Engineering a Converter between two Domain-Specific Languages for Sorting

Johan Fabry
Raincode Labs
Brussels, Belgium
johan@raincode.com

Ynès Jaradin
Raincode Labs
Brussels, Belgium
ynes@raincode.com

Aynel Gül
Raincode Labs
Brussels, Belgium
aynel@raincode.com

***Abstract*—Part of the ecosystem of applications running on mainframe computers is the DFSORT program. It is responsible for sorting and reformatting data (amongst other functionalities) and is configured by specifications written in a Domain-Specific Language (DSL). When migrating such sort workloads off of the mainframe, the SyncSort product is an attractive alternative. It is also configured by specifications written in a DSL but this language is structured in a radically different way. Whereas the DFSORT DSL uses an explicit fixed pipeline for processing, the SyncSort DSL does not. To allow DFSORT workloads to run on SyncSort we have therefore built a source-to-source translator from the DFSORT DSL to the SyncSort DSL. Our language converter performs abstract interpretation of the DFSORT specification, considering the different steps in the DFSORT pipeline at translation time. This is done by building a graph of objects and key to the construction of this graph is the reification of the records being sorted. In this paper we report on the design and implementation of the converter, describing how it treats the DFSORT pipeline. We also show how its design allowed for the straightforward implementation of unexpected changes in requirements for the generated output.**

***Index Terms*—Domain-Specific Languages, Language Conversion, Legacy Systems**

I. Introduction

Mainframe computers typically provide for a wide variety of programs as part of the standard installation. One such program is IBM's DFSORT [1], which is responsible for sorting and reformatting data (amongst other functionalities). The use of DFSORT can be a key part of a software system on the mainframe, complicating efforts to migrate such systems to more common software and hardware environments.

SyncSort is a historical competitor to DFSORT that runs on Windows and Unixes, with a similar feature set as DFSORT. Hence, in the context of a migration it can be used to handle the sort aspect of a software system. Migration of the sort then consists of translating the specifications how how sorting should be performed. DFSORT is configured by specifications written in a Domain-Specific Language (DSL) [2] [3] and SyncSort is also configured using a DSL. However, the two DSLs are radically different and transformation of DFSORT configurations to their SyncSort equivalent is not evident, as we show later in this paper. In our experience, there typically are thousands of sort configurations to be migrated, so manually translating them is not advisable.

To allow DFSORT workloads to be migrated to a Syncsort installation, Raincode Labs has developed a language converter [4] between the DFSORT DSL and the SyncSort DSL. In its current version, it treats files for different Raincode Labs clients, in total translating around 33.000 files successfully. The design of the language converter is a three-phase process of a parser, a model builder and a code generator. Notably, the model builder performs abstract interpretation to recreate the steps in the DFSORT processing pipeline, support for which is missing in the SyncSort DSL.

In this paper we report on the design and implementation of a language converter that translates programs in the DFSORT configuration DSL to their equivalent in the SyncSort configuration DSL. As far as we know, this is the only translator for this domain and this is the first time that the design of an industrial language converter for DSLs is presented in scientific literature (no reported DSL translators [2] [3] [5] have a DSL as target). We detail how the DFSORT processing pipeline is followed by the translator and how this makes it possible to construct models of equivalent SyncSort configurations. We also show how reuse inside the translator is enabled by following the DFSORT processing pipeline. Lastly we illustrate how we were able straightforwardly accommodate change requests for the output by modifying the code generation phase of the translation. In our experience, the three-phase process of a parser, a model builder and a code generator has shown to be an outstanding design for this language converter.

In this paper, we will first describe DFSORT, its processing pipeline and DSL in Section II. We then talk about SyncSort and its DSL in Section III. Section IV describes the design and implementation of the translator, and Section V talks about requirement changes. Section VI then concludes the text.

II. Mainframe Sort

On the mainframe, file handling is radically different from how it is done on more modern operating systems. For one, (almost) all files on the mainframe are considered as a list of records. Even plain text files are a list of records consisting of 80 bytes, where each record is initialized with spaces[1] Thanks to this structured approach of files, the operating system can

[1] Text file line size on the mainframe is always 80 columns.

provide for more file handling utilities. One of these is the sort utility, called DFSORT [1].

