EVALUATION OF DATA STREAM MANAGEMENT SYSTEMS IN THE CONTEXT OF SPACE SURVEILLANCE AND TRACKING

S. Müller, C. Kebschull, and E. Stoll

Institute of Space Systems, Technische Universität Braunschweig, 38108 Braunschweig, Germany,
Email: {sv.mueller, c.kebschull, e.stoll}@tu-braunschweig.de

ABSTRACT

Space debris nowadays is widely recognized as a potentially severe problem for (future) space flight activities. Because of this, surveying orbit regimes via sensors connected to a processing system aimed at creating and maintaining a catalogue of all detected objects automatically is thoroughly researched (Space Surveillance and Tracking, SST). In this context Institute of Space Systems, Technische Universität Braunschweig, develops simulators for such processing systems, one approach being Datastream Management Systems (DSMS). Just like conventional database management systems DSMS cope with huge amounts of data and provide standardized and optimized means for data storing and retrieving. Unlike conventional database management systems DSMS don't store data tuples perminently, but instead continuously expect incoming data streams and provide outgoing data streams while providing standardized and optimized means for controlling data flow and processing. Sensor data streams may, for example, be branched to distribute computation load on different processors and/or machines, joined to collect data from different sensor sources, filtered to eliminate "bad" sensor data, aggregated to calculate minima/maxima values etc. Processing may be modified without recompiling and, in many DSMS, during run-time. Whenever new data tuples arrive, processing is kick-started (data-driven processing). Overall, DSMS have been explicitely developed with dynamic, adaptable, real-time processing of and reaction on high-rate continuous input of data in mind making them an ideal candidate for usage in SST systems.

This paper evaluates a DSMS in the context of a radar system simulator (RSS). For this purpose one basic functionality of a RSS is implemented and executed: Initial Orbit Determination, i. e. a first estimation of an object's orbit based on – in this case: simulated – measurements. It is described what steps have been taken to achieve development of a prototype and what challenges arose. Results of a general test case are presented and discussed. A benchmark test case follows where the prototype is put under stress by sensor data at a realistic rate.

Key words: SST, DSMS, space debris, data bases, data streams, IOD.

1. INTRODUCTION

Modern space environment usage requires an accurate and up-to-date knowledge of as many objects on Earth's orbits as possible. Space Surveillance and Tracking (SST) describes the effort to achieve this by surveying space via ground-based sensors and by collecting and processing the measurements taken at a central place in order to create and maintain a catalogue of all detected objects and their orbits. Such a SST system must be able to cope with sensor data from multiple sensors at high rates. It must be able to task sensors, to schedule the processing of all input data while refreshing the existing information right on time. Because of this, technologies and algorithms for SST systems are a promising subject of research. Institute of Space Systems (IRAS) at TU Braunschweig follows the approach of simulating a SST system specialized for connecting radar sensors: Radar System Simulator (RSS). RSS utilizes a relational database in which all measurement, object and control data is stored. One software tool named PROCOR handles the control data, which is used by other software tools to determine their tasks on the data. This way, processing is coordinated. RSS is detailed in section 3.

This approach, though very promising, can be counted as conventional. The relational database is stored in secondary memory. Despite the trend to use Solide-State-Drives (SSDs) for this kind of computer memory, the factor between their speed and commonly used primary memory like DDR3 RAM reigns at about 4 – with the successor DDR4 being available already. As a result, in SST context, it seems wise to rely more on primary than on secondary memory.

Additionally, in RSS, the data flow is mainly controlled by the separate software program PROCOR and by the way the other seperate software programs interpret the control data in the data base. Changing data flow thus may involve reconfiguring or even recompiling these programs. A more flexible way to adjust data flow could be necessary to face future challanges.

Another point is that conventional SST systems must implement some kind of ”polling” for new data at many places in order to establish a continuous processing. This leads to delays in processing because the polling time intervals set at in one situation may be unsufficient for another situation. A more adaptable way would be advisable in the light of SST systems, since their system load may vary significantly.

Because of all this, another approach researched at IRAS involves a rather new kind of data processing technology designated as Data Stream Management Systems (DSMS). In these variants of common data bank systems, a query to the system results in so-called operators, sources and sinks being created and interlinked. This way, the sources, operators and sinks provide continuous data processing chains which indefinitly wait for new input data and automatically kick-start processing as soon as new data elements arrive (called Data-Driven Processing). The processing is done in primary memory only. This way, fast ”data streams” run from one station in the system to the next allowing fast reaction times. A scheduler which ensures the timely processing of all data is an inherent part of DSMS. Section 2 gives a short overview of the current DSMS environment.

For this approach, a DSMS called STREAM is evaluated for usage in SST context. The system is elaborated upon in section 4.

Two test cases have been defined in order to make first evaluations. The first test case is a general one which targets executing an Initial Orbit Determination (IOD) on simulated radar measurement data. The radar data is produced by a RSS program called Messwertgenerator (MWG, ”Measurement Generator”, cp. section 3). The second test case is a benchmark test case targeting IOD execution with the same kind of data as before, but with a realistic timing for each data element. Several runs are defined with varying data rate of the measurement data. Metrics are defined to allow judging the outcome of these test case runs on the test system. All of this is explained in section 5.

