A Model-Based Framework for Building Extensible, High Performance Stream Processing Middleware and Programming Language for IBM InfoSphere Streams

Buğra Gedik†,1, Henrique Andrade†

† IBM Research, 19 Skyline Dr, Hawthorne, NY 10532, USA

SUMMARY

This work presents an extensive case study on the model-based design of a commercial-grade stream processing middleware (IBM’s InfoSphere Streams), its runtime and language (SPL) compiler. The model-based underpinnings are pervasive throughout the whole environment, from describing inter-process communication interfaces and objects to the design of the extensibility mechanism in the runtime and language. In addition to many software engineering advantages such as consistent, uniform, and self-documented integration among the different parts of the system, we show intrinsic performance benefits to the platform derived from this design approach. First, we demonstrate how an incremental compilation strategy employed by the SPL compiler and rooted on the model description of the application, extracted by the compiler as part of the application building process, leads to better compile-time performance. Second, we discuss how the model-based code generation strategy employed by the SPL compiler also leads to increased runtime performance, by specializing the generated code to particular characteristics of the runtime environment. Finally, we show how the extensibility strategy used in the SPL language leads to automatic syntactic and semantic checks at compile-time, while enabling behavioral reasoning and specific optimizations at runtime.

KEY WORDS: Data Stream Processing, Model-based Code Generation

†Correspondence to: Buğra Gedik via e-mail at bgedik@us.ibm.com. Henrique Andrade did this work while at IBM Research. He is now with Goldman Sachs in New York.
1. Introduction

General-purpose computing platforms and languages must be extensible. This fundamental requirement is centered on providing mechanisms that, while abstract, can be used to decompose the implementation of an application into a set of components devised by the developer, as a close representation of the real-world problem addressed by his application.

While extensibility is a fundamental goal, a major challenge is to provide this capability with the least amount of overhead, avoiding the high cost of runtime introspection and interpretation. The central aspect of this work is to show how these seemingly conflicting goals can be achieved cleanly in a high-performance, distributed stream computing platform by extensively relying on what we term model-driven design and model-based code generation.

In this work, a model is a structured and representative view of the problem or environment to be replicated by a computer program. Models can be refined and extended to incorporate additional detail as needed, helping with the evolution of a computing platform by progressively providing additional capabilities to address new requirements imposed by customers and evolving business demands.

The model-based framework forms the basis of the extensibility provided by our language and runtime system. Their research prototype roots and evolution into a commercial-grade software platform are briefly discussed in Section 2. In Section 3 we look in detail at how we used these ideas throughout the entire system and the specific benefits we derived from relying on this approach. For example, a model description can be used to communicate information between the different parts of a system that must interact with one another. Furthermore, we use the notion of a model to capture and expose as much information as deemed necessary to allow detailed reasoning about a program to be done at compile-time as is described in Section 4. As will be shown, this approach minimizes runtime overheads that are typical of stream computing platforms that rely on interpreting queries, as is the case for many SQL query engines. This leads us to the runtime environment and its model-driven backbone, which we discuss in Section 5.

Specifically, this paper presents a model-driven framework we used to develop a new stream processing language (SPL) as well as the distributed runtime (InfoSphere Streams) supporting this language, starting with their research counterparts, respectively, the Spade language and the System S middleware.

Approaching a design task as a model-driven process, whereby a common integration framework is made explicit, is a methodology that has been used in disparate areas such as manufacturing, embedded systems, communication systems, as well as in certain software technologies. Nevertheless, the work described here is innovative in how these ideas form the basis of every software engineering facet (Section 6) of a large commercial software system.

Our fundamental engineering contributions include the practical development of a model-driven design applied directly to a real-world software system including the following components:

- A naturally extensible programming language: as a general-purpose stream processing platform, Streams must be flexible to accommodate, efficiently, the development of
applications from several engineering and business domains. This requirement forced us to design a language where operators and many syntactic constructs cannot be baked in. In other words, the language evolution cannot be dictated by us, the language designers. Rather it has to be guided by the (somewhat unpredictable and unanticipated) needs of applications. Thus, the language must provide modularity and extensibility mechanisms to allow the designers themselves to innovate and supply the constructs they need.

• A structured and uniform runtime system: a complex and distributed middleware comprising multiple components presents a substantial design challenge. A common blueprint comprising interfaces and objects to be designed by different teams must be in place to ensure uniformity, homogeneity, and co-evolution. Thus, it is imperative that this blueprint, in the form of a set of visible models representing component interfaces and common objects (and their different perspectives), be visible by the different teams and directly connected to the code being written.

• A self-documenting, code generation-based software development process: complex software systems sometimes grow organically as requirements expand and the development team evolves. Thus, rooting the software design process on model descriptions, from which documentation and code can be automatically produced ensures that the internal and external documentation as well as the code evolve together.

While these ideas immediately translate into a great deal of flexibility for a minimalistic stream processing language, it also provides a code base that can evolve to incorporate new functionality, minimizing the problems usually associated with architecture decay [27] and maintainability [10, 15]. We better articulate these points in Section 7 where we distill some of the learned lessons during the implementation of the IBM InfoSphere Streams product.

Our discussion is framed in the context of a stream processing platform, yet we postulate that the strategies we employed and more importantly the engineering and performance benefits we observed can be applied and replicated in other large-scale distributed software projects. We elaborate on this point in our concluding remarks in Sections 8.

2. Foundations: The System S Platform

The framework described in this paper was used to design both the runtime and the SPL compiler that make up InfoSphere Streams [37], or Streams, for short. Since the runtime and compiler were architected to provide a general development framework for multi-purpose stream processing applications, they themselves provide a model-driven interface to developers. Streams originated from the System S middleware developed over the last 7 years at IBM Research [3, 24, 25, 31, 38, 39, 48, 57, 60, 61, 62, 63, 64].

To provide some background, we first describe System S in more detail. This discussion is useful so we can explore how pervasively the model-based design approach has been used throughout Streams. Its architecture inherits many of System S’ design decisions, but also incorporates considerable additional architectural refinement over the earlier research prototype.
2.1. The Spade Language and the System S Middleware

Streams’ SPL [30] language has its roots in the Spade language. Spade [7, 33] is the programming language used for developing stream processing applications [31, 57] on System S. We briefly describe some of the programming language features (shared also by SPL), focusing on the language’s unique features:

Flow Composition: The Spade language provides composition capabilities that are used to create data flow graphs out of basic analytical units called operators. Operators are composed into data flow graphs via stream connections. This is called static flow composition, where the topology of the data flow graph is constructed at application development time and does not change at runtime. The language also supports dynamic flow composition, where connections are established at runtime, based on conditions specified at development time and availability of matching streams at runtime. In addition to these, SPL also supports hierarchical flow composition via composite operators. A composite operator encapsulates a data flow graph as an operator, which enables modular development of large-scale streaming applications.

Flow Manipulation: The Spade language provides a type system and an expression language. They provide the basic constructs for expressing flow manipulations as custom functions and operators. A standard set of operators that are parameterizable using the Spade expression manipulation syntax, as well as a large set of built-in functions are also provided. The standard set of operators include basic relational manipulations [20] such as selection, projection, aggregation, join, and sort, as well as common utility-type support for buffering, throttling, splitting, merging, etc. Fundamental abstractions, such as windows, punctuations, output functions, and operator parameters, among others are provided to facilitate flexible and powerful flow manipulations [2]. The language also provides a set of edge adapters to connect operators to external sources and sinks, for ingesting external stream data and producing results to be consumed by external applications, respectively.

Extensible Analytics: The Spade language supports extensibility through toolkits. A toolkit is a set of reusable operators and functions [60]. The language can be extended with new operators, written in general-purpose programming languages, such as C++ and Java. Such operators can be type-generic and parametrizable. In other words, they can potentially operate on any type (as in a projection operator in relational algebra) and can be customized using Spade’s expression language (as in defining a match condition for a relational join operator). This enables third parties to provide cross-domain as well as domain-specific toolkits of operators and functions, encapsulating various streaming analytics from a wide range of application domains [57] (e.g., from finance engineering, to signal processing, to video processing).

---

2A brief description of these abstractions is provided in Section 4.1.2. We also refer the interested reader to the SPL language specification [36] where longer and more comprehensive discussions of these features can be found.
Distributed Execution: The Spade language provides various configuration options that influence the mapping of the logical data flow graph to a physical one, which can be deployed on a set of distributed hosts. For instance, the language can be used to express operator fusion constraints, which influence how operators are grouped into partitions that map to operating system processes. Similarly, it can be used to specify partition placement constraints (such as co-location, ex-location, and isolation constraints), which influence how processes representing the partitions are mapped to hosts. These constraints can be used to express detailed requirements in terms of how an application executes on the distributed platform. The Spade language can also be used to specify threading and queueing configurations, controlling the data processing behavior within an operator and of an operator's input port, respectively.