As can be expected, the main purpose of DFSORT is to sort a file: it takes a list of keys and sorts all records in the order defined in those keys (ascending or descending). Keys are defined by giving an offset in the record, a length for the key in bytes, its datatype and a sort order. DFSORT can sort character fields as well as the different binary data formats available on the mainframe. In addition to sorting, DFSORT can merge different files into one file, join the records of two different files into records of one file, and even produce reports for a file. A discussion of all these features is out of the scope of this paper, instead here we will focus on sorting.

To configure the behavior of DFSORT, the user provides a list of "DFSORT program control statements". This is essentially a declarative description of how the sort should be performed, written in a Domain-Specific Language (DSL) [2] [3] In this text, we call such sort configurations *sort programs*. We omit a detailed description of the DSL here, as the IBM manual [6] chapter that describes it is 400 pages long.

```
SORT FIELDS=(42,1,PD,D,2,20,CH,A),EQUALS
```

Fig. 1. A basic sort program.

A simple example sort program is given in Figure 1. It states that the sort should happen with primary key being the binary data in Packed Decimal format at byte 42, in descending order, and secondary key being the characters from byte 2 to 22, in ascending order. Moreover, the sort should be stable, i.e. equal-keyed records in the input must be in the same order in the output.

But the sort is more than just a simple sorting of records. The internal flow of a sort operation is described as a pipeline of 14 different stages [6], and in this text we will treat 7 of these stages. They are as follows:

1) Include / Omit: When records are read, they can be filtered by including or omitting them based on conditions on fields. These conditions allow fields to be compared with each other or with constants, and conditions can be compounded. For example, consider the code in Figure 2: if the two characters at position 24 are equal to the character constant JF or the value at position 344 is different from -1, the record will be omitted.

```
OMIT COND=(24,2,CH,EQ,C'JF',OR,
           344,1,PD,NE,-1)
```

Fig. 2. An OMIT with a compound condition.

2) Inrec: Before records are passed to the sort, they can be modified in five different ways. For brevity, we only discuss three of them here. Firstly, a new record can be built by combining parts of the original record and/or constants. For example:

```
INREC BUILD=(1,20,CABC,5C*,
             15,3,PD,EDIT=(TTT.TT))
```

This builds a new record that consists of the first 20 bytes of the original record, the character string ABC, five times the * character, and the three Packed Decimal bytes at position 15 that are converted to a printable number with 3 characters before the dot and two after the dot.

Secondly, data can be overlaid onto the record, effectively replacing only the part of the record as specified. For example:

```
INREC OVERLAY=(20:X40,60:20,1)
```

The above statement alters the record such that the hexadecimal value 40 (an EBCDIC space) is put at position 20 and the byte that was at position 20 is put at position 60.

Thirdly, the above two modifications can be conditionally performed to, for example, to compute an absolute value:

```
INREC IFTHEN=(WHEN=(20,5,PD,LT,0),
         OVERLAY=(20:(20,5,MUL,-1,
                    TO=PD,LENGTH=5)))
```

If the indicated value is less than 0, it is multiplied by -1, the result is cast to a packed decimal of length 5 and overlaid at the same position.

Multiple IFTHEN clauses can be specified, and for each clause it can be decided if INREC processing for this record should continue or stop.

3) Sort: This is where the actual sort happens. Note that since the records that are sorted are those leaving the INREC stage, sort keys relate to the record produced by INREC.

4) Outrec: Once records have been sorted, they can again be transformed. Essentially these are the same transformations as in INREC, but here a record can be constructed that does not contain the keys, since the sort already happened.

5) Outfil: DFSORT reads its data from a standard input stream and writes it to a standard output stream, both configured by the user when calling DFSORT. The DFSORT program control statements can however also specify multiple output streams, and the records will be written to each of these OUTFIL streams instead of the standard output stream. Each of these streams can also be individually modified, as follows:

6) Outfil Include: This is a filter as in step 1, but it filters only data going out to this file. This allows, for example, to distribute the output records to different files, depending on the value of a part of the record, i.e., a control field.

