yield the same result each time it is called with the same inputs. A function has side-effects if it modifies state observable outside the function. For the purposes of this definition, "state observable outside the function" includes global variables in native code, files, the network, etc., but excludes mutable parameters. By default, functions are stateless unless annotated with the `stateful` modifier. (For language experts: functions that are stateless and have no mutable parameters are pure.)

All parameters (mutable or immutable, primitive or composite) are passed by-reference. In other words, the formal parameter in the callee is an alias of the actual parameter in the caller. Note that this is a marked difference from assignments, which have deep-copy semantics (Section 2.3). Function parameters have by-reference semantics (like "`T &v`" in C++), because copying large data into a function would be inefficient when operating only on a small part of that data. The following example illustrates Spade's parameter passing semantics:

```
void callee(mutable list<int32> x, mutable list<int32> y) { // 1
  x[0] = 1;                                                 // 2
  y = [3,4];                                                // 3
}                                                           // 4
void caller() {                                             // 5
  mutable list<int32> a = [0,0], b = [0,0,0];               // 6
  callee(a, b);                                             // 7
}                                                           // 8
```

The assignment `x[0]=1` in the callee yields `a[0]==1` in the caller. And the assignment `y=[3,4]` in the callee yields `b==[3,4]` in the caller. If the caller passes a computed value that does not have a storage location, then the callee stores it in a temporary, but modifications have no side effect on the caller. If you prefer by-value semantics for parameters, you can easily emulate them by copying the by-reference parameters into local variables.

Functions themselves can be passed as parameters to operators, because an operator invocation is fully resolved at compile time. But Spade does not permit passing functions to other functions, or storing them in variables, because it makes code harder to understand for the user, and it is harder to optimize indirect invocations in a static compiler. (For language experts: Spade provides neither first-class functions nor higher-order functions.)

Practical advice: Try to write and call stateless functions whenever possible. If you use mostly stateless functions, you are less likely to write buggy code, and the compiler is more likely to be able to optimize your code, for example, by parallelizing it, or by partially evaluating it at compile time.

Language design rationale: When designing Spade, we considered to categorically outlaw stateful functions. However, we found that this was hard to enforce for native functions, and that some stateful functions are useful, for example, functions that interact with external resources such as databases. Furthermore, stateful functions can yield better performance through memoization or when they make a small modification to a large data structure. Therefore, we decided to permit them in Spade, but the language design encourages mostly writing stateless functions. Unfortunately, it is impractical to statically check statelessness for native functions, so library vendors must be careful to declare native functions stateful when they read or write outside state.

Practical advice: Use the following guidelines to decide whether to write your functions in SPADE or as native (C/C++ or Java) code. Write a SPADE function if you want to avoid the burden of going to a different file or language, or if you want the portability of code that can be generated in either C/C++ or Java, or you want the compiler to check statelessness for you. Write a native function if you want to reuse existing native code, or if the native language permits a more natural implementation, or if the native function is significantly faster than one written in SPADE.

For SPADE 1 users: All functions in SPADE 1 were native.

4 Streams

The purpose of SPADE is to allow users to create streams of data, which are connected and manipulated by operator invocations. SPADE programs are designed to be deployed on distributed hardware for scaling [1]. The main goals of SPADE are scalability (exploiting distributed hardware), extensibility (encapsulating low-level code in high-level operators), and usability (easing the writing of scalable and extensible code).

A stream is an infinite sequence of tuples. As we saw in Section 2, a tuple is simply an instance of a tuple type, consisting of named attributes. Each stream is the result of an operator invocation. An operator invocation consumes zero or more input streams and defines zero or more output streams. Each time a tuple arrives on one of the input streams, the operator fires, and can produce tuples on output streams. An operator invocation in SPADE returns streams, analogously to how a function invocation in a traditional language returns values. However, given that a stream is an infinite sequence of tuples, each operator invocation is active for the duration of the program execution. This section shows how to define streams of tuples by invoking operators.

4.1 Operator Invocation Head

An operator invocation invokes an operator on input streams and defines output streams. Each operator invocation has a head and a body. The head lists the connected streams and the operator used to process these streams. The body elaborates on how the operator is to be invoked. The BNF syntax is:

$$opInvoke \quad ::= \quad opInvokeHead \quad opInvokeBody$$

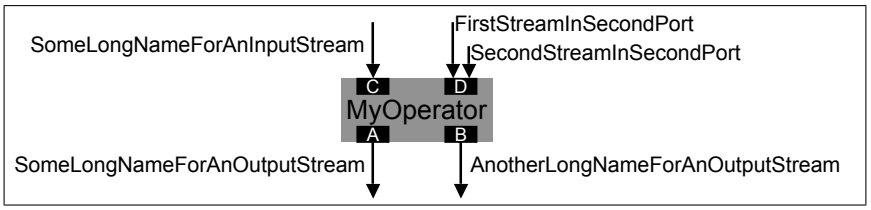

Figure 3: Graphical representation of the information in an operator invocation head.

The head of an operator invocation lists the output and input streams and the name of the operator that processes the data. Figure 3 shows a contrived operator invocation that illustrates multiple output and input streams as well as aliases. The SPADE code for this example is as follows:

```
( stream<int32 a1, int32 a2> SomeLongNameForAnOutputStream as A;           // 1
  stream<float64 b> AnotherLongNameForAnOutputStream as B                  // 2
) = MyOperator(                                                            // 3
  SomeLongNameForAnInputStream as C;                                       // 4
  stream<float64 i> FirstStreamInSecondPort, SecondStreamInSecondPort as D // 5
) {                                                                        // 6
```

```
    output A: a1=1, a2=max(n);                                          // 7
           B: b = C.i + D.i;                                            // 8
}                                                                       // 9
```

Lines 1 and 2 show the two output streams defined by this operator invocation, separated by a semicolon. Each has a type, a name, and an optional alias (`as A` and `as B`). The type lists the attributes of tuples on the stream; the name can be used as input to other operator invocations; and the alias can be used internally in this stream's body. Line 3 shows the name of the operator to invoke, in this case, `MyOperator`. Finally, Lines 4 and 5 show two input ports, separated by a semicolon. Each has an optional type, one or more stream names separated by commas, and an optional alias (`as C` and `as D`). Port `D` combines two input streams, `FirstStreamInSecondPort` and `SecondStreamInSecondPort`, separated by commas. The general syntax is:

opInvokeHead	::=	*opOutputs* '`=`' *ID* *opInputs*
opOutputs	::=	*opOutput* | '`(`' *opOutput*$^{*;}$ '`)`'
opOutput	::=	*streamType* *ID* ('`as`' *ID*)$^{?}$
opInputs	::=	'`(`' *portInputs*$^{*;}$ '`)`'
portInputs	::=	*streamType*$^{?}$ *ID*$^{+,}$ ('`as`' *ID*)$^{?}$

A port is a point at which streams connect to an operator invocation. Figure 3 shows ports as little black rectangles. Each output port produces exactly one output stream, but an input port can combine more than one input stream. When there is exactly one output port, the parentheses around the outputs can be omitted. When there are multiple streams on the same input port, tuples coming from different streams are interleaved. The order of the interleaving is non-deterministic in the sense that no particular order is imposed by the runtime. The as-alias notation assigns short names to ports; it is optional, and is useful to keep code readable in the presence of long stream names.

Output ports carry a mandatory stream type, and input ports carry an optional stream type. A stream type just consists of a tuple body:

streamType	::=	'`stream`' '`<`' *tupleBody* '`>`'

If there are multiple input streams on the same port, they must all have the same type. If the type of an input port is explicitly specified, it must be equivalent to the type of the stream(s) connecting to it. The *tupleBody* component of the stream type may be either a list of attributes, or a list of tuple types (extension), where identifiers can refer to types of other streams by their stream names. For details, see the description of tuple types in Section 2.2.2.

Practical advice: Instead of using the `as alias` notation, we encourage using short stream names in the first place. Stream names that are too long indicate that it might be time to decompose the program into composite operators, see Section 5.1.

Language design rationale: By putting just the connected streams and the operator name in the head, we make it possible for users to quickly scan an program's topology. A folding editor can even hide the stream bodies while the user looks at the stream heads. Types on input streams are redundant, because they also occur when those streams are defined, as outputs of other operator invocations. Nevertheless, we allow users to optionally specify these types when they feel that they make code more readable. This can also lead to more understandable compiler error messages when there is a type mismatch.

For SPADE 1 users: SPADE 1 solved the problem of long stream names by referring to streams positionally (`$1`, `$2`, etc.); SPADE 2 uses aliases instead. SPADE 1 also allowed additional information in stream heads, such as `import/export` modifiers; SPADE 2 replaces them by `Import/Export` operators (see Section 6.3).

4.2 Operator Invocation Body

The body of an operator invocation specifies how the operator is to be configured. SPADE supports a wide variety of operators with the default toolkits shipped with the compiler, and furthermore, developers can extend SPADE with new operators. Each operator can be configured and reused in different places in a data flow graph. To support this configuration, SPADE supports a versatile configuration syntax. All the configuration happens in the operator invocation body, to avoid tangling it with the data flow specification in the operator invocation head. The following example illustrates all possible operator invocation body clauses:

```
stream<string buyer, string seller, string item, int64 id> Sale        // 1
  = Join(Bid; Quote)                                                    // 2
{                                                                       // 3
  logic  state   : mutable uint64 n = 0;                                // 4
         Bid     : n++;                                                 // 5
  window Bid     : sliding, time(30);                                   // 6
         Quote   : sliding, count(50);                                  // 7
  param  match   : Bid.item == Quote.item && Bid.price >= Quote.price; // 8
  output Sale    : item = Bid.item, id = n;                             // 9
  pragma wrapper : "gdb";                                               //10
}                                                                       //11
```

An operator invocation body can have five clauses:

- The `logic` clause consists of local state that persists over the whole program execution, and statements that execute when a tuple arrives on an input port (see Section 4.3).
- The `window` clause specifies how many previously received tuples of each port to remember for later processing by stateful operators such as `Join`, `Sort`, or `Aggregate` (see Section 4.5).
- The `param` clause contains code snippets, such as expressions, supplied to the operator at invocation time; at runtime, the operator executes them whenever needed for its behavior (see Section 4.2.2).
- The `output` clause assigns values to attributes in output tuples each time the operator submits data to an output stream (see Section 4.2.1).
- The `pragma` clause gives non-functional requirements that influence how the compiler builds this operator invocation, or how the runtime system executes it (see Section 4.2.3).

Each clause is optional, and in fact, most operator invocations only use few clauses. The syntax is:

$$
\begin{array}{lcl}
\textit{opInvokeBody} & ::= & \text{`\{'} \\
 & & (\ \text{`}\texttt{logic}\text{'}\ \ \textit{opInvokeLogic}^{+}\ \ )^{?} \\
 & & (\ \text{`}\texttt{window}\text{'}\ \ \textit{nameValues}^{+}\ \ )^{?} \\
 & & (\ \text{`}\texttt{param}\text{'}\ \ \textit{opInvokeActual}^{+}\ \ )^{?} \\
 & & (\ \text{`}\texttt{output}\text{'}\ \ \textit{opInvokeOutput}^{+}\ \ )^{?} \\
 & & (\ \text{`}\texttt{pragma}\text{'}\ \ \textit{nameValues}^{+}\ \ )^{?} \\
 & & \text{`\}'} \\
\textit{opInvokeLogic} & ::= & \textit{ID}\ \ \text{`:'}\ \ \textit{stmt} \\
\textit{nameValues} & ::= & \textit{ID}\ \ \text{`:'}\ \ \textit{expr}^{+,}\ \ \text{`;'} \\
\textit{opInvokeActual} & ::= & \textit{ID}\ \ \text{`:'}\ \ \textit{opActualValue}\ \ \text{`;'} \\
\textit{opInvokeOutput} & ::= & \textit{ID}\ \ \text{`:'}\ \ (\ \textit{ID}\ \ \text{`='}\ \ \textit{expr}\ \ )^{+,}\ \ \text{`;'}
\end{array}
$$

Note that although the clauses do not have exactly the same syntax, most of them are special cases of comma-separated expression lists. Also note that the compiler will, in addition to these syntactic constraints,

enforce semantic constraints on each clause. For example, only the windows described in Section 4.5 are legal in the `window` clause.