Some work had to be done to make STREAM fit for the execution of these test cases. Most importantly, the RSS program which actually executes IOD had to be interfaced and STREAM had to be extended to allow implementation of an IOD operator. Section 6 gives an overview of these modifications.

Section 7 shows the results of the test case executions and discusses them. Section 8 gives a conclusion.

2. RELATED WORK

A wide range of DSMS are available. They can, for example, be divided into commercial and free ones. Commercial distributors are, among others, IBM (InfoSphere Streams [1]), Microsoft (StreamInsight [2]), Oracle (Oracle Stream Analytics [3]) and AT&T (GigaScope [4]). Freely available systems are usually results of projects at research institutions, e. g. the universities of Berkley (TelegraphCQ [5]) and Stanford (STREAM [6]) as well as MIT (Borealis [7]). Some of the commercial DSMS are based on freely available ones.

Another line can be drawn in regards to code availability of these systems. As a rule, this boundary is generally the same as between commercial and freely available systems. The open-source nature of most ”research DSMS” was one of the main factors for selecting STREAM for our research, even though the project from which it emerged from ended in 2006[1]. This is, however, common in research because the availability of project finances is often essential.

Programming languages used for these systems are normally C (e. g. TelegraphCQ), C++ (e. g. STREAM [8]) and Java (e. g. Odysseus [9]).

Contrary to common relational data bank systems, there is no standardized query language to communicate with these systems (yet). As a result, interoperability between the systems is not ensured. However, the languages used generally are derived from Standard Query Language (SQL) – the standard which is used for common data base systems. Some query languages become more and more popular. Most notably, these are StreamSQL and Continuous Query Language (CQL).

The technical approaches of DSMS vary radically. For example, TelegraphCQ is based on the common database system PostgreSQL, which leads to a rather seamless connection between common databases and DSMS. On the other hand, STREAM, for example, has been explicitly designed as a DSMS from the start leading to an easier and more complete integration of new DSMS concepts as well as a more ”lean” system.

STREAM does not go ”all the way through” with the specialization, though. In fact, the idea was to keep most relational methods like tables, table joins, projection, selection etc. intact. Other systems like Borealis (including its predecessor Aurora) turn away from these principles. Borealis also uses a more graphical approach, while other systems, STREAM included, are heavily based on formal languages and formal correctness [10].

Current research in the field is done, for example, towards distributed DSMS, which are able to scale better with input data load by distributing the processing onto different machines. Also, so-called Probabilistic Data Stream Management Systems (PDSMSs) are a subject of current research. These systems are specifically designed to cope with uncertain data sets.

[1] http://infolab.stanford.edu/stream/ (Obtained 2017-04-10)

3. RADAR SYSTEM SIMULATOR

As part of a DLR-funded project, a Radar System Simulator (RSS) has been developed at IRAS. Its purpose is to evaluate the impact a given radar sensor has on Space Situational Awareness (SSA) activities, like building and maintaining an object catalogue, orbit determination accuracy and more. The simulation environment consists of several tools:

- MWG (Measurement Generator),
- SMART (Sophisticated Module for Analyzing Radar Tracklets),
- CAT (Catalogue Analysis Tool),
- PROCOR (Process Coordinator) and
- CAMP (Catalogue Maintenance and Pass Prediction)

The Measurement Generator uses a given population, which can be made of TLE objects or an artificial population based on the MASTER-2009 Space Debris environment [11]. Using a numerical propagator NEPTUNE (NPI Ephemeris Propagation and Uncertainty Extrapolation), the population is moved forward in time [12]. A Radar Performance Model (RPM) is used to simulate the different kinds of radars. The RPM requires the knowledge of the position of the objects on a microsecond time scale. Reflector and array antennas can be simulated. The RPM is supplied by Fraunhofer FHR. The simulation will return tracklets of objects that passed the beam and have been marked as detected. Tracklets may contain multiple measurements of the same object, giving information about azimuth, elevation, range as well as range rate, signal-to-noise ratio and the probability of detection. The tracklets are further processed by the PROCOR tool. It analyses new tracklets and distributes them to SMART processors that can perform orbit determination (OD). A SMART instance pre-processes a given tracklet and determines, which OD method to use. For initial OD the methods Gibbs and Herrick-Gibbs are used as described in [13]. For statistical OD the methods Weighted Least Squares (WLS), Extended Kalman Filter (EKF) and an Unscented Kalman Filter (UKF) are available. The created or updated ephemeris are stored in a database. The tool CAMP will overlook the database and update the states as needed using the numerical propagator NEPTUNE. The implemented pass prediction will provide forecasts of passes over a given sensor location. This can either be passed back to MWG or to a real sensor. Once the entire process chain is running, CAT can monitor the buildup of the catalogue or determine the individual object orbit accuracy over time. All tools have been implemented in FORTRAN using the 2003 standard. Each tool connects to a Postgres database via FORTRAN-C bridge.