System S is the runtime platform on which Spade applications run. It provides management services such as scheduling, security, job management and monitoring, fault-tolerance, and dynamic stream traffic routing. It also provides a high-performance communication substrate for Spade applications, making available highly efficient transport capabilities for cross-process and cross-host communication. Finally, the runtime supplies application monitoring services, including application visualization and flow visualization as well as performance metric access.

2.2. Rationale for Employing a Model-Based Design in Streams

From the brief description in Section 2.1 one can see the vast expanse of a system design space that System S and its commercial counterpart, Streams, cover. Through the multiple iterations in prototyping and implementing the system (spanning more than 7 years), we have found that a model-based organization would be fundamental to the large-scale software engineering effort to produce the commercial-grade version of the system. Furthermore, this approach would help address basic challenges imposed by implementing and ensuring inter-operation between runtime components. Finally, a model-based design was the anchor for providing extensibility capabilities, primarily to the SPL programming language of Streams, but also to the middleware as each of its components evolves and matures as a product over the next few years.

While many of the engineering challenges are well-understood, some of them deserve a more in-depth discussion:

Extreme high-performance runtime: Many stream processing applications must cope with very high data and processing workloads. For example, options market data processing in finance engineering applications must ingest, process, and react to more than a million transactions per second. In situations like this, the middleware overhead has to be decreased to a minimum. From customizing the application code to optimal use of locking and queueing (as described in Section 4), to operator fusion, and minimizing data copies to perform in-place data manipulation, a model description of an application aids in conveying and representing certain aspects of the application to code generators that can produce highly specialized code for that application. As will be described in Section 4.2, a central strategy in...
designing both the runtime and language for Streams relied on code generation to strike the right balance between reuse and a high-degree of customization.

Large-scale applications and support for incremental building: Many stream processing applications are distributed, large, multi-component systems. The process of compiling and linking these applications is usually time-consuming due to the amount of source code that needs to be processed by the compiler. A salient feature of many of these applications is the replication of certain functional data flow subgraphs. For example, in processing data from the stock market, the basic data processing related to each stock symbol is exactly the same. Indeed, some of the data related to each stock symbol is processed in a pipeline, whose stages are functionally the same as far as the computation that they perform. Due to the volume of data, these pipeline stages are replicated, sometimes multiple times per stock symbol. Therefore, the same code can be generated for representing the operators implemented in these subgraph stages. Moreover, modifications to the operators themselves (e.g., for implementing a better asset pricing algorithm) must be carried over and deployed for all stock symbols. Hence, numerous compiler-time optimizations can be put in place to minimize the amount of code to be generated, compiled, and linked as will be described in Section 4.2.3.

Multiple application domains and the need for toolkits: Streams is a general purpose middleware. As such, an intrinsic requirement it has concerns the ability to provide the building blocks for developing application from multiple domains. For example, finance engineering applications have the need for specific types of operators whereas intelligent traffic management might have the need for a different set of operators altogether. While many imperative programming languages provide object-oriented and modularization constructs, the SPL language relies on primitive and composite operators as its fundamental constructs. The design of new operators, the creation of toolkits with closely related operators, and the implementation of tooling to support the building of these elements are greatly simplified if both the code generation and the specific tools can use a model description of the operators. As will be shown in Section 4, model instances of operators and toolkits are extensively used by the SPL compiler, allowing a large degree of extensibility in the language.

Evolving middleware capabilities: Streams was designed to evolve over the next few years as new requirements are put in place by commercial users of the platform. As described in Section 5, both interfaces and objects forming the backbone of the Streams runtime components are represented by human-readable models. These representations capture inter-component interactions and the structure of the objects exchanged in those interactions. Moreover, both descriptions are used in the build process of the overall system to produce the actual Java and C++ source code, together with the rest of the hand-written code, implementing the middleware components. Therefore, developers and architects can readily see the overall architecture, including interfaces and objects, as well as make modifications directly to the models (to incorporate changes and new features), thus automatically producing the new source code, easing the process of evolving the platform. This point will be further analyzed in Sections 4 and 5.
3. Model-Based Design

In this section, we describe the underpinnings of model-based design as observed through the lenses of our own experience in developing Streams and SPL.

Exposing models as entities that are kept separate from the source code makes for a substantial decoupling of the implementation from the interfaces and data objects used by the system, simplifying several, typically hard, software engineering challenges. Among them, we highlight the following:

**Documentation:** A difficult challenge in software projects is to ensure that the system architecture documentation is kept in sync with the implementation as the latter evolves. This is difficult because it requires discipline and close interaction between developers and architects. A model representation is usually self-documenting. It generally includes textual human-readable descriptions, which are kept together with the model description. Because a model usually is described using a structured representation (e.g., using XML or an interface definition language), it is generally easy to extract the information necessary to generate the core pieces of a textual architectural documentation in an automated fashion.

**Maintainability:** Many high complexity systems are implemented such that their backbones (key objects and interfaces) are intertwined with the actual implementation. While the original developers are usually familiar with the decisions and overall code organization, as the system evolves and additional complexity is introduced, it becomes increasingly more difficult to keep track of system-wide interdependencies. With a model-based design, interfaces and objects can be modified in one location and a substantial part of the code can be automatically generated from the models (e.g., object classes, stubbed service interfaces, state machines, etc.). Thus, not only can new developers see, at any point, what the backbone of the system consists of, but there is much less burden on maintaining the code as a substantial portion of the process is automated.

**Uniformity:** Another software engineering difficulty in implementing large-scale systems and, in particular, middleware such as ours is to accomplish structural, nomenclature, and interface uniformity in designing service APIs and object classes across different implementation languages and components. This is even more challenging as the software code base evolves and new developers are onboarded. A model-based design is helpful in this regard as well, as it can function as the cornerstone for cross-language and cross-component uniformity. In other words, different language bindings for interfaces and server service dispatchers are automatically generated and, therefore, are consistent by construction. This also helps with the conceptual uniformity of the code base, where the same names and structures are employed across different components, making end-to-end understanding of the code easier.

**Object Facet Management:** Application development middleware such as ours, where language and runtime support are provided, have the characteristic that multiple constructs representing a single conceptual object might exist in slightly different forms, depending on when and where it is used. For example, information pertaining to a computer program is
usually augmented with runtime information. Specifically, once a program becomes a process, it is usually augmented with information such as a process id, a parent-child process relationship, the physical memory it currently uses, etc. While different object classes representing different facets on the same object can be manually created and managed, and this is often how it is done in practice, a model-based approach can also simplify this aspect. Not only can different object facets be provided, but the relationship between them can also be made explicit, including natural representations of is-a and has-a relationships between related objects. As will be seen in Section 5.1, we have made extensive use of this approach spanning several language constructs. For example, an application and their operators have augmented counterparts in the middleware runtime, where an application becomes a job and an operator becomes an operator instance. Both have associated runtime information used for management, debugging, and visualization tasks.

**Tooling Construction and Extensibility:** Large-scale middleware generally relies on extensive tooling for administration and configuration. Our case is typical. Streams is a reasonably complex middleware and SPL is an extensible language. On the one hand, to make the utilization and administration of the middleware manageable and user-friendly, a combination of a collection of management command-line tools and web-based interfaces must be provided. On the other hand, to ease the process of application development as well as to support language extensions via toolkits (similarly to library development), a graphical IDE as well as
visualization tools [24, 25, 26] must be provided. The model-based design, in this case, provides a structured view (e.g., Eclipse’s EMF-based [28] models) on the objects and interfaces that these tools can rely on to create wizards and the necessary graphical user interfaces. Figure 1 depicts the IDE with the operator model editor and source code editor perspectives as examples of the final product.

3.1. Model Representation and Tools

In general, we employed three representations for the models used by the runtime, compiler, and language extensibility mechanisms:

1. Objects manipulated by the runtime and parts of the compiler were described using an XSD representation [29]. An XSD-based code generator was used to output C++ and Java classes corresponding to the objects (using CodeSynthesis XSD [21] and JAXB [54], for C++ and Java respectively). Examples of these classes of objects are shown in Sections 4 and 5.

2. The language extensibility mechanism for the SPL language employed a home-grown code template description language. Models are used to carry information from the compiler proper to its operator code generators. In this way, the code generators can provide the high performance code specialization necessary for data-/compute-intensive applications. This aspect as well as specific examples are shown in Section 4.

3. System call interfaces were described using an Interface Definition Language (IDL) (we chose Corba’s [22] IDL [46, 47]). Using the omniORB [34] IDL compiler and the native Java JDK IDL compiler, C++ and Java bindings were automatically generated. Note that we did not rely on Corba’s native type system to describe the API parameters given its limitations. Instead we relied on XSD-described objects as mentioned earlier. Examples of runtime interfaces are given in Section 5.