Language design rationale: SPADE separates the body of operator invocations into clauses to make them easy to read. Clause order follows execution order: when a tuple arrives, logic executes first, followed by storing history in windows, executing parameter expressions, and finally assigning output attributes when the operator submits a tuple. Since the user rarely needs to see pragmas to understand what the code does, they have their own clause at the bottom of the operator invocation body. An IDE can facilitate writing structured operator invocation bodies, and can facilitate reading them by selectively folding clauses out of sight.

For SPADE 1 users: In SPADE 1, logic was written in angle brackets (`<...>`) after a Functor stream head. Windows were specified between angle brackets as part of the input stream list. Parameters were written between square brackets (`[...]`), and, depending on whether the operator was built-in or user-defined, could be named or unnamed. Output assignments were written between curly braces (`{...}`) after the parameter section. Pragmas were written after an arrow (`->...`) at the end of an operator invocation. Here is the operator invocation defining stream `Sale` in SPADE 1:

```
stream Sale(buyer: String, seller: String, item: String) := Join      # 1
    #* omitted: Spade 1 only allows logic in Functors *# # logic      # 2
    ( Bid <time(30)>; Quote <count(50)> )                # windows    # 3
    [ $1.item = $2.item & $1.price >= $2.price ]         # parameters # 4
    { item := $1.item }                                  # outputs    # 5
  -> wrapper = gdb                                       # pragmas    # 6
```

4.2.1 Output Clause

The `output` clause assigns values to attributes in output tuples each time the operator submits data to an output stream. For example, "`output Sale: item = Bid.item, id = n;`" assigns values to attributes `item` and `id` in output stream `Sale`. The label, such as `Sale`, is an output stream name or alias, and opens the scope of that stream similarly to a Pascal "with" statement. The right-hand side is a comma-separated list of assignments to attributes.

For most operators, any attribute in the output stream that is not assigned by the output clause is implicitly copied from an attribute with the same name defined on an input stream. If there is not exactly one input attribute to copy, the compiler flags an error. Certain operators may have different forwarding behavior, specified by their operator model (operator models are described in Section 5.5.2).

Practical advice: Avoid calling `stateful` functions in the `output` clause.

4.2.2 Stream Parameters

The `param` clause contains code snippets supplied to the operator at invocation time. For example, the clause "`param match: Bid.item == Quote.item && Bid.price >= Quote.price;`" supplies a boolean expression as the `match` parameter to an operator, such as `Join`. At runtime, the operator executes the expression whenever needed for its behavior. For each time the operator fires, the expression might execute zero, one, or multiple times depending on the operator; but if it executes, it executes after the logic clause and before the output clause. Expressions are just one of a total of five kinds of parameter: (i) a comma-separated list of stream attribute names, (ii) a comma-separated list of expressions, (iii) a function name, (iv) an operator name, or (v) a type. The documentation of an operator specifies which parameters it expects, and the compiler checks that the right kind of entity is passed as a parameter. Primitive operators can only accept parameters of the forms (i) and (ii), composite operators can also accept parameters of the forms (iii), (iv), and (v).

4.2.3 Stream Pragmas

The `pragma` clause gives non-functional requirements that influence how the compiler builds this operator invocation, or how the runtime system executes it. Pragmas include deployment directives, debugging support, or high-availability and fault-tolerance configurations. For example, the `pragma wrapper: "gdb"` directs the runtime to invoke the operator through a wrapper application, in this case, `gdb` (the GNU debugger). Wrappers can specify the path to any application that launches another application, such as `valgrind`, a popular memory safety checker. The IDE and tools for SPADE will also provide external ways to specify pragmas without editing the source code. In that case, pragmas embedded in the source code specify a default, and external pragmas override it.

Practical advice: Embedded pragmas are useful during testing and development, but if you want your code to be reused on different infrastructure, avoid hard-wiring anything that is specific to your local infrastructure.

Language design rationale: We considered disallowing embedded pragmas altogether, but decided against that, because they were popular among SPADE 1 users.

4.3 Stream Logic

The `logic` clause consists of local operator state that persists over the whole program execution, and statements that execute when a tuple arrives on an input port. For example:

```
stream<float64 runningAvg> FStr = Functor(IStr) {             // 1
  logic  state : { mutable float64 avg = 0.0, alpha = 0.1;    // 2
                   mutable int64 no = 0; }                    // 3
         IStr  : { //code executed when tuple arrives on IStr // 4
                   if (no == 0l) {                           // 5
                     avg = sample; //that's IStr.sample      // 6
                   } else {                                  // 7
                     float64 beta = 1.0 - alpha;             // 8
                     avg = alpha * sample + beta * avg;      // 9
                   }                                         //10
                   no++;                                     //11
                 }                                           //12
  param  filter: true;                                       //13
  output FStr  : runningAvg = avg;                           //14
}                                                            //15
```

The `state:` label introduces variable declarations. In this case, there is only one input port `IStr`, and the `IStr:` label introduces code to be executed each time a tuple arrives on that stream. It opens up the scope of the input stream, so attributes can be accessed without qualifiers. SPADE supports custom logic on each port and the `state` variables are shared among them. The code itself is typically a block, but can be any SPADE statement, as described in Section 3.3. Access to operator invocation state is implicitly synchronized. The SPADE compiler inserts the necessary locks to ensure that no two threads experience race conditions from concurrent access to state variables in the stream logic clause.

For SPADE 1 users: SPADE 1 supports custom logic only in invocations of the `Functor` operator; SPADE 2 supports it on invocations of all operators. Here is the SPADE 1 version of the example:

```
stream FStr(runningAvg: Double) := Functor(IStr)               # 1
  < Double $avg := 0d; Double $alpha := 0.1d; Long $no := 0l; > # 2
  < #code executed when tuple arrives on IStr                  # 3
    if($no = 0l) {                                             # 4
      $avg := sample; # that's IStr.sample                     # 5
    } else {                                                   # 6
```

```
    Double $beta := 1.0d - $alpha;                                  # 7
    $avg := $alpha * sample + $beta * $avg;                         # 8
  }                                                                 # 9
  $no := $no + 1;                                                   #10
>                                                                   #11
[ true ] { runningAvg := $avg }                                     #12
```

The first angle-bracket delimited section (<...>) in SPADE 1 corresponds to the `state:` section in SPADE 2, and the second angle-bracket delimited section in SPADE 1 corresponds to the *portName*: section in SPADE 2. SPADE 2 supports multiple logic sections in the same operator, one per port, which can all refer to the same state, e.g., to count tuples from either port.

4.4 Punctuation

A punctuation is a control signal that appears interleaved with the tuples in a stream. SPADE supports two kinds of streams: a punctuation-free stream is an infinite sequence of tuples, whereas a punctuated stream is an infinite sequence of tuples interleaved with punctuations. A punctuation separates groups of consecutive tuples on a stream to create window boundaries. For example, consider an invocation of an `Aggregate` operator with a punctuation-based window. Each time this operator invocation receives a punctuation, it aggregates the accumulated tuples since the last punctuation.

An output port of an operator invocation can either generate, remove, or preserve punctuation markers. Examples for punctuation-generating ports include the output from the `Join`, `Aggregate`, and `Punctor` operators. Punctuation-free ports guarantee that their output stream is not punctuated, and punctuation-preserving ports will forward punctuations, but only if there is a unique punctuated input stream.

An input port of an operator invocation can either be punctuation-oblivious or punctuation-expecting. Ports are oblivious to punctuation if there is no window or any window other than punctuation-based windows. Ports with punctuation-based windows expect that the input stream be punctuated. Punctuation-expecting ports must be connected to exactly one input stream. The compiler enforces these rules of punctuation by reporting errors when they are violated.

4.5 Windows

A window is a logical container for tuples recently received by an input port of an operator. Some relational operators rely on windows for their operation; for example, in `Join(A;B)`, when a tuple arrives on port `A`, SPADE updates the window on port `A` and matches the new tuple against each tuple in the window on port `B`. The following example illustrates different kinds of windows on different input ports; see later subsections for an exhaustive description.

```
stream<float32 c> A = MyOperator(C; X, Y as D) { // 1
  window C : sliding, time(10), count(5);        // 2
         D : tumbling, delta(i, 10);             // 3
}                                                // 4
```

Port `C` uses a *time*-based *sliding* window over the last 10 seconds, which triggers the operator's logic after every 5 tuples. Port `D`, which interleaves tuples from streams `X` and `Y`, uses an *attribute-delta* based *tumbling* window, where attribute `i` in the oldest and newest tuple differs by no more than 10. Attribute names are scoped by the port label in the window clause: for example, `i` corresponds to `D.i`. All expressions in windows must be compile-time constants. The following subsections describe available windowing mechanisms in detail.

For SPADE 1 users: In SPADE 1, the example would look like this:

```
stream A(c : Float)                                              # 1
  := MyOperator(C <time(10), count(5)>; X, Y <attrib(i, 10)>) # 2
     [] {}                                                       # 3
```

SPADE 1 keeps window specifications in the operator invocation head next to the input streams, SPADE 2 keeps them in the body. Another difference is that SPADE 1 window support was not uniform over all operators.

4.5.1 Tumbling Windows

A tumbling window stores incoming tuples until the window is full, then executes the operator logic, and finally flushes all stored tuples, before starting over from scratch. A tumbling window is specified by providing just an expiration policy, in other words, the window size, which determines when the window is "full". SPADE offers four expiration policies:

- `count(`n`)`: Count-based window, storing n tuples.
- `time(`n`)`: Time-based window, storing tuples arriving over a period of n seconds.
- `delta(`a`, `n`)`: Attribute-delta based window, storing tuples until the difference between attribute a of the oldest and the newest tuple exceeds n. The attribute is typically monotonically increasing, and is often a `timestamp`. That way, an attribute-delta based window can emulate a time-based window, but instead of using current time on the local machine, it uses a time attribute streamed along with the data.
- `punct()`: Punctuation-based window, storing tuples until the next punctuation arrives. An upstream operator can generate punctuations, and a downstream operator can use them for windowing.

4.5.2 Sliding Windows

A sliding window is specified by providing both an expiration policy and a trigger mechanism. For example, `sliding, time(10), count(5)` means the expiration policy keeps the window size to 10 seconds, and the trigger mechanism executes the operator logic after every 5 tuples. A sliding window expels the oldest tuples to maintain the window size when new tuples are coming in, and it executes the operator logic each time it reaches a trigger. Unlike for tumbling windows, the expiration policy and trigger mechanism for sliding windows are independent from each other, so an old tuple can still be in the window if the logic of the operator gets triggered again before the tuple expires. The available expiration policies and trigger mechanisms are count-based, time-based, and attribute-delta based, but not punctuation-based. In other words, the window can use one of three expiration policies and one of three trigger mechanism, yielding a total of 9 combinations. If the trigger mechanism is omitted, it defaults to `count(1)`; for example, "`sliding, count(3)`" is equivalent to "`sliding, count(3), count(1)`".

4.5.3 Grouping

The operator parameter `param perGroup: true` specifies that instead of maintaining just one window per port, the operator invocation maintains one window per port per group. A group is a set of tuples that have the same values for grouping attributes. For example, with the parameter `param groupBy: x, y`, tuples belong to different groups if they differ in either attribute `x` or `y`. When a new tuple arrives on a port, it is routed to the window for its own group. Each group window uses an expiration policy and trigger mechanism as described above for tumbling and sliding windows. Grouping is frequently used in conjunction with aggregation; Section 5.3 gives a concrete example. The `perGroup` and `groupBy` parameters are not a language feature. They are operator-specific, but by convention, most operators implement them to follow the guidelines described here.