4. STREAM – A DATA STREAM MANAGEMENT SYSTEM

STanford sTReam datA Manager (STREAM) [6][2], developed by Stanford University, is a freely-available open-source Data Stream Management System (DSMS). It has been written in C++ [8] and is composed of a main library, a Generic Client and a Dedicated Server. The main library contains all main functionality, which is used and extended by the Generic Client and the Dedicated Server in order to provide a console tool and a stand-alone server. A separate package contains a GUI client, written in Java, that is able to connect to the Dedicated Server.

The Generic Client works mainly with text files. It processes a script upon start in order to read queries. Queries tell a DSMS how to adjust its processing flow. STREAM uses a Continuous Query Language (CQL) variant which lets users express nearly everything that can be expressed in most other CQL variants [10]. An example for a STREAM Generic Client CQL script is shown in listing 1.

The ”table” and ”source” statements at the top tell STREAM to create a new data stream which gets its input from a text file called ”numbers.txt”. This data stream consists of data elements which each contain an integer value for an attribute called ”some_number”. The Generic Client only supports text files for reading in and writing out data elements, but STREAM’s Dedicated Server makes it possible to also receive them from or send them over network.

The first ”vquery” and ”vtable” statements perform a CQL query on the number data stream. A 5-second so-called time window is opened in which all numbers bigger than 100 of that stream are stored – as long as they arrived maximal 5 seconds ago. This window mechanism is common among DSMS and allows creating a ”view” on a supposedly never-stopping data stream. This view is updated over time as new data elements arrive. In STREAM, such a view can either be another stream or a ”relation” which is the case here. Just like in common (relational) data base systems, relations can be thought of as tables with each row containing one data element. The attributes of the relation can be thought of as columns of the table. In the background, this relation is actually realized by a data stream of relational updates. However, users that write CQL scripts don’t have to bother with this fact and are able to ”think” in tables. Furthermore, users of common database systems know Standard Query Language (SQL) which CQL is based on. Relations are also familiar to them. The ”leap” towards STREAM relations is, compared to other DSMS, easy.

The first ”query” and ”dest” statements convert the relation into a stream again by defining that a new element is emitted, when a new element is inserted into the previously-defined relation. This stream thus makes it

[2] http://infolab.stanford.edu/stream/ (Obtained 2017-04-10)

Listing 1. Example STREAM Generic Client script. Lines have been indented for clarity. All indented lines must actually be on the same line as the next unindented line above it.

```
table: register stream
         numbers_stream
         (
           some_number integer
         );
source: numbers.txt

vquery: select some_number
        from numbers_stream
          [range 5 seconds]
        where some_number > 100;
vtable: register relation
          current_big_numbers_relation
          (
            big_number integer
          );

query: istream
       (
         select *
         from
           big_numbers_count_relation
       );
dest: new_big_numbers.txt

query: dstream
       (
         select *
         from
           big_numbers_count_relation
       );
dest: old_big_numbers.txt
```

possible to react on the occurence of new big numbers at the time they arrive at the system. This stream is written to the text file "new_big_numbers.txt" which may be read by another program.

The second "query" and "dest" statements do a similar task. In this case, the big numbers are written to "old_big_numbers.txt", as soon as they drop from the 5-second time window.

The term "time" of DSMS normally differs from actual time. STREAM, for example, does not really know about the actual progress of time [14]. The aspect of time is injected into the system by adding a timestamp attribute to each input data element. STREAM processes data elements as fast as it can without bothering about the timestamps – except that data elements need to arrive at the system in timestamp order. It is up to users of STREAM to decide about the timing. As soon as STREAM receives a data element with timestamp τ, commonly counted in seconds, it assumes that time progressed from $\tau - 1$.

The CQL script shown is parsed and interpreted in five phases resulting in five different internal representations (cp. figure 1). The final representation consists of interconnected operators. The script, for example, would be translated to a source operator which emits new stream elements from the text file, a window operator creating a view on the stream, a select operator filtering numbers bigger than 100, an istream and a dstream operator emitting stream elements and two output operators writing stream elements to files (cp. figure 2).

All operators have input and/or output queues for elements that are not processed by the operator yet and for elements that have just been processed and are due to be processed by the next operators in the chain.

Like all DSMS, STREAM includes a scheduler which is responsible for coordinating all operators. The STREAM version used for this paper contains a round-robin scheduler. This is a simple scheduler that cycles through all operators and lets them execute a certain maximal number of elements (100000) one after another.

STREAM's Generic Client is single-core meaning that it distributes the time of one processor over the different operators.

5. TEST CASES SETUP

The test cases data consists of one tracklet containing detections of Envisat by TIRA (Tracking and Imaging RAdar, Germany), simulated by MWG (cp. section 3). The detections span is about 822 seconds on 2013-11-22. They contain, among others, timestamp, azimuth, elevation, range and the respective rates. An overview of the data is given in figure 3.