7) Outfil Outrec: This is the same functionality as the record creation in steps 2 and 4, but it constructs the record for this specific output. This allows, for example, to remove the control field in the output of the previous example.

III. SyncSort

SyncSort is a historical competitor to DFSORT, available on the mainframe since the "early 1970's" [7]. It was built to outperform DFSORT and supports most of the features of DFSORT, including, e.g., reporting. It is considered to be "the first non-IBM product in an lot of customer sites" [7] and runs on Windows and Unixes since 2004. As such, it is an outstanding product to be considered for migrating sort workloads from the mainframe to such systems.

The SyncSort program is configured, analogously to DFSORT, in a sort program written in a DSL. But the problem for a migration is that this DSL is fundamentally different from the DFSORT DSL. Whereas the previous DSL maps straightforwardly to the processing pipeline of the sort, this one does not map to such a pipeline. Instead, the SyncSort language mainly talks in terms of operations on fields. Due to legal requirements, we cannot explain the specifics of this DSL here. Instead we talk about the keywords of the language that are relevant to this text using an invented syntax.

There are around 60 keywords in the language. We will only describe 8 of them here:

1) *Def_Field:* In this DSL, fields are given a name (using the typical syntax allowed for identifiers), offset, length and datatype. One Def_Field may define multiple fields this way, and there may be multiple Def_Field definitions in the program. Note that the field offset considers the record as read from the input stream, as there is no sort pipeline as in DFSORT. For example, consider defining the same two fields as in Figure. 2, naming them Id and Stat respectively. This is done as follows:

```
Def_Field Id(24,2,CHAR), Stat(344,1,PD).
```

2) *Def_Calc:* Syncsort can perform a wide variety of calculations on fields and constants. This keyword defines a name for a 'field' that is calculated by evaluating an expression. Allowed expressions are field names, constants, arithmetic expressions, if-expressions (with conditionals as described below), data type conversions, multiple kinds of string operations, and more.

3) *SortKeys:* The order and direction of sort keys is defined by referring to the names of fields or calculated values. For example, a primary key Id, sorted ascending and a secondary key Stat, sorted descending, is specified as follows:

```
SortKeys Id(A), Stat(D).
```

4) *Def_Cond:* Conditions are given a name and an expression that refers to field names, calculations or constants. For example, below is the equivalent to the condition of Figure 2, with ID and Stat defined as above.

```
Def_Cond OCond(ID=="JF" || Stat!=-1).
```

5) *Def_Output:* This is equivalent to stage 5 in the DFSORT pipeline: describing which files on disk the records should be written to. In contrast to DFSORT however, there should be at least one such definition.

6) *Keep, Remove:* Records can be filtered out of the incoming stream of records to sort, or out of the stream of records that are written to disk. These keywords take one single argument: the name of the condition for inclusion or omission of the record. The location of the keyword in the program has a meaning: if it appears before the first output file definition, the filter is as in phase 1 of the DFSORT pipeline (Include / Omit), while if it appears after an output file definition, it is as in phase 6 of the DFSORT pipeline (Outfile Include).

7) *Def_Record:* When writing to disk, new records can be constructed out of fields, calculations and constants. The new record basically consists of the concatenation of named fields, named calculations or constants. Again the location of the keyword has meaning: it applies to the most recently defined output file.

From this list of keywords, it is apparent that converting a sort program in the DFSORT format to the Syncsort format is not guaranteed to be a straightforward task. Simple sort programs as in Figure 1 are easy to translate. In the SyncSort syntax it could be as in Figure 3.

```
Def_Field Primary(42,1,PD).
Def_Field Secondary(2,20,CHAR).
SortKeys Primary(D), Secondary(A).
Def_Output <omitted>.
```

Fig. 3. The equivalent of Figure 1 in the Syncsort DSL.