4.5.4 History

Expressions in parameters or output assignments of operator invocations can refer directly to tuples received in the past. History access conceptually treats a window as a list of tuples, which can be subscripted by using the input port name. For example, in

```
stream<int32 id, int32 diff, string name> Out = Functor(In) { // 1
  window In     : sliding, count(2);                          // 2
  param  filter : name != In[1].name;                         // 3
  output Out    : diff = id - In[1].id;                       // 4
}                                                             // 5
```

the expression `In[1].name` refers to a historical tuple on port `In`, `1` tuple in the past, and extracts the `name` attribute from it. The current tuple is at index 0, so the above filter parameter could also have been written as `In[0].name != In[1].name`. History access also supports slicing. For example, `In[0:]` would yield a list of all tuples currently in the window, in this case, `[In[0],In[1]]`.

For SPADE 1 users: SPADE 1 used the hat (`^`) notation for history access, and did not support window slicing. Here is the SPADE 1 version of the example:

```
stream Out(id: Integer, diff: Integer, name: String) # 1
  := Functor(In)                                     # 2
     [ name != ^1.name ]                             # 3
     { diff := id - ^1.id }                          # 4
```

4.6 Error Semantics

Even though SPADE has been designed to perform most of its error checking at compile time, runtime errors can still happen in some cases. This includes an invalid subscript for a list, matrix, or map; a size mismatch in the operands of mapped expression operators; division by zero; or exceptions in libraries (such as C++ or Java exceptions).

When any of these occur, the entire execution container (e.g., a processing element in the InfoSphere Streams implementation) enclosing the operator invocation dies, writes the error to a log file, and stops accepting new tuples. The SPADE distribution may come with additional tools and options, such as a debugging tool, or a "resilient" option where the execution container just logs the error and ignores one tuple, but then continues executing.

To help the user with error handling, SPADE provides APIs for assertions and logging. Assertions look like calls to an `assert` function with the following signature:

```
stateful native void assert(boolean condition, string message); // 1
stateful native void assert(boolean condition);                 // 2
```

They are marked stateful, since logging or halting is a side effect. However, unlike regular function calls, and like assertions in C or Java, SPADE assertions can be disabled. When a SPADE program is compiled with assertions disabled, their parameter expressions are not evaluated, saving time and possibly preventing side effects.

Besides assertions, SPADE also provides two other features to help testing and error handling. One is a log function, with the following signature:

```
stateful native void log(DebugLevel level, string message, string aspect);
```

The `level` belongs to a type defined in the SPADE standard library:

```
type DebugLevel = enum { error, info, debug, trace };
```

The values `error` means that the log is unmaskable, whereas the values `info`, `debug`, and `trace` specify that logging should be performed when the user requested least verbose, more verbose, or most verbose logging, respectively.

Besides assertions and logging, SPADE also provides an `abort()` function, which unconditionally terminates the application, even when compiled with assertions disabled.

Language design rationale: We considered adding full-fledged exception handling to SPADE, with `try`/`catch`/`finally` statements. But we decided against that because if the user anticipates the exception, it is easy to check with if-statements, whereas if the exception is unanticipated, the logging functionality that SPADE provides is probably more helpful than exception handlers a user could write.

5 Operators

A stream operator can be invoked to transform input streams to output streams. SPADE supports two kinds of operators: primitive operators and composite operators. A composite operator contains a reusable stream subgraph. A SPADE program can be viewed as a hierarchy of operator invocations, where the leaves are primitive operators, each level groups graphs of operators into composite operators, and the root is a main operator to be deployed as a job on the streaming middleware. All operator invocations follow the syntax described in Section 4, irrespective of whether they invoke primitive or composite operators.

This section is about how to define new operators that can then be invoked to define streams. Most streaming languages are tailored towards a single application domain. For example, StreamSQL targets the stream-relational domain [2]. SPADE, on the other hand, is designed to address a diverse set of domains. Key to this diversity is that SPADE users can define their own operators, either in SPADE itself or by reusing legacy code written in C++ or Java.

For SPADE 1 users: SPADE 1 only supported primitive operators, no composite operators.

5.1 Composite Operators

A composite operator contains a reusable stream subgraph, which can itself be invoked to define streams. Each composite operator definition has a head and a body. The head lists input and output ports, and the body specifies the graph and its parameters. The BNF syntax is:

compositeDef ::= *compositeHead* *compositeBody*

The head of a composite operator definition names the operator and lists its ports. For example:

```
composite M (output K, L; input G, H) {/*...*/}
```

The composite operator `M` has two output ports, `K` and `L`, and two input ports, `G` and `H`. The syntax is:

compositeHead ::= '`composite`' *ID* ('`(`' *compositeInOut*$^{+;}$ '`)`')$^{?}$
compositeInOut ::= ('`input`' | '`output`') (*streamType*$^{?}$ *ID*)$^{+,}$
streamType ::= '`stream`' '`<`' *tupleBody* '`>`'

The body of a composite operator definition specifies what kind of parameters it accepts, and contains a stream graph. Besides these two clauses (`param` and `graph`), the body can also contain a `type` clause with type definitions, a `var` clause with shared variable definitions, and a `pragma` clause with nonfunctional requirements. All five clauses are optional. Here is the syntax:

compositeBody ::= '{'
('`param`' *compositeFormal*$^{+}$)$^{?}$ # Section 5.2
('`type`' *typeDef*$^{+}$)$^{?}$ # Section 5.1.2
('`var`' *sharedVarDef*$^{+}$)$^{?}$ # Section 5.4
('`graph`' *opInvoke*$^{+}$)$^{?}$ # Section 5.1.1
('`pragma`' *nameValues*$^{+}$)$^{?}$ # Section 5.1.3
'}'

Practical advice: We encourage you to use composite operators whenever you write a large program, because it becomes easier to understand when you break it down into smaller subgraphs. We also encourage you to make your composite operators reusable, and put them into libraries, so they can be invoked in different programs. Finally, you can use parameterizable composite operators to implement distributed computing patterns such as map-reduce.

Language design rationale: Besides the obvious structuring and reuse advantages, composite operators help visualizing both static and dynamic views of programs. And composite operators serve as a foundation for higher-level stream programming, for example, with ontology-based planning. The syntax for composite operators separates the head from the body, to make it easy to see port connections, and splits the body into multiple clauses, to make it easier to read and enable folding editors.

5.1.1 Graph Clause

The graph clause of a composite operator describes a stream data flow subgraph, which can then be expanded in different contexts when the operator is invoked. Here is an example:

```
composite M (output K, L; input G, H) {                  // 1
  graph stream<int32 x> I = O(G)    { }                  // 2
        stream<int32 x> J = P(H)    { }                  // 3
        stream<int32 x> K = Q(I; J) { }                  // 4
        stream<int32 x> L = R(J)    { }                  // 5
}                                                        // 6
(stream<int32 x> C; stream<int32 x> D) = M(A; B) { } // 7
(stream<int32 x> E; stream<int32 x> F) = M(A; B) { } // 8
```

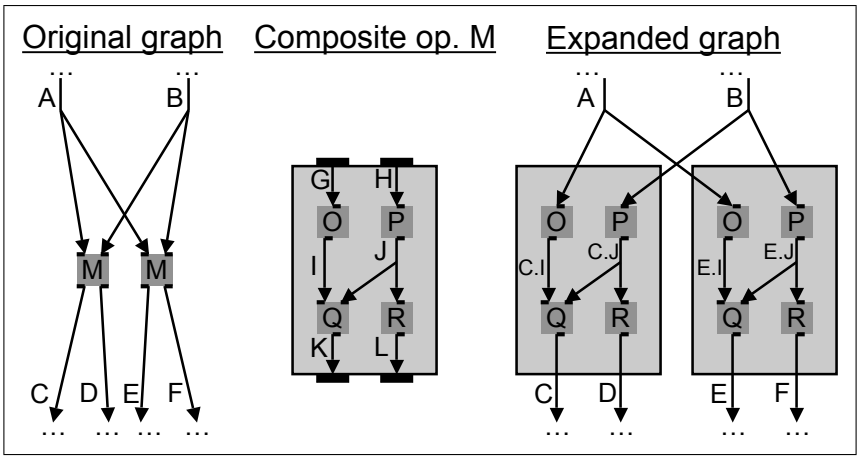

Figure 4: Composite operator expansion example.

Figure 4 illustrates the graph topology in this example. The original graph contains two invocations using composite operator `M`: once to transform `A` and `B` into `C` and `D`, and the other time to transform `A` and `B` into `E` and `F`. The head of the composite operator definition `(output K, L; input G, H)` defines input and output ports. When the composite operator gets used, actual input and output streams are bound to these ports; for example, for the first invocation, the bindings are `G=A`, `H=B`, `K=C`, and `L=D`. In the expanded graph, intermediate streams such as `I` and `J` in the example are duplicated and renamed `C.I`, `C.J`, `E.I`, and `E.J`. (For language experts: The compile-time expansion of composite operators behaves similarly to macros, but is "hygienic", since it generates new names to prevent unintended name capture.) The qualified names, such as `C.I` and `C.J`, use the names of output streams as qualifiers, and thus, follow the direction of expansion ("outside-in"). When there are multiple output streams, the names are not unique: in this example, since

both `C` and `D` are outputs of the same operator invocation, `C.I` refers to the same stream as `D.I`, and `C.J` refers to the same stream as `D.J`.

5.1.2 Type Clause

A type definition gives a name to a type. For example:

```
composite Comp ( /*...*/) {                                          // 1
  type  IntegerList = list<int32>;                                   // 2
        MySchema = string x, IntegerList y;                          // 3
  graph /* ... */                                                    // 4
        stream<MySchema, tuple<int32 z>> SenSource = FileSource() { // 5
          param  fileName: "SenSource.dat"; format: csv, noDelays;   // 6
          pragma host    : SourcePool(0);                            // 7
        }                                                            // 8
}                                                                    // 9
```

The type `MySchema` is defined as a tuple, leaving out the optional constructor `tuple<...>`. The type can be used in operator invocations. In this case, it is used in "`MySchema, tuple<int32 z>`", which extends `MySchema` by adding another attribute "`int32 z`". Tuples on stream `SenSource` have the type `tuple<string x, list<int32> y, int32 z>`, in other words, all attributes of `MySchema`, plus the additional `z` attribute. SPADE also allows using the name of a stream to refer to its type. For example, if there is a stream `A`, then `stream<A> B` declares a stream `B` with the same type as stream `A`, and `stream<A, tuple<int32 i>> C` declares a stream `C` with tuples that extends `A` by an extra attribute `i`.

The syntax of a type definition is:

typeDef ::= '`public`'? *ID* '`=`' (*type* | *tupleBody*) '`;`'

Public type definitions (with the `public` modifier) may be used from anywhere in the program, whereas private type definitions (without `public` modifier) are only visible in the same composite operator body that defines them. Public type definitions must not depend on any operator parameters. To access a type from a different composite operator, qualify it with its operator, e.g., `my.nameSpace::MyOperator.MyType`.

For SPADE 1 users: Here is the same example in SPADE 1:

```
[Typedefs]                                                  # 1
typedef Integers IntegerList                                # 2
[Program]                                                   # 3
vstream MySchema(                                           # 4
  sx : String,  dx : Double,  fx : Float,  lx : Long,       # 5
  ix : Integer, iy : Integer, bx : Boolean )                # 6
stream SenSource(schemaFor(MySchema), id: Integer)          # 7
  := Source() ["file:///SenSource.dat", nodelays, csvformat]{} # 8
  -> node(SourcePool, 0)                                    # 9
```

SPADE 1 supported two different kinds of type definitions. Tuple types were defined in the `[Program]` clause with the `vstream` keyword, whereas all other types were defined in the `[Typedefs]` clause with the `typedef` keyword. SPADE 2 unifies the two. Also, in SPADE 1, you can use the `schemaFor` function to get the type of a stream, whereas in SPADE 2, you can simply use the name of a stream to stand for its type.