Each detection also contains a tracklet ID. Furthermore, it is assumed that all detections arrive at the system in the

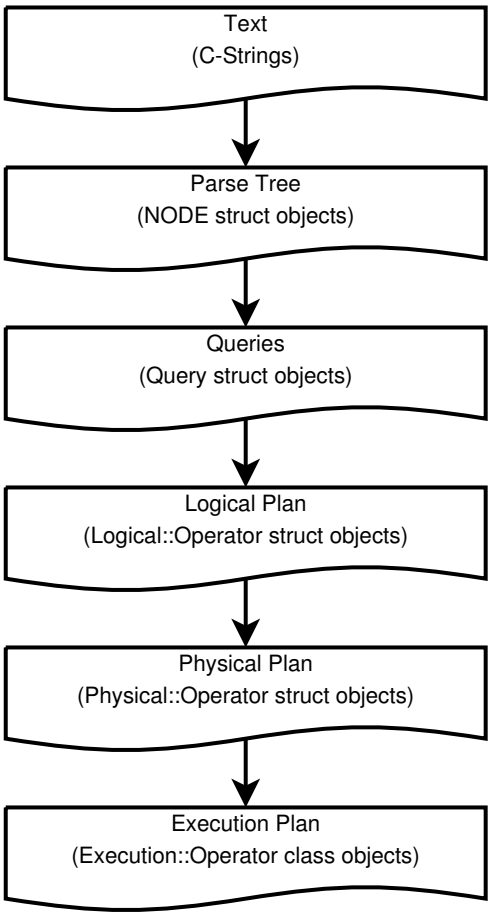

Figure 1. STREAM CQL script parsing and interpretation. This process involves five steps resulting in different internal representations up to execution operators.

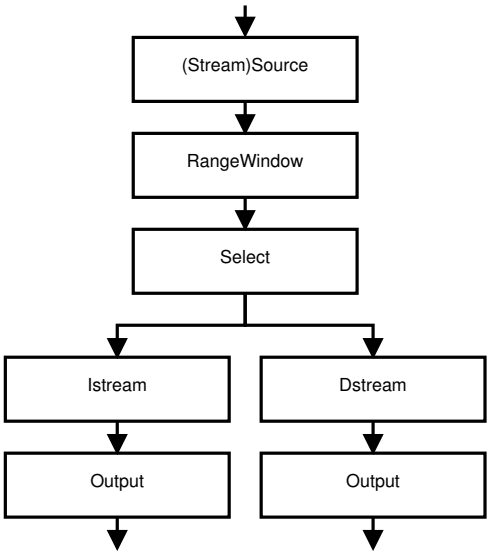

Figure 2. Example CQL script from listing 1, translated by STREAM into execution operators. The arrows indicate data element flows from one operator to another.

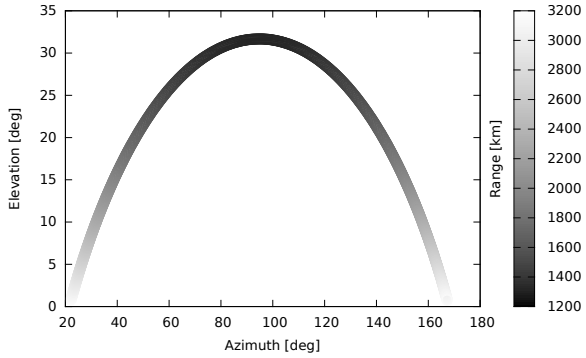

Figure 3. Radar measurements of Envisat by TIRA as simulated by MWG. The measurements span over 822 seconds on 2013-11-22 and are used for the general test case.

correct order concerning the time of their recording and that one tracklet is sent after another. This way, a tracklet can be determined ”complete” when a detection arrives at the system with a different tracklet ID than the detection before that.

The target of the general test case is to ensure that the setup is able to execute an IOD based on the tracklet successfully. The test case execution is considered successful, if the orbit data output by the system matches Envisat's orbit data. The tracklet contains 822 detections, i. e. the detection rate is 1 per second. The tracklet is put into the system as a whole meaning that all detections are input at the same time (but in the correct order). So, in this case, there is no relationship between the detection timestamp and the time at which each detection is inserted into STREAM. The tracklet is only put into the system once which is why the tracklet ID of processed detections does not change. In order to allow this one tracklet to be determined complete at the end, one additional ”dummy” detection with a different tracklet ID is sent as the last detection.

The target of the benchmark test case is to set the system under stress using a realistic detection rate. This target includes the following:

- Put the tracklet into the system using a clocked timing.
- Put the tracklet into the system three times directly after another.
- Increase the input data element rate ι, counted in input data elements per second (ips), stepwise until a realistic value is reached.

Unlike in the general test case, the tracklet is put into the system with a clocked timing. This means that each detection is put into the system at the right time offset. The timestamp of the first detection serves as zero-point for the data and the start of the actual execution of

STREAM's operators (after initialization) serves as the zero-point in STREAM execution. It should be noted that the granularity of this timing is only one second. However, this granularity is enough for benchmark purposes.