While the choice of tools is particular to our platform, the underpinnings of a model-based design are independent from it. As previously discussed, the fundamental property of using a model-based design is to expose the internal structures of a system, specifically the cross-component service call interfaces and objects. In our case, another important requirement was code generation, a technique we used for two main reasons. First, because it ensured that our operator-based language could be extended, but also to ensure that highly-efficient code could be produced for the computational platform at hand [60]. Second, because we wanted to ensure that the internal makeup of the system is exposed, documented and always kept in sync as the platform and code that implements it evolves.

4. The Design of the Compiler and Language Runtime

In this section, we describe the foundations of the language extensibility mechanisms as well as the internals of the SPL compiler. We focus primarily on discussing their reliance on models that guide the code generation process. As we articulated earlier, the model-based design functions
as the primary means for removing runtime overheads as well as for providing language-based extensibility to application and toolkit developers.

### 4.1. A Composition Language

SPL is primarily a dataflow composition language. It is used to compose flow graphs out of stream processing operators. Operators are used to encapsulate reusable streaming analytics, as well as adapter logic to connect to external systems. In this section, we first define the fundamental concepts and terminology involved in creating stream processing applications, followed by a discussion of the extensibility model offered by the SPL language.

#### 4.1.1. Concepts and Terminology

A *stream processing application* is a data flow graph consisting of a set of *operator instances* connected to each other via *stream connections*. A streaming application can be instantiated and run on Streams. An instance of one such application is referred to as a *job*.

An *operator* is a reusable piece of stream processing logic. It is reusable in the sense that it can be instantiated in different ways within a stream processing application or across different applications. An instance of an operator is referred to as an *operator instance*. As an example, consider a *Functor* operator, which can have multiple instances that are configured with different selections and projections. Figure 2 shows one particular instantiation of the *Functor* operator.

An *operator input port* is a channel through which an operator can receive tuples. Pictorially, input ports are represented with a distinctive icon on the left of operators as seen in Figure 3. Similarly, an *operator output port* is a channel through which an operator can send tuples. Examples of that can be seen on the right hand side of operators depicted in Figure 3. Each operator port has a specific *schema* and all tuples coming through this port follow the port’s schema. Examples of schemas can be seen in Figure 2 where we use the keyword *type* to define them, e.g., the *Line*, *WordCharStat*, and *LineStat* types. Each operator output port defines a *stream*, which has a unique name (e.g., *Data* and *Result* seen in the *WordCount* composite operator in Figure 2).

A *stream* consists of a time ordered series of tuples, defined by an operator output port. The tuples in a stream are structured according to the stream’s *schema*, which, as we mentioned, is inherited from the port that defines the stream. A tuple consists of an ordered set of named and typed *attribute* values, for example, the attributes *words* and *chars* in the *WordCharStat* type shown in Figure 2. The names and types of these attributes are defined by the stream schema.

A *stream connection* links an operator’s output port to another operator’s input port. Multiple examples are seen in Figure 3. There could be multiple stream connections originating

---

3A *Functor* operator is used to perform the equivalent of a relational algebra *projection*, i.e., it provides basic tuple and attribute manipulations consisting of transforming a tuple’s attribute value, removing a tuple’s attribute, as well as adding an attribute to a tuple and associating a value with it.
namespace sample;

type Line = string line;
type WordCharStat = int32 words, int32 chars;
type LineStat = int32 lines, WordCharStat wcs;

7 composite WordCount {
8 graph
9 stream<Line> Data = Ingest() {
10   param file : getSubmissionTimeValue("file");
11 }
12 stream<LineStat> Result = Process(Data) {}
13 } as Sink = Report(Result) {}
14 }
15 }
16 composite Ingest(output stream<Line> Out) {
17 param
18 expression<rstring> $file;
19 graph
20 stream<Line> Out = FileSource() {
21   param file : $file;
22   format : line;
23 }
24 }
25 }
26 composite Report(input stream<LineStat> In) {
27 graph
28 () as Sink = FileSink(In) {
29   param file : "/dev/stdout";
30 }
31 }
32 composite Process(input stream<Line> In;
33 output stream<LineStat> Out) {
34 graph
35 stream<LineStat> OneLine = Functor(In) {
36   logic
37     state: mutable list<rstring> tokens;
38     onTuple In:
39       tokens = tokenize(line, " \t", false);
40     }
41     output
42       OneLine: lines = 1, wcs = { words = size(tokens), chars = length(line) + 1 };}
43 stream<LineStat> Out = Custom(OneLine) {
44   logic
45     state: mutable LineStat aggregate = {
46       lines=0, wcs = { words=0, chars=0 } ;
47     onTuple OneLine:
48       aggregate.lines += OneLine.lines;
49       aggregate.wcs.words += OneLine.wcs.words;
50       aggregate.wcs.chars += OneLine.wcs.chars;
51     }
52 onPunct OneLine:
53     if(currentPunct()==Sys.FinalMarker)
54       submit(aggregate, Out);
55 }
56 }
57 }
58 }
59 }
60 }
61 }
62 }

Figure 2: The WordCount application, roughly equivalent to the well-known Unix wc utility, expressed in SPL. The main composite of the application, i.e., the application entry point, is the WordCount composite operator (line 4). It organizes the flow into three high-level components that handle the ingest, processing, and reporting tasks, organized in a pipeline of instances of the following composite operators: Ingest (used in line 3 and defined in line 17), Process (used in line 32 and defined in line 34), and Report (used in line 56 and defined in line 57). The Ingest operator uses a FileSource (line 20) to read the data as a series of tuples, each containing a single line. The Process operator uses a Functor operator (line 34) to count the number of characters, words, and lines for each tuple. It uses a subsequent Custom operator (line 52) to aggregate the per-tuple information into a summary for the entire dataset. The final punctuation (Sys.FinalMarker in line 55), a special marker propagated by the runtime to indicate the end of a stream, is used to output the summary information when all the input is read. The Report operator uses a FileSink operator (line 58) to write the final summary on the standard output.
Figure 3: A logical operator-based data flow view of WordCount, depicted in Figure 2. Operator instances are named using the name of the first stream they produce, unless an explicit name is given to them via the as clause. For instance, in stream<Line> Data = Ingest() {...}, the instance of the Ingest operator resulting from this invocation is named Data. Such operator instance names can be nested. For instance, the FileSource operator is named Data.Out.

at a given output port (a fan-out), and similarly, multiple stream connections arriving at a given input port (a fan-in). All stream connections that are attached to an input port must have the same stream schema, which has to match the schema of the port.

Stream connections can be one of two types: static and dynamic. A static connection is specified explicitly at application development time and is known at compile time, specifically its origin and destination ports. Connecting operators this way is called static composition.

A dynamic connection is specified at application development time and is established at execution time. This is achieved by exporting a stream by associating export properties to an output port defining a stream. On the receiving side, an import subscription is associated to an input port.

Whether a dynamic connection will be established between an output port with an exported stream and an input port depends on whether this input port’s import subscription is satisfied by the set of properties of that exported stream, at execution time. Dynamic connections can be used to establish connections that cross job and application boundaries. They can also be programatically altered during the application lifetime by changing the import subscription and export properties, dynamically. Connecting operators this way is called dynamic composition.

A processing element (PE) is a runtime container for executing operator instances. Similar to operators, PEs also have ports. An operator output port that has a connection going into the input port of another operator in a different PE has a corresponding output port dedicated to

---

4 A processing element is the smallest schedulable execution unit in Streams. A processing element hosts a segment of an application’s data flow graph, comprising a subset of the operators in an application and maps to an operating system process that is individually managed by the Streams job management service as well as by the operating system itself.
Figure 4: Two possible physical data flow graphs associated with the WordCount application depicted in Figure 2. Each larger block corresponds to a processing element. Different operator placements imply different communication costs between operators.

4.1.2. Stream Processing Operators & Concepts

The SPL language provides a set of abstractions that facilitate the development of reusable operators. These abstractions, which are described in the rest of this section, address the common needs that arise in defining generic stream processing operators and in configuring their instances.

An expression language is provided for specifying operator configurations involving references to stream attributes as well as functions to perform data manipulations. Typical examples are the expressions used to represent the processing parameters and assignment of output results. One such use case for the Custom operator instance is seen in Figure 4 (lines 52-54). The same expression language is used also for window configurations and output functions as we will describe shortly.

Operator parameters can be used for specializing an instance of an operator by updating its configuration. An example is a match condition parameter used for a Join operator to specify...
how tuples are to be correlated, or a format parameter for a FileSource operator used to specify the format of the input file, also seen in Figure 2 (lines 17 and 18).

Operator output assignments can be used for expressing how attributes of streams are populated, based on computation performed on attributes from inbound tuples. An example is an expression used to represent output attribute assignments in the Functor operator used in Figure 2 (lines 14–19). In this case, the attribute words is populated with the result of size(tokens) and the attribute chars with the result of length(line) + 1.