5.1.3 Composite Operator Pragmas

In SPADE, a pragma specifies a non-functional requirement (deployment directive, debugging support, high-availability and fault tolerance configurations). As we saw in Section 4.2.3, pragmas can be specified em-

bedded in the source code, or externally at the IDE level. This section describes the `pragma` clause of a composite operator, which specifies pragmas for all operator invocations in its graph at once. For example:

```
type T = int32 i;                       // 1
stream<T> S1 = Op1(S0) { }              // 2
composite Comp(output Out; input In) { // 3
  graph  stream<T> S2  = Op2(In)  { }  // 4
         stream<T> S3  = Op3(S2) { }   // 5
         stream<T> Out = Op4(S3) { }   // 6
  pragma wrapper : "gdb";               // 7
}                                       // 8
stream<T> S4 = Comp(S1) { }             // 9
stream<T> S5 = Op5(S4) { }              //10
```

The `pragma wrapper: "gdb"` directs the runtime to invoke all operators used in the graph `Comp` using `gdb` (the GNU debugger) as a wrapper application.

(For SPADE 1 users: SPADE 1 specifies pragmas after `->` in operator invocations. Since SPADE 1 has no composite operators, you have to specify the pragma repeatedly on each individual operator invocation:

```
vstream T(i : Integer)                                        # 1
stream S1(schemaFor(T)) := Op1(S0) [ ] { }                    # 2
stream S2(schemaFor(T)) := Op2(S1) [ ] { } -> wrapper=gdb # 3
stream S3(schemaFor(T)) := Op3(S2) [ ] { } -> wrapper=gdb # 4
stream S4(schemaFor(T)) := Op4(S3) [ ] { } -> wrapper=gdb # 5
stream S5(schemaFor(T)) := Op5(S4) [ ] { }                    # 6
```

Specifying pragmas separately on each stream is still allowed in SPADE 2, but doing it at the composite operator level can prevent repetitive code.)

For high availability, SPADE supports replicating a part of the stream graph; if the master replica fails, then one of the healthy backup replicas takes over. Users specify availability requirements with composite operator pragmas. For example:

```
composite SymbolHist(output Out; input In) {                                  // 1
  param expression<string> $sym;                                              // 2
  graph stream<In> S = Functor(In) {                                          // 3
          param filter : symbol==$sym && ttype=="Trade";                      // 4
        }                                                                     // 5
        stream<float32 lastPrice, float32 minPrice> Out = Aggregate(S) {  // 6
          window S          : sliding, time(2*60), count(1);                  // 7
          output Out        : lastPrice=Last(price), minPrice=Min(price); // 8
          pragma checkpoint : 10;                                           // 9
        }                                                                     //10
}                                                                             //11
stream<float32 lastPrice, float32 minPrice> HPQ_HIST = SymbolHist(NYSE) { //12
  param  sym        : "HPQ";                                                  //13
  pragma haReplicas : 2;                                                      //14
}                                                                             //15
```

In the example, `pragma checkpoint: 10` states that the internal operator state should be saved every 10 seconds to provide a safe-point for restart, and pragma `haReplicas: 2` creates two redundant high-availability ("`ha`") replicas of the composite operator, one of which will serve as master and the other as backup. In the example, the `haReplicas` pragma could have also been part of the composite operator

definition instead of its invocation. When there are pragmas both on the definition and the invocation of a composite operator, the invocation pragma overrides the definition pragma.

Practical advice: While this section illustrates the language feature pragmas with some examples, these examples are not themselves part of the language specification. Pragmas, along with other runtime behaviors, will be defined in a separate forthcoming document. This might also be a good place to mention that load shedding is not part of the language specification. If your application needs to shed load, we recommend you write filtering operators for that.

For SPADE 1 users: SPADE 1 had a special syntax for defining subgraphs to use as high-availability ("`ha`") sections, instead of using composite operators with pragmas:

```
ha_begin 2                                          # 1
  stream HPQ (price: Float)                         # 2
    := Functor(NYSE)                                # 3
       [symbol="HPQ" & ttype = "Trade"] { }         # 4
  stream HPQ_HIST(lastPrice: Float, minPrice: Float) # 5
    := Aggregate(HPQ <time(2*60), count(1)>)        # 6
       [] { Last(price), Min(price) }               # 7
    -> checkpoint=10                                # 8
ha_end                                              # 9
```

5.2 Operator Parameters

Different invocations of the same operator can yield different expansions, by providing actual parameter values for formal parameter placeholders declared in the operator definition.

5.2.1 Operators as Parameters to Composite Operators

A composite operator encapsulates a subgraph and some of the operators in that subgraph can be given to the composite operator as parameters at the time of invocation. Thus, composite operators can act as generic graph stencils, which are useful for implementing high-level distributed well-structured computation paradigms such as map-reduce [3]. The following code is a parametric version of the composite operator `M` defined earlier:

```
composite M (output K, L; input G, H) {                          // 1
  param operator $Q; //operator (primitive or composite)         // 2
  graph stream<int32 x> I =  O(G)    { }                         // 3
        stream<int32 x> J =  P(H)    { }                         // 4
        stream<int32 x> K = $Q(I; J) { }                         // 5
        stream<int32 x> L =  R(J)    { }                         // 6
}                                                                // 7
(stream<int32 x> C; stream<int32 x> D) = M(A; B) { param Q: S; } // 8
(stream<int32 x> E; stream<int32 x> F) = M(A; B) { param Q: T; } // 9
```

Figure 5 illustrates how the composite operator and its parameter get expanded. The first expansion, on the left, replaces the placeholder `parameter $Q` by the actual operator `S`. The second expansion, on the right, replaces the placeholder `parameter $Q` by the actual operator `T`.

5.2.2 Other Operator Parameters

Both composite and primitive operators can be configured by parameters such as the name of an attribute in a connected stream, or an expression to be used to filter data. The following example illustrates this for a composite operator `Mogrifier`:

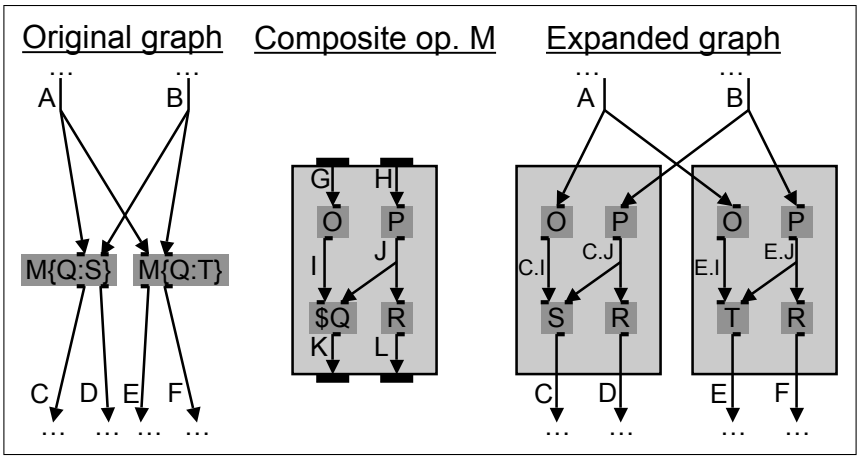

Figure 5: Composite operator parameter example.

```
composite Mogrifier(output Out; input In) {       // 1
  param attribute  $attr;                         // 2
        expression $expr;                         // 3
  graph stream<In> Out = Functor(In) {            // 4
          param  filter: $attr < $expr;           // 5
          output Out   : $attr = $attr + 1;       // 6
        }                                         // 7
}                                                 // 8
stream<int32 i, int32 j> A = Source() { /*...*/ } // 9
stream<int32 i, int32 j> B = Mogrifier(A) {       //10
  param attr: i;                                  //11
        expr: 2 * j;                              //12
}                                                 //13
```

Binding the actual attribute `i` to placeholder `$attr` and the actual expression `2*j` to placeholder `$expr` yields the following expansion:

```
stream<int32 i, int32 j> A = Source() { /*...*/ } // 1
stream<int32 i, int32 j> B = Functor(A) {         // 2
  param  filter: i < 2 * j;                       // 3
  output B     : i = i + 1;                       // 4
}                                                 // 5
```

5.2.3 Operator Parameter Passing Semantics

Operator parameters in SPADE behave similarly to macro expansion, but the semantics are designed to prevent macro mistakes common in languages like C. In particular, SPADE defines a name resolution rule to prevent accidental name capture and a syntactic confinement rule to prevent unintended re-association.

The **name resolution rule** for names in actual parameters is that the SPADE compiler resolves them in the context of the operator invocation, not in the context of the operator definition. As we will see in Section 5.4, you can define variables in the `var` clause of a composite operator. The following example illustrates how SPADE resolves names when there are multiple choices:

```
composite Comp1(output O1; input I1) {                    // 1
  var   int32 s_t = 1;                                    // 2
```

```
  graph stream<int32 x> O1 = Comp2(stream<int32 x> I1) { param cond: x < s_t; } // 3
}                                                                               // 4
composite Comp2(output O2; input I2) {                                          // 5
  param expression $cond;                                                       // 6
  var   int32 s_t = 2;                                                          // 7
  graph stream<I2> O2 = Functor(stream<int32 x> I2) { param filter: $cond; }    // 8
}                                                                               // 9
```

The actual value for parameter `cond` in Line 3 is `x < s_t`, which contains two names `x` and `s_t`. The names are resolved in the context of the operator invocation in Line 3, where `x` is `I1.x` and `s_t` is `Comp1.s_t`, which is `1`. If the names were resolved in the context of the operator definition, then, since `$cond` is used in Line 8, `x` would be `I2.x` and `s_t` would be `Comp2.s_t`, which is `2`. This would make no difference for `x`, because `I1.x` and `I2.x` are the same, but it would make a difference for `s_t`, because `Comp1.s_t` and `Comp2.s_t` differ. By resolving `s_t` to `Comp1.s_t`, the language shields the author of `Comp1` from implementation details of `Comp2`. The rule "resolve names in the invocation, not the definition" supports encapsulation, because it hides implementation details of the operator, rather than allowing them to leak out by capturing names. (For language experts: SPADE operator parameters have "call-by-name" semantics. But note that they are used with partial evaluation: everything is expanded at build time. If the expanded expression only uses build-time values, it is fully evaluated at build time. On the other hand, if the expanded expression also involves runtime values, only the build-time subexpressions are partially evaluated, and the rest is evaluated at runtime.)

The **syntactic confinement rule** for SPADE operator parameters is that argument expressions are implicitly parenthesized. The implicit parentheses "confine" them from associating in unexpected ways when they get used. A corollary of this rule is that you cannot pass something that does not start out confined; for example, an incomplete expression like "`x +`" is not valid as an actual parameter to an operator. The following example illustrates the syntactic confinement rule:

```
composite M(output Out; input In) {                    // 1
  param expression<int32> $q;                          // 2
  graph stream<float32 percent> Out = Functor(In) {    // 3
          output Out: percent = float32(100 * $q);     // 4
        }                                              // 5
}                                                      // 6
stream<float32 x> A = Source() { /*...*/ }             // 7
stream<float32 percent> B = M(A) { param q: x - 1; } // 8
```

The actual parameter `x - 1` on Line 8 is substituted for the parameter `$q` on Line 4. Because of the syntactic confinement rule, `100 * $q` associates like `100 * (x - 1)`, not like `(100 * x) - 1`. As with the name resolution rule, the syntactic confinement rule leads to better encapsulation: when invoking a operator, you need not worry about precedence context in the operator definition. This prevents unintended results. (For language experts: Parameter expressions yield a subtree in the AST and will not spill out of this subtree during expansion.)

Language design rationale: We chose this approach for operator customization, because it makes uses of operators concise, provides flexible typing, and naturally generalizes to native operators. It also simplifies the implementation, since certain entities (operator, type, and function parameters) are fully resolved at compile time, and need not be treated as first-class values at runtime.