Additionally, the tracklet is put into the system three times in a row in order to simulate a long-during tracking session. The run-time of the system is set to $\tau_{\text{max}} = 1800$ seconds which allows two consecutive IODs plus some time buffer.

Lastly, the benchmark test case is executed multiple times starting with a run in which the tracklet of the general test case (but excluding the dummy detection) is set as input for STREAM. The next run is executed using a similar tracklet as before, but with $\iota_{i+1} = \iota_i + o\Delta_{i+1}$, where $i = 0..n$ is the run number, n the number of the last run, o the orientation of the input data rate change, Δ the input data rate change, $\iota_0 = 1$ and $\Delta_0 = \frac{1}{2}$. At first, $\Delta_{i+1} = 2\Delta_i$ and $o = 1$. As soon as the system is not able to cope with the data rate anymore, $\Delta_{i+1} = \frac{\Delta_i}{2}$ and $o = -1$. From this point on, $\Delta_{i+1} = \frac{\Delta_i}{2}$ always applies. As soon as the system is able to cope with the data rate again, $o = 1$. o keeps changing together with the ability to cope with ι_i. The algorithm ends with $\Delta_n = 1$ and $i \neq 0$.

The data-rate-change approach leads to $\iota_5 = 32$ (if the system is able to cope with $\iota_0..\iota_4$) which is roughly the detection rate of a radar sensor for a tracking of Envisat. The intension of this approach is to both fullfill the targets of the test case and to find an estimate for the input data rate which the system can cope with.

The ability of the system to handle ι is determined based on four metrics. The first two concern the operator input and output queue loads. $Q_{\text{max}}(\tau)$ is the maximal queue load in percent at timestamp τ. $\bar{Q}(\tau)$ describes the average queue load in percent at timestamp τ. Both values are applied to all maximal queue loads in $]\tau - 1, \tau]$. Ideally, $Q_{\text{max}}(\tau)$ should never exceed 100, but it is possible for the system to recover, if the value is lower for a long-enough period of time. $\bar{Q}(\tau)$ makes it able to judge if the system only "jams" in a few places or overall.

The third metric is $L(O, \tau)$ which denotes the "lag" of operator O at timestamp τ, i. e. the difference between τ, the actual current time(stamp), and the timestamp of the last element emitted by O. Ideally, $L(O, \tau) = 0$ in all cases, of course, but a small lag can always be expected as the detections have to be propagated through the system one after another. A bigger lag may also be tolerable, but this depends on the scenario.

Contrary to the other metrics, which are used for analysis, the fourth one is specific to the IOD scenario and used to decide if the system is able to cope with the current data rate. It is the idea that the system shall be able to deliver an orbit derived from the detections without too much delay, meaning that, after a whole tracklet arrives at the system, the delay until an orbit is output shall be lower than a certain threshold. Furthermore, the delay shall remain constant for both tracklets that are inserted completely into the system. In the following, the delays are called L_{IOD_0} and L_{IOD_1}. The threshold value for the benchmark test case is set to $L_{\text{max}} = 30$ and the threshold timestamp to $\tau_{L_{\text{max}}} = \tau_{\text{max}} = 1800$ is defined.

In order to ensure that IOD still functions as in the general test case, the orbit data is compared to Envisat's orbit data.

The computer system on which the test cases have been executed had no CPU-time or memory-consuming tasks running at the time of test case execution. The computer had the following specifications:

- Processor: Intel(R) Core(TM) i7-2640M
  - Base Frequency: 2.80 GHz
  - Max. Frequency: 3.50 GHz
  - Number of Cores: 4
  - Number of Physical Cores: 2
  - L2 Cache: 4096 KB
- Memory
  - Type: PC3-10600 Non-Parity (NP) Double Data Rate Three (DDR3) Technology
  - Size: 7886 MiB
  - Peak Transfer Rate: 10666.7 MB/s
- Swap: 15466 MiB
- Operating System: Debian GNU/Linux 8.7 (jessie)
- Word length: 64-bit

6. IOD IMPLEMENTATION

STREAM v0.6 has been chosen for a prototype IOD implementation (cp. section 4). For this purpose, several modifications had to be done.

Lots of the modifications aimed at making STREAM easier to develop based on current developer tools and versions. Some of the changes were rather small – e. g. adding "#include" statements necessary due to GCC updates –, some were rather big – e. g. changing the build system to CMake and supporting 64-bit floating point attributes. These changes, however, shall not be the topic here.

The most important modification was to build an environment in which new operators (cp. section 4) could be easily added to STREAM in a modular way. The problem was that an operator was defined and created at several different places in the code. This, of course, is a result of the STREAM script parsing and interpretation steps (cp. section 4 and figure 1).