However, when DFSORT record transformations in INREC or in the different OUTREC are specified, it can become complicated to establish, respectively, the values for the sort keys, or what is being written to disk. Consider for example the simple sort program of Figure 4.

```
INREC BUILD=(1,8,24,2,9,15,26,14)
SORT FIELDS=(1,20,CH,A)
```

Fig. 4. A DFSORT sort program with an INREC.

The SyncSort equivalent sort program in Figure 5 requires us to define five sort fields, a calculated field concatenating three of them to be used as a single sort key and an output record that is the concatenation of four fields:

```
Def_Field F1(1,8), F2(24,2), F3(9,15),
          F4(26,14), F5(9,10).
Def_Calc Cc1 F1,F2,F5.
SortKeys Cc1(A).
Def_Output <omitted>.
Def_Record F1,F2,F3,F4.
```

Fig. 5. The equivalent of Figure 4 in the Syncsort DSL.

When IFTHEN is used in the DFSORT specifications, constructing the equivalent for SyncSort quickly becomes quite extensive and complex. Due to size constraints, we do not include such an example here.

In brief, transformation of the definitions of sort program from DFSORT to SyncSort is not evident. Moreover, in our experience, sort workloads typically contain thousands of sort programs. Therefore, manually translating these programs as part of the migration project is not recommended, as it will take a large amount of time and the chance of errors in the translation is significant. Consequently, while the SyncSort product at first glance seems suitable for allowing DFSORT workloads to be migrated off of the mainframe, the difference in configuration DSLs poses a substantial hurdle for such migrations.

IV. The SyncSort Translator

To allow DFSORT workloads to be migrated off of the mainframe onto a SyncSort installation, Raincode Labs has developed a source-to-source translator for sort programs. This is essentially a language converter [4] between the DFSORT DSL and the SyncSort DSL. To the best of our knowledge, this is the only translator for this domain. Moreover, as far as we are aware, this text is also the first time that the design of a language converter from a DSL to a DSL is presented in scientific literature [2] [3] [5].

In this section we report on the design and implementation of the translator. The goal of the translator is explicitly **not** to cover all features of the DFSORT DSL. Instead we have a corpus of programs of multiple clients of Raincode Labs that needs to be adequately covered (the exact details of which are confidential). We identified groups of language features present in the corpus and developed the translation in steps: we implemented support group by group so that in each step the number of translatable programs increased.

The translator is written in C# and is a three-phase process consisting of a parser, a model builder and a code generator:

A. DFSORT *Configuration Parser*

The first step of the translator is parsing the sort program. Raincode Labs has developed an in-house parser generator that is used to specify the parser for DFSORT programs. The parser generator is a variant of PEG [8] with various improvements such as packrat parsing [9]. A discussion of the parser generator is out of the scope of this text, and we cannot divulge the full grammar for the DFSORT parser for commercial reasons. Suffice it to say that the grammar is around 400 lines of code and it contains 86 non-terminals.

The parser produces a DFSORT configuration: an abstract syntax tree (AST) of the sort program that is subdivided in five main parts that mirror the structure of the DFSORT pipeline:

1) Include: The include or omit filter of the pipeline is kept as a tree of condition objects. Primitive conditions include comparing between fields and comparing fields to constants, where fields are instances of a Field class and constants instances of a Constant class. Composite conditions (conjunction and disjunction) are binary. Sort programs can only contain an INCLUDE orn OMIT condition, not both, so we only keep a condition for include. If the program specifies an OMIT condition it is negated, which produces a new condition tree (instead of bluntly wrapping the tree in a negation operator).

2) Inrec: Arguably the biggest part of the grammar treats the transformation of records as used in INREC and both OUTREC. Parsing such transformations yields a Transformation object. Put simply, such an object is a list of fragments where the transformed record is the concatenation of the list of fragments. Each fragment hence describes a transformation to one specific field, e.g., an arithmetic operation, a datatype and length change, a constant, and so on. All fragments are kept as objects, and each kind of fragment is a different subclass of a common ancestor.