5.2.4 Operator Parameter Types and Default Values

A parameter declaration in a composite operator definition must specify a meta-type, and can optionally specify a default value. For example:

```
composite ThresholdFilter(output Filtered; input Unfiltered) { // 1
```

```
  param expression<float32> $threshold : 5.0;                    // 2
  graph stream<float32 x> Filtered = Functor(Unfiltered) {       // 3
          param filter : x > $threshold;                         // 4
        }                                                        // 5
}                                                                // 6
```

The declared type of parameter `$threshold` is `expression<float32>`. This type serves as documentation and helps produce better error message. For example, if the composite operator is used with the wrong type:

```
stream<float32 x> B = ThresholdFilter(A) { param threshold: "abc"; }
```

then the SPADE compiler gives the error message:

```
type mismatch: $threshold expects a float32, but "abc" is a string
```

Besides the meta-type `expression<`T`>`, where T is a concrete runtime type, a parameter type can also be just `expression` without a type parameter, indicating that an expression of any type works as long as the expansion type-checks. In addition to `expression`, parameter type declarations can also use other meta-types:

$$
\begin{array}{lcl}
\textit{metaType} & ::= & \text{`}\texttt{attribute}\text{'} \\
 & | & \text{`}\texttt{expression}\text{'}\ \ \textit{typeArgs}^{?} \\
 & | & \text{`}\texttt{function}\text{'}\ \ \textit{typeArgs}^{?} \\
 & | & \text{`}\texttt{operator}\text{'} \\
 & | & \text{`}\texttt{type}\text{'} \\
\textit{typeArgs} & ::= & \text{`}\texttt{<}\text{'}\ \ \textit{type}^{+,}\ \ \text{`}\texttt{>}\text{'}
\end{array}
$$

For example, `attribute` is the name of one or more attributes in an input stream, and `operator` is a primitive or composite operator. The meta-type `function` also has optional type parameters. For example, `function<void,int32,int32>` accepts any function returning `void` with two formals of type `int32`.

Still using the same example, the declared default value of parameter `$threshold` is `5.0`. This default value makes it easier to use the composite operator, because the user can omit the parameter. For example, the following invocation is short yet complete:

```
stream<float32 x> B = ThresholdFilter(A) { }
```

This is equivalent to providing the default. Besides making operators more reusable, default values also serve as documentation, because they give an example for the kind of value expected by a parameter.

The syntax for composite operator parameter declarations is:

$$
\textit{compositeFormal} \quad ::= \quad \textit{metaType}\ \ \textit{ID}\ \ (\ \text{`}\texttt{:}\text{'}\ \ \textit{opActualValue}\ )^{?}\ \ \text{`}\texttt{;}\text{'}
$$

5.2.5 Example for Composite Operator with Logic

Section 4.3 describes the logic clause in an operator invocation, which is an expressive mechanism for writing imperative code in SPADE without having to write native (Java or C++) code. Section 5.1 describes composite operators, which are a mechanism for creating reusable operators in SPADE. This subsection does not introduce any new language features, it merely points out that by combining logic with composite operators, you can write expressive and reusable operators in SPADE without native code. Here is an example:

```
composite KeyedMapper(output Output; input Mappings, Queries) {        // 1
  param type      $MapType;                                            // 2
        attribute $key;                                                // 3
```

```
        attribute $val;                                                    // 4
  graph stream<Output> Output = Custom(Mappings; Queries) {                // 5
          logic                                                            // 6
            state    : mutable $MapType m = { };                           // 7
            Mappings : m[$key] = $val;                                     // 8
            Queries  : if ($key in m)                                      // 9
                         submit(Output, {key=$key, val=m[$key]});          //10
    }                                                                      //11
}                                                                          //12
stream<string key, int32 val> Lights = KeyedMapper(Sensors; Queries) {     //13
  param MapType : map<string, int32>;                                      //14
        key     : color;                                                   //15
        val     : intensity;                                               //16
}                                                                          //17
```

The `state` clause (Line 7) declares a map variable used by the per-port logic. There are two input ports: `Mappings` and `Queries`. The `Mappings` port logic (Line 8) records or overwrites a key-value pair in the map and does not produce any output. The `Queries` port logic (Lines 9+10) answers queries by consulting the most recent value for that key, as recorded by the `Mappings` logic. The `Queries` port logic sends its outputs, if any, to the `Output` stream by calling the `submit` function. The `submit` function is available in the `Custom` operator, which can receive and send any number of streams, and does not do anything by itself. `Custom` offers a blank slate for customization. For example, it also offers API functions for generating punctuation.

5.3 Domain Toolkits

SPADE groups operators in namespaces (see Section 6). A toolkit is a set of namespaces that logically organize a collection of operators belonging to the same application domain. For example, the stream-relational toolkit groups operators for the stream-relational domain, similarly to StreamSQL [2]. Operators in the stream-relational toolkit include `Aggregate`, `Join`, `Sort`, `Functor` (which subsumes filter and project), and others. Another SPADE toolkit is responsible for I/O adaptation, i.e., ingesting and externalizing data, with operators like `Source`, `Sink`, `Import`, and `Export`.

An example for an operator from the relational domain is `Aggregate`. It supports aggregating values from a window using aggregators like `Any`, `Min`, `Sum`, etc. Here is an example invocation of `Aggregate`:

```
stream<string ticker, string exchange,                                     // 1
       decimal64 minprice, decimal64 maxprice, decimal64 avgprice,         // 2
       decimal64 sumvolume, decimal64 vwap>                                // 3
  PreVwap = Aggregate(TradeFilter)                                         // 4
{                                                                          // 5
  window TradeFilter : sliding, count(4), count(1);                        // 6
  param  groupBy     : ticker, exchange;                                   // 7
         perGroup    : true;                                               // 8
  output PreVwap     : ticker   = Any(ticker), exchange = Any(exchange),   // 9
                       minprice = Min(price),  maxprice = Max(price),      //10
                       avgprice = Avg(price),  sumvolume = Sum(volume),    //11
                       vwap     = Sum(price * volume);                     //12
}                                                                          //13
```

The `Aggregate` operator requires a window, and has a boolean parameter `perGroup` that specifies whether the window and the aggregation occur separately for each attribute in the `groupBy` parameter. Section 4.5 describes windows in SPADE. In this example, `perGroup` is true and `groupBy` is `ticker/exchange`, which means that each incoming tuple is added to the window corresponding to its own `ticker` and

`exchange` value, and triggers an output tuple that aggregates that window. Since all tuples in the window have, by definition, the same values for `ticker` and `exchange`, aggregation with `Any` in Line 9 yields unique values for that pair of attributes. All other outputs must also be aggregated. As the output assignment `vwap = Sum(price * volume)` illustrates, the argument to an aggregator can be an expression, which can involve more than one input attribute. In fact, the argument can even involve a function call, such as `y = Avg(pow(x, 3))`. When aggregators and functions can have the same name, such as `y = Max(Max(i, j))`, Spade treats the outer call as an aggregator and the inner call as a function. To make the code more readable, the user can disambiguate them by explicit qualifiers, e.g., `y = Aggregate::Max(MathLib::Max(i, j))`. But we encourage avoiding such name collisions in the first place by starting function names with lower-case letters and aggregator names names with upper-case, e.g., `y = Max(max(i, j))`.

Practical advice: The semantics of aggregation and groupBy are defined by the library, not by the language. A separate document will describe the library APIs in more detail.

Language design rationale: The `Aggregate` operator's `groupBy` parameter is a declarative description of functionality that other languages might achieve in a more imperative way. We picked the declarative style because it is familiar to people who know SQL.

For Spade 1 users: Spade 1 had BIOPs (built-in operators); Spade 2 replaces BIOPs by operators in the standard library. Some operators in the Spade 2 library will be composite operators, others will be mixed-mode primitive operators, but since they are invoked in the same way, that distinction should be transparent to the user. Besides the Spade 1 BIOPs, Spade 2 also introduces a few new operators, e.g., `Import`, `Export`, and `Custom`. Other operators will be improved in Spade 2. For example, Spade 2 has more general support for expressions inside aggregations than Spade 1.

5.4 Shared Variables

The design of shared variables in Spade is preliminary. The description in this section is a sketch, details are likely to change.

Shared variables contain state that is shared between multiple operators, which can even belong to multiple jobs or be exposed on a dashboard for a user to monitor a job. Note that sharing data in a distributed system incurs a high cost and can make synchronization difficult. That said, when used responsibly, this feature is very powerful. For example, shared variables can store control state in adaptive applications, or classifier models in learning applications, or distributed data in map-reduce applications [3].

The syntax for a shared variable definition is:

sharedVarDef	::=	*sharedVarModifier** *type ID* ('`=`' *expr*)$^?$ *sharedVarPragmas*
sharedVarModifier	::=	'`public`' | '`static`' | '`mutable`'
sharedVarPragmas	::=	'`;`' | '{' '`pragma`' *nameValues*$^+$ '}'

Static shared variables (with the `static` modifier) have only one instance per job, whereas instance shared variables (without `static` modifier) are instantiated separately each time their operator is invoked. Since static variables do not belong to any particular operator invocation, they can not depend on operator parameters. Instance variables, on the other hand, can depend on parameters of their operator's invocation.

Public shared variables (with the `public` modifier) may be used from anywhere in the system, whereas private shared variables (without `public` modifier) are only visible in the same composite operator body that defines them. Only static variables may be public, all instance variables are private. To access a shared variable from a different composite operator, qualify it with its operator, e.g., `com.ibm.someNameSpace::DefiningOp.s_var`. To access a shared variable from a different application, further qualify it with the name of the other application. For example, in

`definingApp::com.ibm.someNameSpace::DefiningOp.s_var`

the first part (`definingApp`) indicates the application (see Section 6.2), the second part (`com.ibm.someNameSpace::DefiningOp`) indicates the defining composite operator, and the last part (`s_var`) indicates the shared variable.

Mutable shared variables (with the `mutable` modifier) can be written or modified, whereas immutable shared variables (without `mutable` modifier) never change throughout the program execution. Immutable variables of composite types (such as a list) are deeply immutable: all of the contents of the variable (including, for example, list elements) are constant.

Shared variables can optionally have an initializer. For immutable shared variables, the initializer is mandatory. If there is a circular dependency between variable initializers, the compiler produces an error message.

The following example illustrates two shared variables, `s_thresh` and `s_m`:

```
composite CompositeWithStateKeepers(output Out; input In) {            // 1
  var   int32 s_thresh = 10;                                           // 2
        public static mutable map<string, int32> s_m {                 // 3
          pragma lifetime        : eternal;                            // 4
                 consistency     : causal;                             // 5
                 sizeHint        : 16 * 1024 * 1024 * 1024; // 16 GB   // 6
                 writesPerSecond : 5;                                  // 7
                 readsPerSecond  : 500;                                // 8
        }                                                              // 9
  graph stream<In> Tmp = Classifier(In) { param usingMap : s_m; }      //10
        stream<In> Out = Functor(Tmp) { param filter : x > s_thresh; } //11
}                                                                      //12
```

Shared variables can optionally be annotated with a number of pragmas. The example illustrates pragmas for consistency (`atomic > causal > loose`) and lifetime (`eternal > transient`), and estimates of size and access load, which the runtime system uses to pick an appropriate implementation and to allocate resources to implement necessary system support for the shared variables. The exact set of pragmas will be decided as the implementation progresses.

SPADE statically enforces some limitations on shared variable usage to help the user and the compiler reason about consistency. To access a shared variable from a different composite operator than the one that defines it, it must be public and static. To access a shared variable from a primitive operator, the operator model must mention that the operator is stateful, and the operator invocation must mention the variable as part of a parameter expression. And finally, to access a shared variable from a function, it must be passed as a parameter; in a function body, shared variables are not in scope under any other name. Note that functions can only modify parameters that are declared mutable (see Section 3.4).

The implementation of shared variables in the middleware is called *state keepers*. While the language itself does not provide locks or synchronization primitives, the library will expose an API for distributed mutex and condition variables in the state keeper implementation.

Practical advice: Shared variables pose performance and consistency challenges. Therefore, you should use shared variables only when absolutely necessary. If you do use them, you should only declare them public or mutable when you have to. In this language reference, all shared variable names start with "`s_`" to make them stand out in the source code. While this naming convention is optional, we encourage you to follow it in your code as well.

Language design rationale: Given their performance and consistency challenges, we debated whether to support shared variables in SPADE at all. We decided to put them in because there are several compelling use cases for them. If we had omitted the language feature, people would have emulated it by hand, and each such work-around would have been hard for other users to understand and for tools to reason about.