In the first data structure transition (from text to a parse tree), for example, the `parser` subsystem of STREAM

Listing 2. Grammar rule in STREAM `parser` subsystem to recognize the two binary (requiring two parameters) set operators `union` (∪) and `except` (−).

```
binary_op
   : T_STRING RW_UNION T_STRING
     {$$ = union_node ($1, $3);}
   | T_STRING RW_EXCEPT T_STRING
     {$$ = except_node ($1, $3);}
   ;
```

operates on parsing rules. One of these rules is shown in listing 2 and is intended to recognize the two binary (requiring two parameters) set operators `union` (∪) and `except` (−). If, at the place in the CQL script where it is allowed to use a binary operator (represented here by `binary_op`), a string is followed by the `union` keyword which then is followed by another string, a `union` operator is recognized and the two strings are given to the function `union_node(...)`. `union_node(...)` creates and fills a union-kind `NODE` struct and adds the struct to the parse tree (the first internal representation of the parsed script). The `union`-specific contents of that node are only the two strings containing the two table names connected via `union` in the CQL script. If two strings are connected via `except` keyword at the place where the two binary operators may be used in the script, a similar processing occurs. At this level, just a small part of the creation of a specific operator is done. However, it is essential: Recognizing which of these two binary operators a user wants to use. This happens in the `parser` subsystem of STREAM.

Several steps later in the process, during the last data structure transition (from a physical to an execution plan, cp. figure 1) a C++ class `PlanManager`, which is part of the `metadata` subsystem of STREAM, is told to `instantiate` the `Execution::Operators`. The `PlanManager` then executes serveral steps to do this. One of these steps is to equip the execution operators with input and output queues. Figure 4 shows how this is done for the `union` operator.

Both examples above show important steps of `union` operator definition (recognition of the operator's keyword) and creation (setting an operator's input and output queues), but they take place in completely different subsystems of STREAM. Many other steps are involved, spread over more parts of the system. Because of this, adding a new operator quickly was rather complicated.

The implemented solution introduces a new operator named `processing`. This one operator serves as a base class for an operator family `processing x` where `x` is the processing operator type (IOD, for example). There is now only one (new) subsystem called `processing_operators` which has to be modified in order to define and create a new operator. Its main constituents are shown in figure 5. A `ProcessingOperatorPool` holds an arbitrary

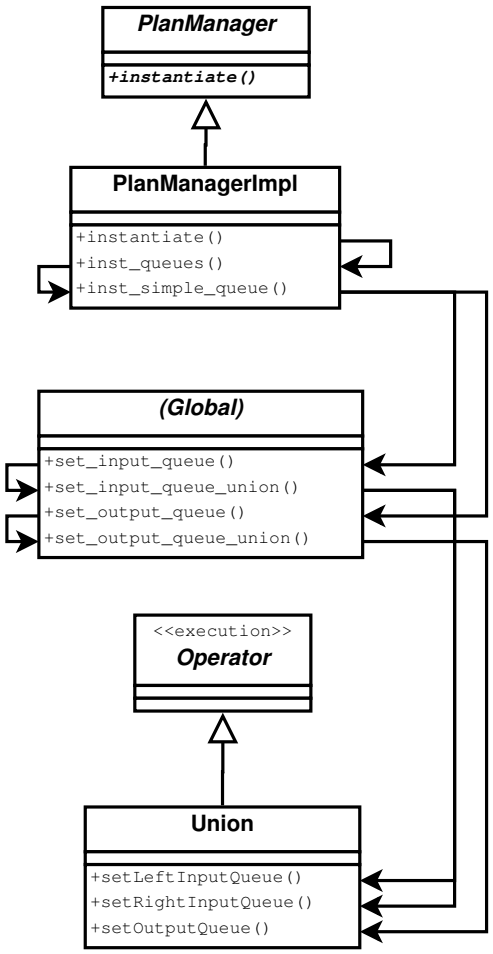

Figure 4. C++ classes and functions of STREAM `metadata` subsystem responsible for setting `union` operator's input and output queues. `PlanManager`, `PlanManagerImpl`, `Operator` and `Union` are C++ classes. Directly beneath the class names the functions of the respective class are listed. The two arrows with unfilled ends mean "is derived from". The arrows with the filled ends mean "calls".

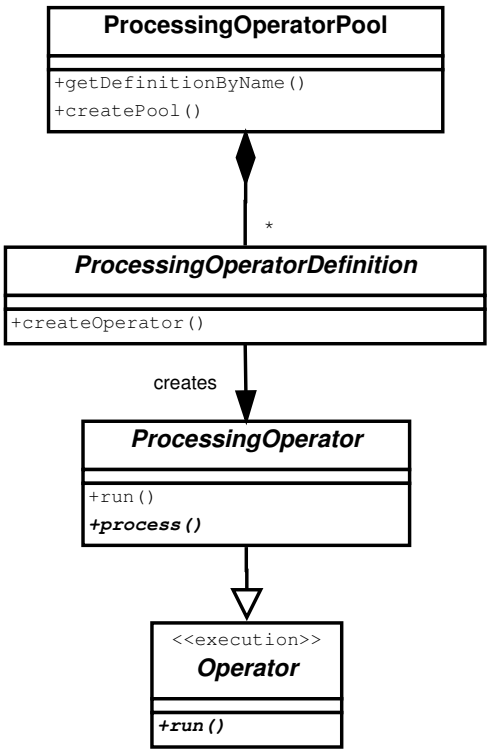

Figure 5. Main C++ classes of the new `processing_operators` *subsystem. The notation resembles the one of figure 4. However, the arrow with the filled end here means "creates" and the line beginning with a diamond means "contains an arbitrary amount of".*

amount of `ProcessingOperatorDefinitions`. Each definition is capable of creating `ProcessingOperators`. These operators are derived from `Execution::Operators`. A new operator can now be easily added by creating a new definition class and a new corresponding operator class based on `ProcessingOperatorDefinition` and `ProcessingOperator`. Lastly, the new definition and operator are connected to STREAM by adding one code line to `createPool()` of `ProcessingOperatorPool` in which a definition instance is added to the pool.