3) Sort: The keys of a sort are kept as a simple list of SortField objects. SortField is a subclass of the Field class mentioned above, adding one extra instance variable for the direction to sort.

4) Outrec: The specification of the transformation of the (sorted) results of the Inrec phase are kept as the Outrec AST. This is also an instance of the Transformation class as described previously, hence seamlessly reusing all the parser logic for transformations.

5) Outfil: The list of OUTFIL statements, if any, is kept here. Basically, each object in this list wraps an include condition and outrec transformation. Both of these seamlessly reuse the parser logic for building the conditions and transformations, respectively.

B. *SyncSort Model Builder*

The second phase in the translator produces a model for the SyncSort sort program that will be produced in the next phase. The SyncSort model builder in effect performs abstract interpretation of the DFSORT sort program, considering the different steps in the DFSORT pipeline at translation time. The model that results is a graph of objects and key to the construction of this graph is the reification of records. Records are objects that are aware of their internal structure as a concatenation of fields and are the sole entity responsible for the creation of fields.

The builder proceeds through the different steps of the pipeline as follows:

1) Include: Starting from a blank input record, the builder constructs an include condition tree, where references to fields are resolved by the input record. Since it is blank, creation of fields corresponds to simply creating field objects with offset, length and type as given in the sort program.

2) Inrec: If an INREC has been defined, the builder creates a new record by first treating the different fragments of the transformation. Each of these are converted to their SyncSort equivalent. Broadly speaking, for each DFSORT kind of transformation fragment (recall that these are distinct C# classes), the builder creates the equivalent SyncSort counterpart as a graph of (different) C# classes. After all fragments are treated, a new record is then instantiated with contents the concatenation of these different fragments. If no INREC has been defined, the result of this step is the original input record.

For simple BUILD transformations that do not manipulate the underlying fields as in Figure 4, a fragment corresponds to the creation of a field object by the input record created in step 1. More complex transformations, such as IFTHEN, usually result in a Def_Calc where the expression of the calculation is a graph of concepts of the SyncSort DSL, e.g., for IFTHEN an if-expression would be used.

This part of the translator is arguably the most intricate, mirroring the complexity of the transformation of records in the grammar of the parser. In the translator, there are 30 classes whose major responsibility is modeling a record transformation fragment in terms of the features offered by SyncSort.

3) Sort: A sequence of sort keys is then built by asking the record produced in step 2 to provide field objects for the keys defined in the DFSORT program.

For example, consider the translation of Figure 4, shown in Figure 5, We have one sort key with offset 1 and length 20. The record is a concatenation of `F1`, `F2`, `F3`, `F4`. When requested for a field with offset 1 and length 20, the record responds with a new concatenation `Cc1` that contains `F1`, `F2`, and a newly created `F5`. The creation of `F5` was necessary because `F3` is 5 bytes too long. Hence `F5` was created with the same offset as `F3` but with length 10 instead of 15.

4) Outrec: If an `OUTREC` has been defined in the `DFSORT` sort program, the builder creates a new record based on the record produced in step 2. It effectively reuses the implementation of step 2. If there is no `OUTREC`, the output record is set to the record produced in step 2.

If there is no `OUTFIL`, the building of the model stops. An example of this is Figure 4, which yields a model that can be described by considering Figure 5. There are two entry points in this graph of objects: the list of keys as built in step 3, and the final record. We can see in Figure 5 that the keys refer to one concatenation `Cc1` that, in turn, refers to the fields `F1`, `F2` and `F5`. The output record refers to `F1` through `F4`, completing the graph.

5) Outfil: For each file stream defined in `OUTFIL`, the builder constructs an include condition as in step 1, if any, and an outrec record as in step 4, if any. The logic for this effectively reuses the implementation of these respective steps, each of them taking as input record the record of step 4.

C. Code Generator

The role of the Code Generator is to serialize the graph of objects that is constructed by the model builder, producing a SyncSort sort program.