5.5 Primitive Operators

Composite operators are composed of graphs with other operator invocations written in SPADE. All such graphs bottom out in primitive operators, which consist of imperative code written in a native language (C++ or Java). There are two kinds of primitive operators. Simple primitive operators are written directly

in a native language, whereas mixed-mode (MM) primitive operators are written as Perl code generators that generate native code. In both cases, the native code takes advantage of the performance and productivity of traditional languages for straight-line code. Simple primitive operators are easier to write than mixed-mode primitive operators, but mixed-mode primitive operators are more customizable to different invocations.

5.5.1 Simple Primitive Operators

Simple primitive operators are primitive operators written directly in a native language such as C++ or Java. On the SPADE side, they are used with the `Native` pseudo-operator. Here is an example:

```
stream<CoolSchema> CoolStream = Native(Stream1; Stream2; Stream3) { // 1
  param nativeName : "MyCoolUserDefinedOperator";                  // 2
        frequency  : 440.0;                                        // 3
        color      : "Kammerton";                                  // 4
}                                                                  // 5
```

Given this code, the SPADE compiler generates low-level skeletons for `Native_MyCoolUserDefined-Operator`. The user implements these skeletons, using a native API to manipulate the streaming data. For example, a call to `submit` sends streaming data to an output stream. When building the program, the SPADE compiler links the compiled native operator based on compiler options for looking up native code.

If requested, SPADE offers a reflection API, which a simple primitive operator can use to inspect tuple types. The reflection API also supports forwarding attributes from input to output streams where needed.

The details of the native API will be similar as in SPADE 1, to be worked out as we implement it.

For SPADE 1 users: In SPADE 1, simple primitive operators were called "UDOPs", for "User-Defined OPerators". The mechanism was similar, but the look-and-feel was different:

```
[Libdefs]                                                                  # 1
incpath "/inc/myCoolInc"                                                   # 2
libpath "/lib/myCoolLib"                                                   # 3
libs    "myCoolLib"                                                        # 4
[Program]                                                                  # 5
stream CoolStream(schemaFor(CoolSchema)) := Udop(Stream1; Stream2; Stream3) # 6
  ["MyCoolUserDefinedOperator"]                                            # 7
  { switch1 = "test1", switch2 = "test2" }                                 # 8
```

SPADE 1 also had a notion of "raw UDOPs", which operated on tuples directly without going through SPADE's serialization or deserialization. SPADE 2's richer type system makes serialization and deserialization easier to optimize, making it unnecessary to support "raw UDOPs".

5.5.2 Mixed-Mode Primitive Operators

Mixed-mode primitive operators are code generators written in Perl that generate primitive operator instances in a native language (C++ or Java). For example, most operators in the relational toolkit are mixed-mode primitive operators. In operator invocations, mixed-mode primitive operators behave the same as composite operators, attaching streams to ports, passing parameters, defining windows, and so on (see Section 4). For each mixed-mode primitive operator, there is an operator model, which specifies what constitutes a legal operator invocation. For example, the operator model specifies the number of ports and what kinds of streams they support, and specifies the names and meta-types for parameters. As another example, the operator model specifies whether the operator modifies tuple attributes, and if it does not, the compiler checks that parameter expressions do not call functions that can modify tuple attributes.

Figure 6 shows how MM operators are compiled. When the SPADE compiler encounters an operator invocation, it uses a lookup path to find a definition of either a composite operator or a MM operator. Suppose it finds a MM operator. The compiler first consults the operator model, which consists of XML generated

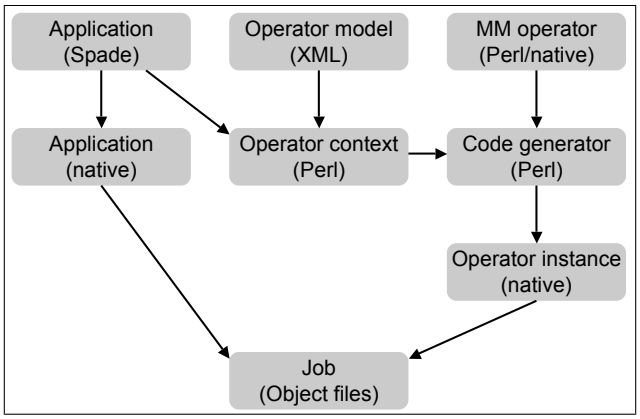

Figure 6: Build process for mixed-mode primitive operators.

by wizards in the IDE. If the operator invocation violates the model, the compiler reports error messages to the user. Otherwise, it combines the operator model with the information from the operator invocation into an operator context, which is a Perl object supplied to the Perl code generator. The MM operator is written as a mix of Perl snippets (the code generator) and native code (code to be generated), hence the name "mixed"-mode. The Perl code is embedded in the native code using ASP-style tags (`<% ... %>` or `<%= ... %>`), similar to how PHP or JSP code is embedded in HTML code using tags. At code generation time, the Perl code uses a Perl API to access the operator context. At runtime, the native code uses a native API to access tuples from streams. For example, the same native `submit` call as for simple primitive operators sends data on an output stream. In the end, both the SPADE code and the MM operators turn into native code, which gets compiled into object files to be deployed as a job.

Practical advice: When writing your own MM operators, keep in mind that parameters expressions may be expensive or cause side effects. Parameters with side effects are bad practice, but not prohibited. You should program your MM operator defensively by maintaining a predictable execution order, and documenting that to the user so they know what to expect when invoking your operator.

Language design rationale: Mixing C++ and Perl code into each other sounds intimidating. However, we argue that this complexity is inherent rather than accidental. Library developers for SPADE need a mechanism that supports flexibility, performance, and reuse of legacy code. To achieve flexibility and performance, we need a code generator framework, where compile-time customization provides flexibility without impacting runtime performance. And to achieve performance and reuse of legacy code, the generated code must be native, since a lot of code already exists in native languages, and native languages provide more traditional imperative programming features than SPADE aspires to support. Note that users can choose to forgo the flexibility of MM primitive operators by authoring simple primitive operators, thus avoiding the complexity of code generation.

For SPADE 1 users: SPADE 1 referred to the operator model as "restrictions", and to mixed-mode primitive operators as *UBOP*s, for "User-defined Built-in OPerators", because they have the performance and ease-of-use of operators that are hardwired and built into other streaming languages. The mechanism was the same, but compared to SPADE 1, restrictions will be improved. For example, SPADE 2 operator models can specify that two input streams must have the same schema; that a parameter must be a number between 0 and 9; or that an operator parameter requires an enumeration, such as the `Source format`, which can take the enumeration constant `csv` for comma-separated values.

5.5.3 Extending Operators

An operator parameter can specify executable code, perhaps compiled from another language such as C++ or Java, to be plugged into the operator's logic, e.g.:

```
stream<int32 linecount, int32 wordcount, int32 charcount> PerLine // 1
  = FileSource()                                                  // 2
{                                                                 // 3
  param fileName     : "/tmp/my_input_file";                      // 4
        udfTxtFormat : "WCParser";                                // 5
        eolMarker    : "\n";                                      // 6
}                                                                 // 7
```

The parameter `udftxtformat` specifies a user-defined text format for data read from the file. In this case, the value `"WCParser"`, or word-count parser, specifies a parser inside the `FileSource` operator.

For SPADE 1 users: SPADE 1 used the same mechanism, for example:

```
stream PerLine(linecount:Integer, wordcount:Integer, charcount:Integer) # 1
  := Source()                                                           # 2
     [ "file:////tmp/my_input_file",                                    # 3
       udftxtformat = "WCParser", eolmarker = "\n" ]                    # 4
     { }                                                                # 5
```

6 Program Structure

Section 6.1 describes the SPADE source code artifacts, whereas Section 6.2 describes how they map into a job that runs on the streaming middleware. Section 6.3 shows how multiple jobs on the same middleware communicate with each other, and Section 6.4 describes how the source code communicates with other static tools.

6.1 Compile-Time Entities

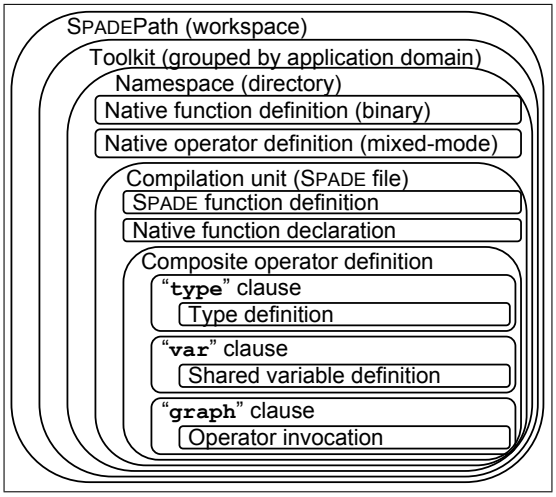

Figure 7: Compile-time entities.

The following example illustrates the various compile-time entities in a SPADE file.

```
namespace com.ibm.samples.myNameSpace;                    // 1
use com.ibm.anotherNameSpace::*;                          // 2
int32 mySpadeFunction(int32 x) { return 2 * x; }          // 3
native string myNativeFunction();                         // 4
```

```
composite SourceSink {                                                          // 5
  type   MyType = int32 i, int32 j;                                             // 6
  var    int32 s_mySharedVar = 10;                                              // 7
  graph  stream<MyType> B = FileSource() { param fileName: "a.dat"; }           // 8
         stream<MyType> C = Native(B) { param nativeName: "SomeNativeOp"; }     // 9
         stream<C> D = SomeOperator(C) { param someParam: foo(i) + bar(j); }    //10
  pragma debugLevel : trace;                                                    //11
}                                                                               //12
```

Each SPADE file belongs to a namespace (Line 1). The `use` directive makes entities from other namespaces available under a simple name (Line 2). For example, instead of `com.ibm.anotherNameSpace::SomeOperator`, the user can simply write `SomeOperator` (Line 10). A SPADE file contains function definitions (Lines 3-4) and composite operator definitions (Lines 5-12). A composite operator can contain clauses with type definitions (Line 6) and shared variable definitions (Line 7). The `graph` clause of a composite operator contains operator invocations that define streams (Lines 8-10). Figure 7 shows how all these compile-time entities fit together. The start symbol for parsing a SPADE file is the compilation unit:

compilationUnit	::=	*namespace*$^{?}$ *useDirective** (*compositeDef* | *functionDef*)$^{+}$
namespace	::=	'`namespace`' *ID*$^{+\cdot}$ '`;`'
useDirective	::=	'`use`' *ID*$^{+\cdot}$ '`::`' ('`*`' | *ID*) '`;`'

A namespace contains all the operators and functions defined in files belonging to the namespace. Thus, a namespace in SPADE is similar to a package in Java. A namespace can also contain native operators or functions. For each native function, the function signature must be defined in a SPADE file (as shown in Line 4 of the example above). Simple primitive operators are defined on the SPADE side by an invocation of the `Native` pseudo-operator (Line 9). Mixed-mode primitive operators are not defined on the SPADE side, though they may be invoked from SPADE (Line 10).

Functions and operators are available under their simple name if they are either part of the same namespace, or made available by a `use` directive. Examples for simple names include `FileSource` (Line 8), `SomeOperator` (Line 10), and `foo` (Line 10) above. For a composite operator to be visible across files, it must have the same name as the file that declares it. For example, an operator `MyOp` would have to be defined in `MyOp.spade` to be found from other files. The `use` directive comes in two forms (similar to Java): it can either refer to a single entity by name, or all entities in a namespace using the wildcard `*`. For example:

```
use com.ibm.yourNameSpace::YourOperator; // 1
use com.ibm.yourNameSpace::yourFunction; // 2
use com.ibm.yourNameSpace::*;            // 3
```

Language design rationale: We picked `::` instead of `.` for namespace qualifiers to distinguish it from attribute access, e.g.: `com.ibm.myNameSpace::myVariable.someAttribute`.

For SPADE 1 users: Here is the same example program written in SPADE 1.

```
[Application]                                                                   # 1
SourceSink trace                                                                # 2
[Program]                                                                       # 3
#include "com.ibm.anotherNameSpace/otherSpadeFile.din"                         # 4
use      com.ibm.anotherNameSpace.SomeOperator                                  # 5
vstream  MyType(i : Integer, j : Integer)                                       # 6
#* no shared variables in Spade 1, omitted s_mySharedVar *#                     # 7
stream   B(schemaFor(MyType)) := Source() [ "file:///a.dat" ] {}                # 8
stream   C(schemaFor(MyType)) := Udop(B) [ "SomeNativeOp" ] {}                  # 9
stream   D(schemaFor(C)) := SomeOperator(C) [ someParam: foo(i) + bar(j) ] {} #10
```

SPADE 1 has no composite operators and no function definitions in the SPADE file. Those features are new in SPADE 2.