Operator output functions can be used for representing transformations applied to a set of input tuples. An example is an aggregation function used in an Aggregate operator to specify how the set of tuples in the operator’s input port window are to be reduced. In other words, one can employ Min and Max reduction functions on a subset of input tuples, currently residing in the operator’s input port processing window [36] to compute the attribute values of the corresponding output tuple. Note that operator windows can be associated with each input port to specify how inbound tuples are to be buffered, and when to operate on them. Various window types are available in SPL, for example, tumbling and sliding windows, with different eviction and trigger policies [36].

Finally, stream punctuations are out-of-band markers transported by a stream. Punctuations are used for creating application-specific boundaries within a stream. For instance, punctuations can be used to specify a custom window of tuples within the stream, by interleaving the punctuations with the tuples within the stream. An example of an operator processing incoming punctuations can be seen in Figure 2 (lines 60–61) in the context of the Custom operator in the form of the onPunct OneLine: statement. In this case, the operator instance is looking for an end-of-stream marker (i.e., Sys.FinalMarker) as a means of finalizing the computation performed by that operator.

Virtually all of these operator properties are captured by an operator model as will be described in further detail in the next section.

4.1.3. The Extensibility Mechanism: Toolkits

SPL’s extensibility mechanism is based on the concept of toolkits. A toolkit is a set of SPL constructs (e.g., operators, functions, type definitions), placed in a package. The main goal of toolkits is to make these constructs reusable across different applications. For the purpose of this paper, we will mostly focus on the infrastructure for defining new operators.

In SPL, operators are implemented as code generator templates (an example is shown in Figure 3). We provide a detailed description of code generation templates later in this section. In addition to the operator implementation itself, the SPL toolkit management tooling associates a model with each operator (an example is shown in Figure 4). This model, referred to as the operator model, is specified by the operator developer guided by a Streams Studio wizard.

---

5An eviction policy specifies how tuples are to be expired from a window, whereas a trigger policy specifies when operations are to be performed on the set of tuples in a window.
Figure 5: A sample code generator template for the Beacon operator. This operator can generate tuples periodically, with user-defined output attribute assignments for the outbound tuples. The code generator template is a mixture of target code (C++) and generator code (Perl in \%
blocks). The generator code uses a model object ($model in the code) to introspect the operator invocation at compile-time and generate specialized code based on the specific configuration of the operator instance at hand. In this particular example, the code specializations include the handling of the period of tuple production ($period in line 2), the iteration count ($iterations in line 5), and the assignments of the output tuple attributes (lines 12-14).

This model serves three important purposes. First, it feeds the compiler with the basic syntactic properties of the operator. This is used by the compiler to verify the correctness of the operator instances configured as part of a streaming application. Second, the model provides the compiler with the basic semantic properties of the operator. This is used by the compiler to perform compile-time semantic checks, compile-time optimizations, as well as run-time reasoning enabling several dynamic optimizations. Third, the model describes to the compiler all the external dependencies the operator has, for example, on proprietary as well as open source libraries. In Figure 1, the panel named “Operator Model” depicts a schematic form of the operator model, while the panel named “Properties” zooms in on the attributes associated with an operator’s input ports as represented in that operator’s model.

The syntactic elements in an operator model include the number of input and output ports (which can potentially be variable for different instances of the same operator), parameter names and types, syntactic constraints on parameter and output expressions, the list of available output functions, window specifications, among various other settings. These elements
Figure 6: A sample operator model fragment for the Beacon operator, capturing its semantic and syntactic properties. As an example of a semantic property this code segment demonstrates how the model represents this operator’s threading behavior (line 4), which in this case is always single-threaded. As an example of a syntactic property, the parameters section (lines 6-18) lists the parameters accepted by this operator (e.g., the iterations parameter in lines 9-15). The model specifies that this parameter is optional, that it is defined by an attribute-free expression, i.e., its value cannot refer to any inbound stream tuple attributes, that its value must be of type uint32, and it can only have one single value associated with it.

enable an operator developer to declare the valid syntax of the operator so that error checking and reporting can be performed by the compiler.

As an example, a binary Join operator requires two input ports on all of its instances, whereas Barrier operator instances may have different number of input ports. In the rare case that the declarative features supported by the operator model are not sufficient for capturing advanced syntactic requirements, the code generator can be used for detecting more complex syntax violations.

The semantic elements in an operator model are used to declare the behavioral facets of the operator, including aspects such as threading semantics, punctuation semantics, mutability of operator ports, and state management. A stateful operator generally manages internal data structures representing its state. This aspect of an operator is particularly important to capture in the model due to the implications to fault tolerance, which are discussed later in this paper.
The compiler can use these elements to perform semantic checks. For instance, knowing whether an operator expects a punctuated stream to work properly can be used to detect invalid flow graphs during the compilation process. Some of the semantic elements are also used to perform compile-time optimizations, such as eliminating locks on single threaded segments of the flow graph or eliminating tuple copies between operator ports that are not mutating their tuples. Furthermore, semantic elements are also used to perform dynamic optimizations at run-time, such as auto-parallelizing stateless operators or creating replicated segments to improve performance via data parallelism.

In general, without semantic elements in the operator model, the operators are seen as black boxes by the system, which would prevent various opportunities for compile-time and runtime optimization of the sort we just discussed. Conversely, with semantic elements in the operator model, the runtime system and the compiler both have increased visibility into the operator internals, without requiring the operator implementation to follow a rigid framework.

While the responsibility of implementing the semantics declared in the operator model lies on the shoulders of the developer, the operator model can also be used by a runtime verification component to automatically check for violations. Simple examples of such checks include detecting multi-threaded tuple submissions by an operator that has declared itself as single-threaded or detecting a punctuation submission by an operator that has declared its output ports as punctuation-free ports. The potential for implementing these checks is still not fully exploited by the existing tooling, but the underpinnings are present as a means to allow for smooth evolution of the product, leveraging one of the advantages of using a model-driven design as we posited in Section 3.

Finally, the dependency information in an operator model defines the set of external libraries used by the instances of the operator. This information is used by the system to perform dependency analysis at compile-time to locate and fetch the required external libraries. It can also be used by a provisioning system to resolve dependencies as part of the application deployment process, ensuring that the required dynamic libraries are available at runtime. Again, this is one of the several examples of the synergy between compile-time constructs and runtime processing afforded by the model description associated with an operator.

---

7 More precisely, these optimizations can only be activated upon inspection of where that operator instance is in the overall application flowgraph, how it is used in that context, and in conjunction with the information represented in the operator’s model. The SPL compiler has specific rules for eliminating copies and locks. For example, when the compiler can establish that only one upstream output port is feeding another operator’s input port, it can eliminate locking on that operator’s internal processing logic. Similarly, copying a tuple on arrival at an input port is not necessary, if the operator processing logic does not modify the incoming tuple (e.g., when only filtering on incoming tuples is performed).

8 In our experience since early on in the development cycle, operator developers make extensive use of existing open source libraries, which provide certain domain-specific functionalities such as image processing or data mining. This approach not only allows SPL applications to leverage these types of software assets, but also proprietary legacy analytics, making them available in a stream processing context.
4.2. Code Generation

As previously mentioned, the SPL language uses a code generation approach as a means to provide language extensibility. Fundamentally, operators are added to the language as code generator templates. Such templates define a skeleton for the operator implementation and are customized based on specific configurations provided by the operator instance used in an application.

The code generator templates are written as C++ fragments as seen in Figure 5, wherein a scripting language is used to customize these fragments using the operator instance model as the input. Various helper routines are provided for common code generation tasks, usually providing boiler plate code such as the prologue and epilogue of the operator corresponding C++ class, emitted by the generator.

For instance, in Figure 5, the initial section within the <%%> markers (starting in line (**)) is used to retrieve the operator instance configuration, using the operator instance model stored in the $model object. Perl is the language used in these fragments in the template.

The remaining code is C++, interwoven with locations customized via code generation directives. These directives typically employ information from the operator instance model. For instance, Beacon’s period parameter, which is used for pacing the process of generating new outbound tuples, as well as its iterations parameter, expressing how many outbound tuples to produce, are populated from values retrieved from the operator instance model. Similarly, the assignment of values to the attributes in the outbound tuples is also configured with information from the operator instance model.

When an operator instance is found in an SPL program, the compiler constructs the operator instance model. Subsequently, it uses the code generator template for that type of operator to emit code that is specialized based on the contents of the operator instance model. As hinted before, this specialization is a function of how that operator instance is being used in the application (i.e., its configurations as elected by the programmer) as well as a function of where in the flow graph it appears (i.e., which will determine locking and threading configurations).