Fundamentally, each object in the graph knows how to serialize itself as a SyncSort representation on a stream of strings. SyncSort constructs that are named, such as fields, conditions and computed values, can serialize themselves in two different ways. Firstly a definition and secondly a reference. When these objects are created by the model builder, their name is left empty. The first time that their reference is requested they will first serialize their definition, giving themselves a unique name, before returning their name. Serialization then amounts to a traversal of the graph, as implemented by these constructs.

The code generator produces a program by traversing the graph from three different entry points, as we show next.

1) Sort Keys: The first traversal starts with the definition of the sort keys. It amounts to serializing each key field by asking each key field for its name. In effect, this is a post-order traversal of the graph since it triggers each key field to first serialize its definition. In case the key field is a calculated value, this causes the expression to be serialized, typically causing the fields and conditions in the expression to be serialized. For example, consider Figure 6, which is the equivalent of Figure 4, as produced by the translator. The first five lines show the result of code generation for the sort keys. The fifth line is the specification of the keys, which first caused the definition of the calculated value `Cc4` on the fourth line, which required the first three lines to define the fields of the concatenation.

```
Def_Field F1(1,8).
Def_Field F2(24,2).
Def_Field F3(9,10).
Def_Calc Cc4 F1,F2,F3.
SortKeys Cc4(A).
Def_Output <omitted>.
Def_Field F5(9,15).
Def_Field F6(26,14).
Def_Record F1,F2,F5,F6.
```

Fig. 6. The equivalent of Figure 4, as produced by the translator.

2) Include: After serializing the sort keys, the include condition is serialized. It is serialized in post-order in the same fashion as sort keys are serialized.

3) Output file and Record: Last but not least, the code generator iterates over the list of output files and serializes the record that corresponds to the output file. If the output record is the original input record, no code is produced, since it would be superfluous to define it. If an output record was constructed (either through an `INREC` or `OUTREC`), this record is serialized in post-order in the same fashion as before. An example of this is shown in Figure 6, where the last line defines the output record. It requires the definition of two new fields `F5` and `F6`, which are defined in the two preceding lines.

Conclusion: The implementation of the language converter is a three-phase process of a parser, a model builder and a code generator. The parser creates an AST and the builder creates a model from it by performing abstract interpretation of the steps in the `DFSORT` pipeline. In this model, the record abstraction is responsible for the creation of field objects, enabling the mapping of field references to their their computed value or their location in the original input. The code generator then performs a post-order traversal of this graph, starting from three different entry points.

V. The Translator in Practice

The SyncSort translator is currently being tested by clients, with its first use in production planned before the end of 2020. The percentage of files that have been successfully translated varies per client. In our test corpus of almost 35.000 files we can translate over 95% of the files successfully. In some client corpuses we have been able to translate over 97% of the files. Note that a coverage of over 95% is typically acceptable for the clients. This is because it can be more cost-effective to manually translate untranslatable files (to translatable files or to equivalent SyncSort files) than to pay for support of the features that are missing in the translator.

However, the fact that a program produces correct output for a large set of inputs does not necessarily imply that it is well-engineered. In contrast, what arguably is a good token of a well-engineered program, is how gracefully changes in requirements can be handled. We have had two such changes during development, and we talk about them here.

A. Requirements change: Filename Placeholders

The original concept of the language converter was to keep the sort programs in the DFSORT language and have the converter translate on the fly whenever a sort is performed. All calls to the DFSORT executable would be replaced by a script that first transforms the programs and then calls SyncSort, passing it the output of the translation. The downside of translating on-the-fly is the overhead of the translation. Over time this will accumulate a significant cost in execution time and memory, translating to a higher monetary cost for the server performing the sort. To overcome this issue a client requested to do an offline translation: translate DFSORT sort programs to SyncSort sort programs that are kept for later execution. The outputs, however, need to be templates because parts of the program may change between sort executions. These parts are placeholders in the template. Then, instead of running the language converter on-the-fly before each sort, a preprocessor would replace the placeholders as needed, which is computationally much less intensive. Placeholders are needed as, for example, the sort may be part of a larger batch process where input and output files can be created by the batch, and hence their path differs between runs.