The `ProcessingOperatorDefinition` class also defines the keyword with which the processing operator is called in CQL scripts. An example is shown in listing 3 where `exampleoperator` is used to process the input attribute `example_input_attr` in order to produce the two output attributes `example_output_attr0` and `example_output_attr1`. Currently, input and output schema (the ordered list of attribute names and types) of a processing operator have to be identical which is why two dummy input values are given as input. The output value produced for `example_input_attr` is invalid in this case.

Formally, a processing operator has the signature

Listing 3. Example Generic Client script section using an example processing operator `exampleoperator`*. Lines have been indented for clarity. All indented lines must actually be on the same line as the next unindented line above it.*

```
vquery:
  select processing exampleoperator
    example_input_attr,
    42.0,
    42.0d
  from example_input_relation;
vtable:
  register relation
    example_output_relation
    (
      example_input_attr integer,
      example_output_attr0 float,
      example_output_attr1 double
    );
```

$$Relation \rightarrow Relation$$

which is the case for many STREAM operators [8]. This means that an input relation is processed and an output relation is delivered as output. However, in order to define processing operators correctly, a more detailed signature must be provided, since attribute number and types are strictly specific for them. The signature for `exampleoperator`, formally named Ex, for "Example", would be:

$$Ex : I \times F \times D \rightarrow I \times F \times D$$

or

$$Ex : I \times F \times D$$

for short where I, F and D describe the sets of possible values for 32-bit integers, 32-bit floating point numbers and 64-bit floating point numbers (further sets are C and S describing single characters and strings). In order to describe the attribute semantics, a general mapping must be given, as well:

$$Ex(in_{num0}, a_F, a_D) = (a_I, out_{num1}, out_{num2})$$

with "a" for arbitrary. Using both notations together, it becomes visible that `exampleoperator` has a schema of three attributes, the first one being of type integer, the second of float and the third of double. The first attribute is an input attribute named `num0`, while the other two attributes are used as output attributes named `num1` and `num2`.

With the processing operator mechanism in place, an IOD processing operator was created. For this purpose, an `EcsdIodOperator` class has been derived from `ProcessingOperator` class which can be used in CQL scripts by writing `processing ecsdiod`. The name was chosen as such, rather than `IodOperator`,

because other IOD operators may be included in the future, apart from the operator presented for this European Conference on Space Debris (ECSD) paper.

Formally, using the notation introduced before, `EcsdIodOperator`, named IOD_{ECSD} is defined as

$$\begin{aligned} IOD_{ECSD}: \quad & I \times I \\ & \times D \\ & \times F \times F \times F \\ & \times F \times F \times F \\ & \times F \times F \\ & \times I \times F \times I \\ & \times F \times F \times F \\ & \times F \times F \times F \end{aligned}$$

with

$$\begin{aligned} & IOD_{ECSD}(in_{tracklet\_id}, in_{detection\_id}, \\ & \quad in_{mjd}, \\ & \quad in_{range}, in_{azimuth}, in_{elevation}, \\ & \quad in_{range\_rate}, in_{azimuth\_rate}, in_{elevation\_rate}, \\ & \quad in_{RCS}, in_{SNR}, \\ & \quad in_{integrations}, in_{prob}, in_{score}, \\ & \quad a_F, a_F, a_F, \\ & \quad a_F, a_F, a_F) \\ & = (a_I, a_I, \\ & \quad out_{mjd}, \\ & \quad a_F, a_F, a_F, \\ & \quad a_F, a_F, a_F, \\ & \quad a_F, a_F \\ & \quad a_I, a_F, a_I, \\ & \quad out_x, out_y, out_z, \\ & \quad out_{v_x}, out_{v_y}, out_{v_z}) \end{aligned}$$

IOD_{ECSD} operates on "complete" tracklets. A tracklet is said to be complete, if one can reasonably assume that no more detections will be added to the tracklet in the future. In our scenario (cp. section 5), this is the case, when a detection with a $tracklet\_id$ arrives at the system that differs from the detection that arrived directly before that. So, the operator executes IOD as soon as a $tracklet\_id$ change takes place. To achieve an IOD, IOD_{ECSD} has been outfitted to execute RSS' SMART (cp. section 3) via system call.