There are several advantages to this approach. First, code generation allows a great deal of flexibility in terms of the operator’s implementation, while achieving good runtime performance compared to frequently time-consuming runtime schemes (for example, reflection and introspection, which tend to be computationally more expensive). In fact, specific instances of an operator can be fully customized without requiring runtime data introspection or code interpretation.

Second, compile-time checks can be implemented to define a very precise syntax for the operator. The operator instance model can be used to programmatically validate the configuration of the operator instance. Furthermore, error messages can be issued such as to point to specific locations in the SPL source code containing the invocation of the operator. In other words, the operator code generator has access to the line information contained in the operator instance model created by the SPL compiler, allowing the mapping back to the application source code on the occurrence of compile-time errors pertaining to the operator invocation syntax. This organization make the operators that come from third party toolkits.
Third, the code generator templates can be used to connect to external systems at compile time and perform code generation and error checking tasks based on information obtained from these systems. Examples include connecting to databases to perform schema specific code generation as well as schema validation. This capability is used by the operators that perform what we term edge adaptation, i.e., ingesting and converting data from external databases (such as IBM DB2 [23]) or other middleware (such as IBM MQ [45]) where input records must be translated into SPL tuples. To accomplish this, the malleability afforded by the code generation templates comes handy. Specifically, the operator instance model includes the schema information associated with the operators’ ports required to, for example, map stream schemas to database schemas and to convert database records into SPL tuples.

4.2.1. Operator Code Generation Overview

We now describe the basic compilation and execution workflow for SPL applications. Figure 7 gives a schematic overview of this workflow, with a focus on operators and how they are compiled and executed.

We start with the elements authored by software developers. We divide them into two categories. The first one is the application elements written as SPL source code by the application developers. The second one is the toolkit artifacts in the form of operator code.

---

Figure 7: Overview of the code generation process employed by the SPL compiler.

---

9The SPL’s standard toolkit includes a set of relational, utility, and adapter stream processing operators. This toolkit has also been completely developed using the approach outlined in this section, where each operator is developed as a code generator template.
generator templates (operator templates in short) and the corresponding operator models written by the toolkit developers as described in Section 4.1.3.

The SPL compiler first takes the operator models and the application source code as input. It uses the operator model to perform syntactic and semantic checks on the operator instances found in the application source code. It also creates operator instance models by specializing the corresponding operator models, based on the configurations of the operator instances found in the application source code.

The instance model contains all the customizations and information related to that specific instance of the operator for that application and for the site in the application flow graph where that operator instance is located. Once the operator instance models are generated, the SPL compiler takes as input the operator templates and feeds them to a template preprocessor. The task of the template preprocessor is to convert the operator templates into code generators. This step is only performed once per operator template. In fact, the operator templates are converted into code generators immediately prior to compilation, as part of the toolkit packaging workflow implemented by the Streams Studio IDE. During this process, the skeleton C++ code in an operator template is converted into emit statements in the code generator, whereas scripted pieces that involve logic for generating C++ code based on the operator instance model are inserted verbatim into the code generator. Once a code generator is produced, the SPL compiler feeds the operator instance models to the code generators to produce pure, fully customized, C++ code.

In addition to generating code for operator instances, the SPL compiler also generates code for the tuple types used by an application. For each stream schema, a class representing tuples of that type is generated. When tuple types involve other nested tuple types, additional classes are generated for them.

Tuple types are also represented as models by the compiler. These models serve two purposes. First, they are used as part of the operator instance model to provide compile-time type information to the developer of an operator code generator template. Second, they are used by the compiler to generate in-place data manipulation and highly efficient binary and character serialization and deserialization functions for the tuple types and their attributes. Figure 8 gives an excerpt from a tuple model for a nested tuple type.

Finally, the C++ code generated for operator instances and types is compiled by a native compiler (in our case, gcc), and native shared libraries corresponding to the applications processing elements are generated.

10The SPL type system [85] supports the declaration of fixed-size data types including strings, hash maps, and lists. Their fixed-size nature allows the pre-computation of attribute offsets, which in turn, allows for in-place manipulation of these attributes, avoiding deserialization costs as well as data copies. This is one of the techniques contributing to lower the overall tuple processing costs in an application [85]. The C++ classes generated by the SPL compiler provide getters and setters that transparently support these in-place data manipulations. In our previous work [85], we termed them façade classes.
4.2.2. From the Generated Code to a Running Application

In summary, there are two types of artifacts that result from an SPL application compilation: the shared libraries mentioned in Section 4.2.1 and the application instance model\(^\text{11}\) which is output as an XML document (an example is depicted in Figure 9b). The application model instance fully describes the topology of the application and includes details about the operator instances and their execution containers (i.e., the set of application processing elements).

The application instance model can subsequently be submitted to the Streams runtime to instantiate a job corresponding to the application. The Streams runtime uses this model to schedule and instantiate the processing elements that constitute a job. As will be described in more detail in Section 5, each processing element is then started as a Unix process and this process loads the shared libraries that contain the application code, corresponding to the

\(^{11}\)The application instance model is an application-specific instance of the application model.
set of operator instances hosted by this PE. This is performed using an augmented PE model instance (seen in Figure 10), which is a runtime enhanced version of the PE model that is generated from the application model. This augmented PE model has sufficient information in it for the SPL language runtime to start all operator instances that are part of the PE, connect them to each other, as well as connect the PE to its upstream and downstream PEs running on the same or remote hosts as separate processes.

4.2.3. Management of Code Generation Artifacts

The SPL compiler manages the artifacts produced as part of code generation using two important mechanisms: incremental compilation and code sharing.

Incremental Compilation: The SPL compiler’s heavy reliance on code generation has minimal impact on the length of the edit/compile/debug cycle, even for very large applications in the finance engineering domain such as options processing [48], since the compilation process is incremental.

When there are changes to the source of an SPL application, additional code generation and binary compilation happens only for the operator instances and types that have changed. This is achieved by caching the operator instance models and type models and comparing the current ones with the previously generated ones to decide if new/modified C++ code needs to be generated at all [4]. Indeed, this process is extremely important [4], not because the SPL code generation is time-intensive, but rather because the C++ compilation and linking process is.

Code Sharing: While incremental compilation significantly reduces the time it takes to rebuild an application, compiling a SPL program from scratch may take a long time due to the generation and compilation of code for the individual operator instances. However, the SPL compiler mitigates this problem by detecting similarities between different instances of the same operator and sharing their code whenever possible.

This is achieved by performing compile-time evaluation of constant expressions, separating resulting constants from the operator instance model, and replacing them with placeholder variables that are initialized at runtime [5]. This enables sharing of code for different operator instances that only differ slightly in their configurations.

The code sharing opportunities are detected by comparing the operator instance models after performing constant expression evaluation and substitution of constants with placeholders. Use cases include multiple instances of a source operator [46] that read data from different origins (e.g., different files in the file system or different TCP socket endpoints) or multiple instances of a filter operator that are configured with different thresholds (i.e., filter expressions that are structurally the same, but with different constants in the expression parse tree leaf nodes).

Furthermore, many streaming applications involve repeated subgraphs resulting from the data parallel nature of the flows, in which case the opportunity for code reuse across different operator instances is very high [7, 8]. When operator instances that have different

---

12While we will not describe this process further. Please refer to our earlier work for additional detailed information [4, 5].
configurations share their code, the specialization of their functionality is performed at runtime by loading the configuration constants that were extracted at compile-time. Such constants are included as part of the application instance model. They propagate to the augmented PE model and eventually reach the language runtime (which we describe next). There, they are used to customize the operator instances by initializing the placeholder variables with their respective constant values at startup time.

We empirically verified that a trivial change in an options processing application can trigger a multi-hour rebuild process, if the incremental compilation and code sharing functions are disabled, as opposed to a few seconds, when they are enabled.

4.3. The Language Runtime

The SPL language runtime is responsible for managing operators that are co-located within the same processing element as well as for interacting with the transport subsystem to manage the physical connections supporting the stream connections between PEs and their own operators. The transport subsystem is not used for stream connections between operators running in the same processing element. In this case, fusion is used to implement the connections between the operators. Nevertheless, in both cases, two types of communication endpoints are represented in the augmented application instance model:

Operator Communication via Fusion: The augmented PE model contains information about the internal topology of a PE, including the operator instances this PE hosts as well as the stream connections between them. The SPL language runtime instantiates these operators and establishes the connections between them by attaching the output ports to input ports. Thus, as an operator runs, tuple read/write operations are implemented using ordinary C++ function calls (in contrast to network exchanges for non-PE-co-located operators).