Generating output with such templates needs for all sort steps to also execute, since the output of one sort process can be the input of a next sort process. Hence, the translation must be integrated into a 'normal' translation. This turns out to be quite straightforward requiring only two changes to the converter. Firstly, all entities in the model that need to produce template output check the 'template mode' flag and, if set, produce output with placeholders as needed. Secondly, if 'template mode' is enabled an extra code generation pass is performed after normal code generation: the code generator resets the names of all named constructs (see IV-C) and performs generation again. This produces the additional template output.

B. Requirements change: New SyncSort Syntax

With a target language being a DSL for a product that exists for more than 40 years on the mainframe and 15 years on contemporary platforms, an arguably reasonable expectation would be for the language to be quite stable. This is however not the case. A request was made by a client to also provide support for a new Syncsort DSL, called DTL. The main reason for this was DTL's ability to allow the sort to run more efficiently in some cases. Unfortunately, DTL is not backward compatible with the old language and furthermore does not provide support for a number of features available in the old language, e.g. reporting.

Fortunately, a study of DTL revealed that the fundamental underlying concepts remain the same (if they are supported by DTL). The most drastic difference in DTL is that fields can no longer be defined in separate `Def_Field` statements. Instead they should be aggregated in one `Def_Layout` statement. Therefore the implementation of a DTL code generator is sufficient to allow us to produce sort programs in the DTL syntax. Furthermore, since the SyncSort executable still has support for the old syntax, when producing DTL output fails because of lack of support in DTL, we can fall back on the code generator for the old syntax to generate a program.

The DTL code generator differs from the code generator for the older syntax in two ways: firstly all entities that have a different concrete syntax in DTL serialize them in the DTL syntax when needed. Secondly, recall that by traversing the graph the code generator forces the different fields to serialize definitions for themselves when they are first referred (see IV-C). Instead, the DTL code generator collects all these definitions and then adds the aggregate definition when the graph has been fully traversed.

VI. Conclusion

In this paper we reported on the design and implementation of the Raincode Labs language converter from the DFSORT DSL to the SyncSort DSL. As far as we know, it is the only translator for this domain, and the first time the design and implementation of an industrial language converter is published in scientific literature.

The language converter is a three-phase process of a parser, a model builder and a code generator. The model builder performs abstract interpretation to recreate the DFSORT processing pipeline and key to this recreation is the responsibility of record objects for the creation of field objects. We have shown how following the pipeline in the construction process allows for significant reuse in the translator, and how the model permits straightforward implementation of change requests for different kinds of output. In our experience, it is this design that made implementing the language converter possible.

References

[1] "DFSORT product documentation," IBM Corporation. [Online]. Available: http://www.ibm.com/storage/dfsort

[2] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000. [Online]. Available: https://doi.org/10.1145/352029.352035

[3] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005. [Online]. Available: https://doi.org/10.1145/1118890.1118892

[4] A. A. Terekhov and C. Verhoef, "The realities of language conversions," *IEEE Software*, vol. 17, no. 6, pp. 111–124, 2000. [Online]. Available: https://doi.org/10.1109/52.895180

[5] J. Fabry, T. Dinkelaker, J. Noyé, and É. Tanter, "A taxonomy of domain-specific aspect languages," *ACM Comput. Surv.*, vol. 47, no. 3, pp. 40:1–40:44, 2015. [Online]. Available: https://doi.org/10.1145/2685028

[6] *z/OS DFSORT Application Programming Guide*, IBM Corporation, Poughkeepsie, NY, USA, 2019.

[7] L. Johnson, "Oral history of Duane Whitlow," Computer History Museum, Tech. Rep. X5710.2010, 1998.

[8] B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, N. D. Jones and X. Leroy, Eds. ACM, 2004, pp. 111–122. [Online]. Available: https://doi.org/10.1145/964001.964011

[9] ——, "Packrat parsing: : simple, powerful, lazy, linear time, functional pearl," in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, M. Wand and S. L. P. Jones, Eds. ACM, 2002, pp. 36–47. [Online]. Available: https://doi.org/10.1145/581478.581483