There are a few program-level SPADE 1 features that have been removed from SPADE 2. SPADE 1 had two preprocessors: one preprocessor with features such as for loops, and another mixed-mode (Perl embedded in SPADE) preprocessor. SPADE 2 drops the first preprocessor altogether, and will, at some unspecified time in the future, evolve to incorporate all the other needed scripting capabilities, at which point it drops the second preprocessor too. Note that this is unrelated to mixed-mode operators; there is no plan to drop those from SPADE 2. Besides the preprocessors, SPADE 2 drops bundles. Bundles were in the SPADE 1 language to make the preprocessor, and then mixed-mode, easier to use, but can be easily emulated by attaching multiple streams to the same port in SPADE 2.

6.2 Compilation and Deployment

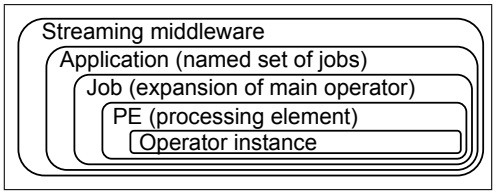

Figure 8: Runtime entities.

To turn the compile-time entities from Section 6.1 into runtime entities, they must be compiled and deployed as a job. To create a job, the user specifies two things: a main operator and an application. The main operator can be any composite operator that has no input or output ports, in other words, any composite operator whose connections clause is empty. The application is a runtime concept, consisting of a set of jobs, and serves for naming communication partners in dynamic application composition (see below). Figure 8 illustrates how all the runtime entities fit together. At the outermost level, there is an instance of the streaming middleware. The streaming middleware hosts one or more applications, and each application consists of one or more jobs. Each job has a main composite operator. To create each job, the compiler starts from main, and expands all operator invocations in the graph clause: it expands mixed-mode primitive operator invocations by running their code generator, and it expands composite operator invocations by expanding their graph clause. The end result is a directed graph where vertices are primitive operator instances and edges are streams. There can be multiple operator instances corresponding to a single operator invocation in the source code; this happens if the composite operator containing the operator invocation is expanded multiple times, as is the case in Figure 4. All expansion happens at compile time. The compiler may fuse multiple operator instances in the job graph into units called processing elements (PEs).

The compilation typically involves source code in multiple files. It starts from a SPADE file with the main operator, but must discover other source files when expanding operator or function invocations. This file lookup follows the SPADEPATH shown in Figure 7, an ordered list of root directories or toolkit manifests from which to start lookup. A typical implementation of SPADE supports SPADEPATH specifications with compiler options. For example, if SPADEPATH contains `/first/root/dir/`, the compiler looks for files implementing namespace `com.ibm.my.nameSpace` starting at the directory `/first/root/dir/com.ibm.my.nameSpace/`. This directory contains SPADE files with functions and composite operators, as well as native source files with native functions and primitive operators. To be more accurate, the implementation of a mixed-mode primitive operator has its own directory, with at least one `generic` subdirectory for code that should work on all platforms, and possibly also platform-specific directories. As of now, SPADE assumes homogeneous hardware on the hosts employed by the streaming middleware.

While pragmas can appear in many places in the code (see Sections 4.2.3 and 5.1.3), certain pragmas typically only appear in a composite operator that is intended as the "main" composite operator in a job.

These include the debug level (one of `error`, `info`, `debug`, or `trace`) and host pool declarations (which can be referenced from pragmas in stream or composite operator definitions, where they serve as deployment directives). The following example shows some of those pragmas:

```
composite SourceSink { // main                                          // 1
  graph                                                                // 2
    stream<int32 i> A = FileSource() { param fileName: "a.dat"; } // 3
    ()                = FileSink(A)  { param fileName: "b.dat"; } // 4
  pragma                                                               // 5
    debugLevel  : trace;                                               // 6
    hostpool    : SourcePool = ["localhost", "10.8.5.6"],              // 7
                  ComputingPool[15] = [];                              // 8
    defaultpool : roundRobin(5);                                       // 9
}                                                                      //10
```

For SPADE 1 users: Here is the same compilation unit written in SPADE 1.

```
[Application]                                          # 1
SourceSink  trace                                      # 2
[Libdefs]                                              # 3
incpath     "/inc1" "/inc2"   # UDOP include path      # 4
libpath     "/lib1" "/lib2"   # UDOP linking path      # 5
libs        "mylib" "yourlib" # UDOP linked libraries  # 6
package     "pk1.h" "pk2.h"   # user-defined functions # 7
[Nodepools]                                            # 8
nodepool    SourcePool[] := ("localhost", "10.8.5.6")  # 9
nodepool    ComputingPool[15] := ()                    #10
defaultpool roundrobin                                 #11
[Program]                                              #12
stream A(i : Integer) := Source() ["file:///a.dat"] {} #13
Nil := Sink(A) ["file:///b.dat"] {}                    #14
```

SPADE 1 divided the compilation unit into sections labeled with square brackets (`[...]`). SPADE 1 `[Libdefs]` become SPADE 2 compiler options; `[Nodepools]` become SPADE 2 composite operator pragmas; and the SPADE 1 `[Program]` becomes the topology of the SPADE 2 main operator.

6.3 Dynamic Application Composition

When a composite operator in the source code is intended as the main operator in a compiled job, its graph clause typically invokes operators for I/O. In particular, it uses various source and sink operators for I/O to URLs, files, databases, etc.; and it uses the operators `Import` and `Export` for I/O to other jobs on the same streaming middleware. Another way for multiple jobs to communicate with each other is via shared variables, see Section 5.4.

Streaming applications commonly run for long periods of time. Streams generated by one application often serve as inputs to another, and the second application may be deployed when the first is already running. The `Import` and `Export` operators support dynamic application composition, where one job exports a stream, another job imports the stream, but both jobs are started separately and may even be mutually anonymous. For example, the streaming middleware may host one or more long-running backbone applications that carry out the bulk of the data processing. Here is an example of exporting a stream to any jobs that may be interested:

```
() = Export(E) {                            // 1
  param howPublished : x="baz", y=[5,6,7]; // 2
}                                           // 3
```

The `howpublished` parameter specifies properties of the stream as name-value pairs. Users can then launch transient applications that import streams, for example, to visualize results in a dashboard. Here is an example of importing a stream from a different job:

```
stream<int32 z> I1 = Import() {                  // 1
  param subscription : x == "baz" && y[0] < 10; // 2
}                                                // 3
```

The middleware connects the `Import` and `Export` streams if the subscription predicate matches the export properties. If the import predicates match multiple streams exported by jobs running in the middleware, they are all connected. If there are no matching streams, nothing arrives on the `Import` operator.

Besides this property-based anonymous connection, SPADE also supports connecting streams across jobs by name. This means that the `Export` operator invocation must define a stream, but does not need a `howpublished` parameter:

```
composite A(output O; input E) {                       // 1
  graph stream<SomeTupleType> O = Export(E) { }        // 2
}                                                      // 3
composite B(output Invoke2; input F) {                 // 4
  graph stream<SomeTupleType> Invoke2 = A(F) { }       // 5
}                                                      // 6
composite Main {                                       // 7
  graph stream<SomeTupleType> G = SomeSourceOp() { } // 8
        stream<SomeTupleType> Invoke1 = B(G) { }       // 9
}                                                      //10
```

The importing job again uses the `Import` operator, but this time with a name instead of a predicate:

```
stream<SomeTupleType> I2 = Import() {                              // 1
  param application : "definingApp";                               // 2
          //application name selected when exporting job launched // 3
        job : some.nameSpace::MainOp;                              // 4
          //main operator selected when exporting job launched   // 5
        streamId : Invoke1.Invoke2.O;                              // 6
          //outside-in name in case of nested composite invokes  // 7
}                                                                  // 8
```

The outside-in stream naming scheme is exemplified by stream `C.I` in Figure 4. Note that there may be multiple instances of the same job in one application. If they export a stream, and an `Import` operator subscribes to that stream by name, the `Import` produces a merge of all the exported streams.

Practical advice: When streams are exported and imported across jobs, there may be a "dual maintenance" problem: the developer must keep both schemas in sync. Therefore, we recommend that you define the schema as a public static tuple type in a separate composite operator. This operator can then be used from both jobs to obtain stream types. There may also be unintended circles of imported and exported streams, so look out to avoid those.

Implementation note: Dynamic optimization in the transport fabric ensures that the bandwidth for these connections is only used when a connection is actually in place. As a rule of thumb, by-name subscription is more efficient than predicate-based anonymous application composition.

For SPADE 1 users: In SPADE 1, import/export were not stand-alone operators, but rather, were an optional part of every stream definition. For example, the exporting job could have a stream like this:

```
export properties [x: "baz"; y: 5, 6, 7] stream E(z : Integer) # 1
  := Functor(A) [ ] { }                                        # 2
```

SPADE 1 translates `properties [x: "baz"; y: 5, 6, 7]` into XML like this:

```
<stream>
  <x>baz</x>
  <y>
    <member>5</member>
    <member>6</member>
    <member>7</member>
  </y>
</stream>
```

SPADE 1 expressed import predicates as XPath expressions. Bearing in mind that XPath indexing is 1-based, the SPADE 1 code corresponding to our earlier SPADE 2 example looks like this:

```
import stream I2(z : Integer) tapping "stream[x='baz' and y/member[1]<10]"
```

6.4 Embedded Documentation

Implementation note: SPADEDOC comments are a special form of delimited comments (see Section 1.4) that contain an extra asterisk (`*`) in the opening tag. In other words, SPADEDOC comments start with `/**` and end with `*/`. They are a form of embedded documentation, similar to doxygen (C), docstrings (Python), or javadoc (Java). Here is an example SPADEDOC comment on a composite operator definition:

```
/** Short sentence describing what the composite operator does.     // 1
    Longer description of the operator's functionality and purpose. // 2
  @input   G description of input stream G                          // 3
  @input   H description of input stream H                          // 4
  @param   Q description of parameter Q                             // 5
  @output  K description of output stream K                         // 6
  @output  L description of output stream L                         // 7
  @tags    G IBM HPQ                                                // 8
  @tags    H ?x                                                     // 9
  @tags    Q TickerAggregationOperator UserDefinedParameter         //10
  @tags    K ?x GOOG                                                //11
  @tags    L IBM "some multi-word free-flow tag" ?x                 //12
  @tagfile "http://mariotagserver.gov/FinanceTags.xml"              //13
  @tagvar  Company ?x                                               //14
*/                                                                  //15
composite M (output K, L; input G, H) {                             //16
  param operator $Q;                                                //17
  graph stream<int32 x> I =  O(G)    { }                            //18
        stream<int32 x> J =  P(H)    { }                            //19
        stream<int32 x> K = $Q(I; J) { }                            //20
        stream<int32 x> L =  R(J)    { }                            //21
}                                                                   //22
```

The comment is semi-structured into optional clauses with labels such as "`@input G`" or "`@tags K`". These clauses can be consumed by other SPADE-related tools. For example, the IDE (integrated development environment) uses the `@input`, `@param`, and `@output` clauses to display documentation on a composite operator's interface. The inquiry services tool uses the `@inq` clauses to automatically assemble programs from end-user specifications.