Operator Communication via the Transport Subsystem: The augmented PE model also contains information about the connections that originate at operators located in a particular PE, but destined to operators hosted by other PEs, as well as connections that are destined to operators contained in that PE, but originating in operators running in the context of other PEs. This information is used to setup transport subsystem connections between PEs. Each PE input port serves as a data receiver whose endpoint address is registered with the Streams distributed name server, using a label known to the PEs that will be sending data to this port. Each PE output port serves as a data sender that looks up the endpoint addresses of its destination input ports using the aforementioned labels. This enables the data sender operators to discover and connect to data receiver operators. Static composition is implemented based on this mechanism. Dynamic composition works similarly, with the exception that the

---

13We usually distinguish the language runtime from the Streams runtime. The former is the runtime infrastructure in charge of providing services to the operators, whereas the latter is the middleware infrastructure itself in charge of managing and supporting the middleware components and the global well-being of the applications sharing the runtime environment as is described further in Section 5.

14The assignment of operators to PEs is performed at compile-time, either based on directions from the application developer or based on a profile-driven optimization process.
Streams runtime is involved in the process, providing notifications to PEs about newly available consumer or producer operators, dynamically.

Streams supports multiple network substrates [48] as part of its transport subsystem. The default TCP transport is the general purpose transport layer, requiring no other configurations beyond the communication endpoint labels. For applications that have more stringent latency and/or throughput requirements, a more advanced transport layer that has a large number of configuration options to tune buffering, thread usage, among other parameters, is also provided. This advanced transport utilizes the IBM MQ Websphere Low Latency Messaging [40] middleware, and can work with TCP/Ethernet, InfiniBand, or shared memory (for establishing stream connections between PEs that are running on the same host).

All of this information is represented in structured form in instance models. Excerpts showing the representation of the two types of communication endpoints can be seen in Figures 9b and 10 for intra-PE connections and for inter-PE connections, respectively.

5. The Design of the Runtime System

In this section we describe how the Streams runtime system employs a model-based design, allowing its interfaces and object class definitions seamlessly integrate with the compiler-generated objects described in Section 4.

The Streams runtime exists to provide services to the multiple applications that can simultaneously be running on the platform and to manage the computing environment, which might comprise a large collection of distributed compute and data storage nodes [16].

The Streams runtime includes the following services: job and lifecycle management services [38], scheduling [61], fault tolerance [38, 39], debugging [31] and performance monitoring [42]. Other services like physical infrastructure monitoring and general management and administration are also provided, but they are peripheral to the discussion in this section, albeit making use of Streams model-based design too.

In most operating systems, there exists a distinction between a program (a binary executable generated as a result of compiling and linking the source code that defines an application) and a process (the runtime construct spawned as a result of running the program). As previously discussed, in Streams we use the term application and job to denote the logical entity corresponding to an OS program and the runtime entity corresponding to an OS process, respectively. As alluded to in Section 4 a model-based representation is used by the compiler as a means to define the structure of an application and to communicate that information to the runtime system, which augments it with additional information pertaining to the physical resources an application makes use of as it runs.

---

15 Note that inter-PE connections are enhanced with runtime identifiers, in this case the PE identifiers, which are only available at runtime and, therefore, only available in the augmented model shown in Figure 10.

16 While Streams runs on a single laptop, large-scale applications, for example, in the finance engineering domain [8, 45], may make use of and span more than a hundred nodes organized in a cluster.
Before delving into that, we will start by describing the different views on closely related conceptual objects used by the compiler and runtime environment and how the model-based design we employed simplified the internal architecture of both the compiler and runtime components.

5.1. Compile-Time and Runtime Object Views

A developer describes an application using the SPL language as discussed in Section 4.1. From a logical standpoint, an application is simply a collection of operators interconnected by streams. For the most part, developers are insulated from most runtime issues, but may elect to configure several settings that will affect how the application is deployed. These settings range from constraints such as co-placement or ex-placement of operators (meaning, respectively, placing two or more operators in the same processing node or forcing their placement to be on separate processing nodes) [62]. A developer may also choose to create application partitions, i.e., subsets of operators that are placed inside a single PE. A developer might also decide to place partitions on specific pools of processing nodes to benefit, for example, from better network connectivity or increased storage. Finally, a developer might choose to make use of queues, deployed in the input ports of operators. The use of these queues allows multiple operator threads to be consuming and processing incoming data concurrently, effectively parallelizing the internal operator logic.

The logical characteristics of an application coupled with the optional physical configurations we just discussed are captured by the compiler, which configures an application model producing the corresponding application instance model as described in detail in Section 4.2.2. In Figure 9a we show a small excerpt of the application model whereas in Figure 9b we depict the structured representation generated by the SPL compiler, corresponding to an application instance model. The information in the instance model can be directly used at runtime. Services like job management, fault tolerance, debugging, and visualization must be able to remit an application analyst back to the source code and its constructs. For example, when seeing a visual representation of applications that are currently running in the platform [24], the analyst must be able to place a breakpoint on the arrival of tuples to a particular input port [31]. Hence, the compile-time application representation is maintained by the runtime, specifically by the system’s job management component, as the application runs. This data can be queried and manipulated in many ways via the different client interfaces.

Despite the substantial amount of information produced by the compiler, the compile-time representation of an application must be augmented with runtime-only information. For example, logical processing node pools must be converted to actual nodes available in the runtime environment; logical stream connections must be converted to physical instantiations (e.g., TCP ports) of the Streams transport subsystem (see Section 4.3).

An augmented application instance model (seen in Figure 10) is created from the application instance model, once the job submission utility (command-line or web-based) instructs the runtime to launch a new application, by making use of job management system calls (Section 5.2). Note that this instance model includes job-specific logging directives (logLevel) and user-related information (user), which are only relevant to the middleware runtime.
5.2. Providing Runtime Services: System Calls

Processing elements provide an execution context for a set of operators. The placement of operators on processing elements may be done manually (via language directives) or automatically (via an optimization step \[62\]), during the compilation step.

A PE’s lifecycle starts with the set of Streams job management components orchestrating the PE startup, after a request issued by a user. The last step of this process consists of spawning a Processing Element Container (PEC), which in turn invokes the PE execution entry point function, handing to it an augmented PE model instance, which includes both compile-time configurations (as we alluded to in Section \[5.1\]) as well as runtime information filled in by the job management component. The runtime information includes the physical endpoints for static and dynamic streams (Section \[4.1.1\]), fault-tolerance directives, but also
Figure 10: Excerpt from an augmented application instance model. The augmented application instance model contains, among other things, a list of augmented PEs. This excerpt shows various properties associated with an augmented PE, such as the PE id, job ID, PE launch count, and user id, which represent runtime-augmented information (lines 1-3). The augmented PE also contains information that is borrowed from the compile-time representation of the PE from the application instance model, such as the job name, language, restartable/relocatable modes, etc. The model instance also shows augmented PE ports and connections (lines 5-13). Unlike their compile-time counterparts, the augmented ports and connections reference runtime ids.

hooks that enable the PEC to service system calls pertaining to the processing element itself and the operators it hosts.

These system calls might be invoked by the system management components (e.g., in the form of routing notifications pertaining to dynamic subscriptions an operator might have) or by users (e.g., to retrieve instantaneous performance metric readings from an operator, stating, for example, the current rate of arrival of tuples to one of its input ports). Similarly, operators might themselves make use of system calls querying management components, in particular, when they employ information on the system dynamics to tweak their own behavior (e.g., to adjust to a cheaper operating point when the runtime environment is being overly taxed [57]).

The system calls employed by the processing elements themselves as well as by external parties (e.g., other Streams management components or directly by users) are implemented as Corba remote procedure calls. These calls as well as the objects that they transport as input/output parameters are defined in the PEC service interface model. Similarly, each management component exposes its own interface, expressed in Corba’s IDL and each object class used by that interface is expressed in XSD form. In Figure 11 we show an IDL excerpt of the Streams job management component (referred to as the Streams Application Manager, or SAM for short) with the job startup and cancelation service interfaces. The appSet object in the submitJob interface is roughly a collection of application instance models, and
module Streams {
  interface SAMService {
    void submitJob(in string rpcTid, in string session,
    in Opaque appSet, in string submitParams,
    out string jobInfo, out string samException);
    void cancelJob(in string rpcTid, in string session,
    in long long jobid, in string cancelParams,
    out string jobInfo, out boolean forceToContinue,
    out string samException);
  }
}

Figure 11: IDL fragment for job submission APIs. These APIs are part of the Streams Application Manager (SAM) that manages the lifecycle of the applications running on the system. The submitJob (lines 3-5) and cancelJob (lines 6-9) APIs are used for basic job submission and cancellation functionality. Note that the appSet parameter to the submitJob (line 4) is an Opaque type. The structure of the appSet is defined using XSD, as shown in Figure 9.

As we mentioned, all these object classes and the service interfaces described in a component’s model, once compiled by the IDL and XSD compilers, are directly used in the implementation of these middleware components and also by the SPL compiler.

In the next sections, we will look a little closer at some of the system interfaces and their model descriptions, which provide the system calls that directly aid the construction of scalable and extensible stream processing applications.

5.2.1. Job Management

The job management service interface includes functions for starting, stopping (for a potential subsequent re-start), re-starting, canceling, and monitoring applications.

The model-based design was instrumental in providing substantial code reuse and helping with managing the distinct conceptual facets of objects such as applications, processing elements, and operators as described in Section 3. In this case substantial code reuse is possible because both the IDL and the XSD descriptions are used to automatically generate the code employed by both the client side of the job management interface as well as the server side, where the service interface is automatically stubbed out by the IDL compiler [34]. Several versions of the client interface are created without any further work, enabling the same job management services to be provided in C++ and Java and made available to end users via a web-based management graphical user interface (GUI), a command-line interface (CLI), and embedded in Streams Studio, the Eclipse-based IDE.
<sam:systemTopology>
  ...
  <sam:applicationSets>
    <sam:applicationSet name="sample::Vwap.appset">
      <application>
        <pes>
          <pe index="0" language="C++" logLevel="error" optimized="true"
            relocatable="false" restartable="false">
            ...
          </pe>
        </pes>
      </application>
    </sam:applicationSet>
    ...
  </sam:applicationSets>
  ...
  <sam:jobSets>
    <sam:jobSet>
      <sam:job applicationScope="Default" appsetId="0" id="0" name="sample::Vwap"
        state="INSTANTIATED" submitTime="1288919641" user="bgedik">
        <sam:pes>
          <sam:pe host="10.4.40.210" id="0" index="0"
            jobId="0" launchCount="1" reason="NONE" state="RUNNING">
            <sam:health healthSummary="UP" isHealthy="true"
              optionalConnectionsSummary="UP" requiredConnectionsSummary="UP"/>
          </sam:pe>
        </sam:pes>
      </sam:job>
    </sam:jobSet>
    ...
  </sam:jobSets>
</sam:systemTopology>

Figure 12: Excerpt from the model instance representing the system topology. It contains both compile-time information about applications (the applicationSets element in lines 3-15) as well as run-time information about their instances, aka jobs (the jobSets element in lines 17-32). The latter contains runtime information such as the host a PE is running on, its state, and its health status (lines 19-20).

As we have discussed earlier, the submission of an application for execution triggers a chain of events that results in a collection of PEs being deployed in the runtime environment. Throughout the different stages of the application submission process, the objects representing an application, its collection of processing elements, and each of the PEs' operators go through a model augmentation process, whereby additional runtime management information is tacked on, creating runtime versions of these objects. The lineage of these objects is kept by having identifiers that relate, for example, a logical description of a processing element produced by the compiler with the runtime processing element augmented object. These identifiers, seen in
Figure 10 plus the corresponding state information related to each of these runtime objects are the handles by which application administrators can interact with running applications, querying their status and manipulating their state. Figure 12 shows a partial view of one such object, depicting the runtime environment state as a structured model. A PE in this augmented view has fields such as `state`, representing the current status of the PE in its lifecycle, and `launchCount` representing the number of times the PE was launched, which are relevant only at runtime.

Finally, as we discussed in Section 5, the descriptions of all these model objects are externalized by the job management interface model description and are readily available as part of the documentation that captures the inner workings of the system.

5.2.2. Fault Tolerance

Streams supports two main fault tolerance mechanisms: processing element mobility \[38\] and checkpointing \[39\]. Checkpointing enables an operator to periodically persist its internal state in durable storage, whereas mobility enables an operator to be moved \[17\] from a failed host to another one. As the SPL compiler parses the source code describing an application, it collects information on the fault tolerance behavior supported by the operators used in the application (from the operator models described in Section 4) as well as fault tolerance configurations specified by the programmer in the application source code.

Once this application instance model is submitted for execution using the appropriate job management API (described in Section 5.2.1), the fault tolerance data is also augmented. In particular, the augmented PE model instance contains information such as the processing element `launch count` \[18\] (seen in Figure 12) and the location on the distributed file system housing the Streams installation of where the checkpointed state resides. This information is used at runtime to configure fault-tolerant workflows should a failure occur.

5.2.3. Debugging Service

An important runtime service used extensively by application developers and administrators is debugging. The model instance that describes an application and its operators carries debugging flags from the compiler invocation command line (as commonly done by other programming languages) alongside with directives specified in the source code, for example, instructing the runtime to either run a processing element under the C++ debugger \[19\], or

---

\[17\] To be more precise, moving an operator requires moving the processing element and its container from a host to another one.

\[18\] The launch count is used by the PE to inform its operators of how many times they have re-started, so on a restart an operator can potentially fetch the latest checkpointed state and re-populate its internal data structures after a fault. The launch count is also used by the job management component to keep track of intermittent failures and, eventually, mark the PE as having failed permanently.

\[19\] Streams provides its own debugger `sdb` \[31\], which is geared towards inspecting stream-related aspects of an application. The `sdb` debugger also integrates seamlessly with the GNU `gdb` debugger, allowing developers to use conventional debugging techniques with their application.
simply instrument and run a specific operator (and its processing element) by itself in a window displayed in the developer’s terminal, instead of in the background on a remote host.

Debug directives are represented in the application instance model in the form of PE wrappers, as shown in the excerpt in Figure 13. Guided by these directives, the runtime is instructed to launch the processing element process under specific debuggers (i.e., sdb in the depicted example, but gdb and other tools like valgrind are currently supported).

In this particular case, the model-based design helped integrating all the different components involved in debugging (from the compiler to the specific debuggers) via a common way of expressing where and how much debugging infrastructure should be put in place, seamlessly.

### 5.2.4. Visualization Services

Large, distributed applications are notoriously difficult to debug, optimize, and manage. Not only are they more complex due to their size, but their distributed execution model coupled with parallelization constructs and the occurrence of asynchronous events introduce additional challenges. To ameliorate this situation and to provide tooling that can display actionable information for application understanding, optimization, and debugging, a model representation of both the static aspects of an application (e.g., its operators and PEs) as well as the application’s dynamic aspects (e.g., on-demand stream connections, specific data and processing rates) are central to the architecture. Moreover, such information must include enough contextual data to provide the necessary insight regarding performance and correctness, dynamically, to aid the analysts as well as to pinpoint specific source code locations to direct developers in applying changes and performing modifications as a result of their analysis.

Streams provides extensive visualization capabilities, focused on reducing clutter and presenting information that is directed at identifying and solving problems. While we have discussed these tools in our previous work, their connection with the runtime system and the role that the different models play have not yet been discussed.

Not surprisingly, these tools make extensive use of the application instance model. Streams Studio, the Streams application development and visualization hub, is capable of depicting
most of the information available in the collection of application instance models obtained and dynamically updated on-demand by the runtime. It is also able to distill, summarize, and aggregate data from these model instances, providing different visualization perspectives, depending on the analytic task at hand. For example, the visualization can provide a bird’s eye view focusing on the interaction of multiple applications. It is also possible for analysts to drill down all the way to the specific traffic pattern between two operators.

The retrieval of live, runtime data, represented as augmented runtime information is carried out via one of the job management client interfaces, again expressed in Corba’s IDL and with objects whose structures are defined in XSD form, as was the case for all interfaces described so far. Figure 12 depicts an instance of the system topology model. This representation provides the most comprehensive snapshot of the runtime environment depicting the applications that might be concurrently running and interacting with one another. This model provides the raw information used to produce the visual representation of the overall data flow graph such as the one seen in Figure 14.

An interesting aspect related to visualization and other management interfaces is that the amount and level of detail to be retrieved can be configured via a scoping object, enabling the customization of what information should be in the instance model object and, therefore, be returned to the visualization system. For example, Figure 15 depicts a model that is used for creating views on the system state. In this case, the model is a selector that can be used to define what type of entities should be included when dynamically generating the instance object for the system topology instance model. This capability is important in providing incremental and localized updates, thus reducing the amount of data exchanged between visualization tools and the runtime management components.
Figure 15: One of the scoping models used for creating views on the system state (in XSD). The model provides options to globally retrieve the system state (line 3) or to select only a subset of it (lines 4-11), such as selecting only the PE information (line 10) or only the port connections (line 11).

As we have described in Section 3, flexible object view management was one of the goals of employing a model-based design. Object scoping coupled with a granular design of the objects managed by the middleware enabled us to provide that flexibility to the visualization as well as to other administrative tools.

6. Related Work

This work has several points of contact with multiple lines of research previously pursued by the community as well as with many existing technologies. As we have discussed in Section 3, a model-based design requires externalizing the interface and object models. A number of technologies have evolved throughout the years for developing component-based systems. Technologies such as Corba [22], DCOM [14], webservices [16], and several flavors of interface definition languages [46, 47, 55] have been proposed. We have found, however, that no one technology alone addressed all the requirements we had. From a non-sophisticated type system in Corba to the relatively low-performance of the communication protocol for webservice interfaces, in implementing System S and later Streams, we decided to combine the best parts of what these technologies had to offer.