Here is an example SPADEDOC comment on an operator invocation:

```
  /** Split into streams of low and high values.                    // 1
      The threshold value is 100: values under 100 are considered  // 2
      low, values of 100 or above are considered high.              // 3
    @input  I  stream with all values, including both low and high // 4
    @output Lo stream with values below 100 only                    // 5
    @output Hi stream with values above 100 only                    // 6
  */                                                                // 7
  (stream<int32 x> Lo; stream<int32 x> Hi) = LoHiSplitter(I) {      // 8
    param loFilter : x < 100;                                       // 9
          hiFilter : x >= 100;                                      //10
  }                                                                 //11
```

A VWAP Example

VWAP, or "volume-weighted average price", is a common calculation in financial trading. Below is an example that uses VWAP to illustrate Spade features.

```
composite VWAP {                                                                    // 1
param                                                                               // 2
  expression<set<string>> $monitoredTickers : { "IBM", "GOOG", "MSFT" };           // 3
                                                                                    // 4
type                                                                                // 5
  TradeInfoT  = decimal64 price, decimal64 volume;                                  // 6
  QuoteInfoT  = decimal64 bidprice, decimal64 askprice, decimal64 asksize;          // 7
  TradeQuoteT = TradeInfoT, QuoteInfoT,                                             // 8
                tuple<string ticker, string dayAndTime, string ttype>;              // 9
  TradeFilterT= TradeInfoT, tuple<timestamp ts, string ticker>;                     //10
  QuoteFilterT= QuoteInfoT, tuple<timestamp ts, string ticker>;                     //11
  VwapT       = string ticker, decimal64 minprice, decimal64 maxprice,              //12
                decimal64 avgprice, decimal64 vwap;                                 //13
                                                                                    //14
graph                                                                               //15
  stream<TradeQuoteT> TradeQuote = FileSource() {                                   //16
    param fileName : "TradesAndQuotes.csv.gz";                                      //17
          format   : csv, compressed, nodelays;                                     //18
          columns  : irange(1,3), 5, irange(7,9), [11, 15,  16];                    //19
  }                                                                                 //20
                                                                                    //21
  stream<TradeFilterT> TradeFilter = Functor(TradeQuote) {                          //22
    param  filter      : ttype=="Trade" && (ticker in $monitoredTickers);           //23
    output TradeFilter : ts = timeStringToTimestamp(dayAndTime);                    //24
  }                                                                                 //25
                                                                                    //26
  stream<QuoteFilterT> QuoteFilter = Functor(TradeQuote) {                          //27
    param filter : ttype=="Quote" && (ticker in $monitoredTickers);                 //28
  }                                                                                 //29
                                                                                    //30
  stream<VwapT, tuple<decimal64 sumvolume>> PreVwap = Aggregate(TradeFilter) {      //31
    window TradeFilter : sliding, count(4), count(1);                               //32
    param  groupBy     : ticker;                                                    //33
           perGroup    : true;                                                      //34
```

```
    output PreVwap     : ticker   = Any(ticker), vwap = Sum(price*volume),      //35
                         minprice = Min(price),  maxprice = Max(price),         //36
                         avgprice = Avg(price),  sumvolume = Sum(volume);       //37
  }                                                                             //38
                                                                                //39
  stream<VwapT> Vwap = Functor(PreVwap) {                                       //40
    output Vwap : vwap = vwap / sumvolume;                                      //41
  }                                                                             //42
                                                                                //43
  stream<timestamp ts, decimal64 index>                                         //44
    BargainIndex = Join(Vwap as V; QuoteFilter as Q)                            //45
  {                                                                             //46
    window V            : sliding, count(1);                                    //47
           Q            : sliding, count(0);                                    //48
    param  equalityLHS  : V.ticker; // can also be written as nested loop join: //49
           equalityRHS  : Q.ticker; // "condition : V.ticker == Q.ticker"       //50
           perGroupLHS  : true;                                                 //51
    output BargainIndex :                                                       //52
      index = vwap > askprice*100.0 ? asksize*exp(vwap-askprice*100.0) : 0.0;   //53
  }                                                                             //54
                                                                                //55
  () = PerfSink(BargainIndex) { }                                               //56
                                                                                //57
pragma                                                                          //58
  debugLevel: trace;                                                            //59
}                                                                               //60
```

B Grammar Overview

This appendix surveys the SPADE syntax top-down, with references to the sections that discuss the semantics. Refer to Section 1.4 for the lexical syntax, and Section 1.3 for the grammar notation.
The compilation unit is the start symbol of the SPADE grammar.

compilationUnit	::=	*namespace*$^?$ *useDirective**	# Section 6.1
		(*compositeDef* | *functionDef*)$^+$	
namespace	::=	'namespace' $ID^{+.}$ ';'	
useDirective	::=	'use' $ID^{+.}$ '::' ('*' | *ID*) ';'	

Composite operators are defined at the top-level in a namespace.

```
compositeDef      ::= compositeHead compositeBody                          # Section 5.1
compositeHead     ::= 'composite' ID ( '(' compositeInOut+; ')' )?
compositeInOut    ::= ( 'input' | 'output' ) ( streamType? ID )+,
streamType        ::= 'stream' '<' tupleBody '>'
compositeBody     ::= '{'
                      ( 'param' compositeFormal+ )?
                      ( 'type' typeDef+ )?
                      ( 'var' sharedVarDef+ )?
                      ( 'graph' opInvoke+ )?
                      ( 'pragma' nameValues+ )?
                      '}'
compositeFormal   ::= metaType ID ( ':' opActualValue )? ';'               # Section 5.2.4
opInvokeActual    ::= ID ':' opActualValue ';'
opActualValue     ::= type | expr
nameValues        ::= ID ':' expr+, ';'                                    # Sections 4.2.3 and 5.1.3
```

Streams are defined in a composite operator's `graph` clause.

```
opInvoke          ::= opInvokeHead opInvokeBody          # Section 4
opInvokeHead      ::= opOutputs '=' ID opInputs          # Section 4.1
opOutputs         ::= opOutput | '(' opOutput*; ')'
opOutput          ::= streamType ID ( 'as' ID )?
opInputs          ::= '(' portInputs*; ')'
portInputs        ::= streamType? ID+, ( 'as' ID )?
opInvokeBody      ::= '{'                                # Section 4.2
                      ( 'logic' opInvokeLogic+ )?
                      ( 'window' nameValues+ )?
                      ( 'param' opInvokeActual+ )?
                      ( 'output' opInvokeOutput+ )?
                      ( 'pragma' nameValues+ )?
                      '}'
opInvokeLogic     ::= ID ':' stmt                        # Section 4.3
opInvokeOutput    ::= ID ':' ( ID '=' expr )+, ';'       # Section 4.2.1
```

Functions are defined at the top-level in a namespace.

```
functionDef       ::= functionHead functionBody                            # Section 3.4
functionHead      ::= functionModifier* type ID '(' functionFormal*, ')'
functionModifier  ::= 'public' | 'native' | 'stateful'
functionFormal    ::= 'mutable'? type ID
functionBody      ::= ';' | blockStmt
```

Shared variables are defined in a composite operator's `var` clause.

```
sharedVarDef      ::= sharedVarModifier* type ID                           # Section 5.4
                      ( '=' expr )? sharedVarPragmas
sharedVarModifier ::= 'public' | 'static' | 'mutable'
sharedVarPragmas  ::= ';' | '{' 'pragma' nameValues+ '}'
```

Imperative statements can appear in function bodies or the logic clause of an operator invocation.

```
stmt          ::= localDecl | blockStmt | exprStmt                    # Section 3.3
                | ifStmt | forStmt | whileStmt
                | breakStmt | continueStmt | returnStmt
localDecl     ::= 'mutable'? type ( ID ( '=' expr )? )+, ';'
blockStmt     ::= '{' stmt* '}'
exprStmt      ::= expr ';'
ifStmt        ::= 'if' '(' expr ')' stmt ( 'else' stmt )?
forStmt       ::= 'for' '(' type ID 'in' expr ')' stmt
whileStmt     ::= 'while' '(' expr ')' stmt
breakStmt     ::= 'break' ';'
continueStmt  ::= 'continue' ';'
returnStmt    ::= 'return' expr? ';'
```

Expressions can appear in many places in the grammar. For precedence and associativity, see Section 3.1.

```
expr            ::= prefixExpr | infixExpr | postfixExpr                # Section 3
                  | conditionalExpr | '(' expr ')' | ID | literal
prefixExpr      ::= prefixOp expr
prefixOp        ::= '!' | '-' | '~' | '++' | '--'
infixExpr       ::= expr ( infixOp | mappedOp | assignOp ) expr
infixOp         ::= '+' | '-' | '*' | '/' | '%' | '<<' | '>>' | '&' | '^' | '|'
                  | '&&' | '||' | 'in' | '<' | '<=' | '>' | '>=' | '!=' | '=='
mappedOp        ::= '.+' | '.-' | '.*' | './' | '.%' | '.<<' | '.>>' | '.&'
                  | '.^' | '.|' | '.<' | '.<=' | '.>' | '.>=' | '.!=' | '.=='
assignOp        ::= '=' | '+=' | '-=' | '*=' | '/=' | '%=' | '<<=' | '>>='
                  | '&=' | '^=' | '|='
postfixExpr     ::= ID '(' expr*, ')'
                  | type '(' expr ')'
                  | expr '[' subscript+, ']'
                  | expr '.' ID
                  | expr postfixOp
subscript       ::= expr | ( expr? ':' expr? )
postfixOp       ::= '++' | '--'
conditionalExpr ::= expr '?' expr ':' expr
```

Literals are the highest-precedence expressions denoting values.

```
literal          ::= primitiveLiteral | listLiteral | setLiteral        # Section 3
                   | mapLiteral | tupleLiteral
listLiteral      ::= '[' expr*, ']'                                     # Section 2.2.1
setLiteral       ::= '{' expr*, '}'
mapLiteral       ::= '{' ( expr ':' expr )*, '}'
tupleLiteral     ::= '{' ( ID '=' expr)*, '}'                           # Section 2.2.2
primitiveLiteral ::= 'true' | 'false' | STRING | FLOAT | INT            # Section 2.1;
```

Types are defined in a composite operator's `type` clause.

```
typeDef     ::= 'public'? ID '=' ( type | tupleBody ) ';'        # Section 5.1.2
metaType    ::= 'attribute' | 'expression' typeArgs?             # Section 5.2.4
              | 'function' typeArgs? | 'operator' | 'type'
type        ::= ID | 'void' | primitiveType | compositeType      # Section 2
typeArgs    ::= '<' type+, '>'
typeDims    ::= '[' expr+, ']'
```

Primitive types are types without other types as arguments.

```
primitiveType ::= 'boolean'                                      # Section 2.1
                | 'enum' '{' ID*, '}'
                | 'int8' | 'int16' | 'int32' | 'int64' | 'int128'
                | 'uint8' | 'uint16' | 'uint32' | 'uint64' | 'uint128'
                | 'float32' | 'float64' | 'float128'
                | 'decimal64' | 'decimal128'
                | 'complex32' | 'complex64' | 'complex128'
                | 'timestamp'
                | 'blob'
                | 'ustring'
                | 'string' typeDims?
```

Composite types are type constructors for composing new types out of other types.

```
compositeType ::= tupleType                                      # Section 2.2
                | 'list' typeArgs typeDims?
                | 'map' typeArgs typeDims?
                | 'set' typeArgs typeDims?
                | 'matrix' typeArgs typeDims
tupleType     ::= 'tuple' '<' tupleBody '>'                      # Section 2.2.2
tupleBody     ::= attributeDecl+,
                | ( ID | tupleType )+,
attributeDecl ::= type ID
```

C Acknowledgments

This language specification benefited from substantial, detailed, critical, constructive, and sometimes humorous feedback by many people. We incorporated some suggestions directly, many in modified forms, and for those we did not incorporate, we clarified our rationale. The following people commented on earlier drafts of this document: Bob Blainey, Norman Cohen, Dan Debrunner, Fred Douglis, Jim Giles, Michel Hack, Paul Jones, Richard King, Tracy Kimbrel, Senthil Nathan, Rodric Rabbah, Anand Ranganathan, Anton Riabov, and Chitra Venkatramani.

References

[1] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. SPC: A distributed, scalable platform for data mining. In *Workshop on Data Mining Standards, Services and Platforms (DM-SSP)*, 2006.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *Journal on Very Large Data Bases (VLDB J.)*, 2006.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation (OSDI)*, 2004.

[4] Buğra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and MyungCheol Doo. SPADE: The System S declarative stream processing engine. In *International Conference on Management of Data (SIGMOD)*, 2008.

Index