The representation of models and, in some cases, of instance of objects expressed in those models was done using XML [17]. The models and the collections of models making up the system were expressed by XSD [29] descriptions. While an XSD-based object representation has limitations, for example, in imposing fine-tuned validation rules for attribute values, it provides a relatively sophisticated mechanism for establishing relationships between objects (parent-child and is-a) and also has the capability for managing textual documentation. More
importantly, however, is that there is excellent tooling for automatic code generation, including support for both C++ and Java.

An important contribution we discussed in this work was the mechanism for extending the SPL language and the associated mechanism for code generation. Programming language and code building frameworks have both contended with these issues. General-purpose programming languages usually allow extending existing classes via inheritance \cite{43} and/or the instantiation of type specific ones via generics/template programming \cite{13}. Inheritance has the drawback of runtime overhead for dispatching virtual methods, while generics are limited to compile-time only. In fact, these strategies are at both ends of the spectrum, while our approach provides a configurable middle ground as discussed in Section 4. On the application building infrastructure, a large set of tools have been developed over the years for build management. From the famous Bell Labs make program \cite{30,50}, to GNU’s autoconf \cite{41}, to the Apache Foundation’s ant \cite{35}. While we were inspired by all these tools, we have implemented a sophisticated mechanism to allow minimal code re-generation and re-build operations. Its basis, as we discussed in Section 4.2.3, is the identification of common generated code. We believe that this approach can also be employed by scripting language interpreters and other code generation systems.

Large-scale system design has been a long-running topic in the software engineering community, including even forums dedicated to discussing some of its unique issues \cite{19}. There is not, however, enough literature discussing the practical implementation issues of real systems as we did in this work. In particular, we were unable to find studies encompassing the engineering challenges found in building the full spectrum of components in complex software platforms: from tooling, to language, to the runtime system. In fact, many other comparable frameworks, including MPI, PVM, and Corba grew organically. System S, on the other hand, has been re-designed and re-implemented a few times with the latest iteration being entirely model-based. As described in this work, models provide the key architectural backbone for Streams and SPL.

As discussed, both the middleware and the language were designed to support high-performance, distributed stream processing applications. This area has enjoyed roughly one decade of research, including the sprouting of commercial systems such as StreamBase \cite{52} and Progress’ Apama \cite{9} built on ideas generated by a plethora of academic systems such as Aurora \cite{14}, Borealis \cite{2}, STREAM \cite{11}, among others. A number of programming languages have also sprung up, including StreamSQL \cite{53}, CQL \cite{12}, StreamIt \cite{56}, etc. To the best of our knowledge, and perhaps due to the broad scope of Streams, none of the existing middleware nor any of the existing languages employed a model-based approach as we did. Therefore, the tight architecture integration of tooling, runtime system, and language as well as the extensibility and sophisticated incremental building capabilities Streams has are quite unique.

7. Lessons Learned

Streams has been growing in capabilities and in complexity as reflected in the features made available in the several versions produced by IBM since the official product launch in 2008. Likewise, the development of applications from multiple engineering domains \cite{48,57} reflects
expanding programming language requirements. Both of these trends had been anticipated by the product architects, who wanted to ensure a smooth evolution of the code base and language.

In this work, we demonstrated the architectural basis to achieve this goal. In other words, how the design of its programming language as well as the architecture of the system were, from the outset, put in place such that they can be extended to accommodate new and evolving requirements, incrementally.

As discussed in Sections 4 and 5, the software implementing the language and runtime was rooted on a set of inter-related models meant to fully describe inter-component relationships. These models expose to architects and developers the internal structure of the system. They also provide the input to code generators that can automatically produce the interface and object classes expressed by them.

With a focus on extensibility, the SPL language and its compiler expose a model-driven framework primarily as a means to provide a high-degree of specialization when generating code, but also to reduce runtime overhead, to support an efficient build infrastructure for large applications, as well as to provide powerful language modularity capabilities.

In the process of developing Streams some lessons were learned along the way. Many of these lessons can be employed in other large-scale, multi-component software systems. Among them, we highlight the following:

- **Models should be designed by considering future evolution.** Models should be structured to enable additional extensions, while keeping backward compatibility. Although models help prevent architecture decay by fostering an extensible system design, models themselves are subject to decay if extensibility is not given sufficient consideration during model design. As an example, metrics reported by Streams support only a single data type, yet the relevant models are structured in such a way that a hierarchical type system can be supported in the future.

- **Models should be versioned.** Model versioning helps in a few areas. First and foremost, it serves as a life vest in case the backward compatibility of a model has to be broken due to a major overhaul. While such major modifications should not happen in a well-architected system, barring extensive changes in requirements, in the rare case that they are unavoidable the version numbers can be used to maintain system-level backward compatibility by updating the processing logic to inspect the model version and take appropriate action accordingly. Second, version numbers help with serviceability of the product, by clearly identifying the version of the model used without requiring additional knowledge external to the model. Last, version numbers can help with dependency resolution in the context of extensibility. As an example, the SPL language allows toolkit models to specify the minimal version of the Streams product required to support the toolkit. Model version numbers are used to enforce such dependency rules.

- **Model documentation should be updated in conjunction with the models themselves.** In practice, we found that the updates to the model documentation sometimes lag behind the updates to the model itself. This kind of documentation problems is common for infrastructure models as well as infrastructure source code, since the documentation for these are not end-user visible and the pressures of product development deadlines often
lead to developer oversights in this area. However, such oversights in documentation reduce productivity in the long run, especially when multiple developers and multiple components are involved. Build-time scripts that enforce completeness of model documentation as well as code reviews are effective ways of addressing this problem.

- **Model verbosity may hinder performance.** Examples include low network performance when exchanging models across machines (e.g., as part of the control flow among Streams services), disk performance when reading/writing models from/to non-volatile storage (e.g., as part of code generation and incremental compilation in SPL), database performance and field size limitations when managing models as part of a recovery database, etc. When performance is an important consideration, several techniques can be applied to reduce the model verbosity. These include, but are not limited to:

  - *Minimizing repetition via normalization.* This can be achieved by factoring out repeated segments into separate sections and using indexes to reference them. An example of this is file names in a model that contains debugging information, where the file names would otherwise be repeated a large number of times.
  
  - *Improving the object serialization method.* This can be achieved by using binary serialization formats and compression when necessary. For instance, some XML-to-object model compilers, including the one we employed [21], support binary serialization formats that are less verbose than XML. Alternatively, if sticking with the XML representation, compression can be used.

- **Clean separation of logical versus physical representation of modeled elements is important.** The logical representation of an object is what an end user typically sees in the source code or when employing a visualization tool. Conversely, the physical representation of an object is what the implementation uses internally. When this distinction is not clear in the models, details of the implementation may leak into the presentation layer. Furthermore, the lack of separating these two representations may result in information loss and implementation difficulty if the mapping between the logical and the physical representations are not maintained between the layers of the software system. For instance, composite operators in SPL do not have a physical representation. Instead, they are fully expanded at compile-time. Yet, if the mapping between composite operators and the resulting primitive operators is not available at run-time, presentation and implementation difficulties will arise in the visualization interfaces as well as in the debugging subsystem.

8. **Conclusion**

This paper describes the use of a model-based design in Streams, a large-scale stream processing platform. Settling on this approach was the result of multiple iterations in designing the middleware and runtime system as well as its programming language, compiler, and application building infrastructure.

We found that bringing together all these components has been made substantially easier by relying on external models that describe most of the system objects as was summarized
Specifically, the use of code generation for outputting interface stubs and object classes in C++ and Java provided software engineering efficiencies as well as the means for evolving both interfaces and objects via a simple recompilation step. Moreover, the models provide self-documentation, with fresh and up-to-date structural information available to software engineers and architects in the development team.

When it comes to the SPL programming language, the model-based framework provided the foundations for language extensibility, for supporting incremental building, and for tooling design going ranging from barebones debugging capabilities to sophisticated visualization interfaces.

In summary, the use of model-based design throughout the system have enabled us to create a software ecosystem that is simpler to document and maintain, while having conceptual uniformity in its code base, APIs, and user interfaces alike. While this paper provided mostly an extensive case study on large-scale system design for a stream processing platform, we believe that many of the techniques we discussed here can be leveraged elsewhere as several of the issues we tackled are not unique to our particular situation.

Acknowledgments: The implementation of System S and Spade and, later, Streams and SPL was done by a large team of developers, with the participation of multiple architects, designers, and project managers under the direction of Nagui Halim. The techniques and lessons described here are the product of numerous discussions and refinements with the larger team. Therefore, we acknowledge the collaboration of colleagues from multiple IBM sites, including Rochester (MN), Toronto (Canada), Silicon Valley (CA), Raleigh (NC), the China Research Lab, and from the Thomas J. Watson Research Center (NY).

REFERENCES