SMART has been wrapped. The original code stayed untouched except that the FORTRAN programming construct `program smart`, which ontains the main SMART execution code, was converted into a `subroutine` named `smart_run_default` within a new `module` called `smart`. Based on SMART's execution arguments, the wrapping code decides to execute IOD in a STREAM context or to execute the original code residing now in `smart_run_default`.

The communication between STREAM and SMART is, for this prototype, realized via text files. The operator writes the detections of a tracklet into a text file as soon as it assumes that it received all detections of the tracklet. This decision is based on the tracklet ID of which a change will trigger writing the text file. SMART, executed via system call, reads the file, executes IOD and writes the resulting ephemeris into another text file which is then read by `EcsdIodOperator`.

The Generic Client (cp. section 4) is used for the execution of the test cases. In order to achieve this, the `FileSource` class of Generic Client was equipped with a "timing mode" and a "repeat mode" that could be switched on and off. The timing mode makes the `FileSource` read data elements from the input text file only if the timestamp of the data elements equals the time since the start of STREAM operator execution. The repeat mode makes `FileSource` read the data elements of the text file again (with increased timestamps and different tracklet IDs), after the end of the file is reached. Both modes are deactivated for the general test case and activated for the benchmark test case.

A decision had to be made concerning the timing mode. If `FileSource` is executed "too late", data elements from the input text file with a timestamp less than the time since STREAM operator execution start may be either ignored or inserted into the system together with data elements with the current timestamp. Ignoring these elements would simulate a dropping of data elements that can't be processed at the current time, inserting them would simulate a buffering of incoming elements until they can be processed. Both approaches may be seen as legitimate and it depends on what system actually should be simulated. An analogy could be drawn to the sending of data via a network. It is possible to send data packets, for example, via the so-called UDP protocol which does not guarantee the arrival of all data packets. But it is also possible to use the so-called TCP protocol which does make this guarantee. The design of the system which should be simulated is unknown here. Because of this, the buffering approach has been selected without a special reason.

The Generic Client script used for the test cases translates to four operators (cp. figure 6). A source operator reads the tracklet input file. This input file has been created beforehand by a script which converts MWG output files to a format the Generic Client `FileSource` can process. Contrary to the example in section 4, there is no window operator necessary, since the script makes use of an "unbounded" time window meaning that all elements that streamed in until the current time are being looked at. For an actual application of STREAM, this should be changed, but for the test cases, the time between the first

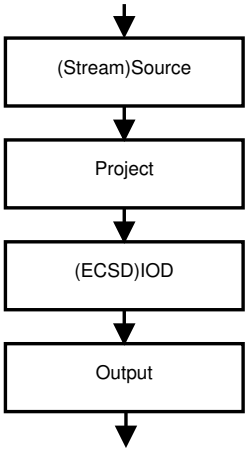

Figure 6. Generic Client script used for the execution of the test cases, translated into STREAM operators. The arrows indicate data element flow from one operator to another.

detection of a tracklet and the last one is known (about 822 seconds). And since all detections of a tracklet shall be used, no time limit is necessary. The project operator is responsible for adding dummy values to the attributes which are only used as output attributes. The IOD operator executes IOD and the output operator writes the resulting ephemeris (the orbit representation output by SMART) into a text file.

Because of the operator chain resulting from the CQL script, the scenario-specific metric L_{IOD} (cp. section 5) can be expressed as $L_{\text{IOD}_0} = L(\text{Output}, \tau_{\text{tracklet 0}})$ and $L_{\text{IOD}_1} = L(\text{Output}, \tau_{\text{tracklet 1}})$ with $\tau_{\text{tracklet 0}}$ and $\tau_{\text{tracklet 1}}$ being the timestamps at which the first and the second tracklet are completely input into the system.

7. RESULTS

The execution of the general test case produced the following state vector:

$$r = \begin{bmatrix} -6299.2412109375\ km \\ -3384.61645507812\ km \\ -80.8650131225586\ km \end{bmatrix}$$

$$v = \begin{bmatrix} 0.0268006287515163\ \frac{km}{s} \\ 0.161169663071632\ \frac{km}{s} \\ -7.42095041275024\ \frac{km}{s} \end{bmatrix}$$

The values translate to an orbital altitude of roughly 773 km and a velocity of 7.4 $\frac{km}{s}$. Hence, they correspond to Envisat's current orbit data and IOD functionality has been successfully connected to STREAM. Execution time was 2 seconds.

16 benchmark test case runs have been executed after that. The first run with $\iota_0 = 1$ ips, as expected, produced the same state vector as in the general test case.

The performance results are shown in figure 7. The queue loads are very promising, as both average and maximal queue loads are practically 0, which means that there is no "element jam" in the system. The operator lags of the source and project operators have values of 0 or 1 over the whole simulation time – which should be expected: On the one hand, the queues are empty most of the time resulting in practically no lag. On the other hand, a small time is necessary for reading the input text file and propagating detections from one operator to another, resulting in a small lag. IOD and output operator lags increase steadily, since IOD does not "fire" after each incoming detection, but after a complete tracklet has been received. As soon as this is the case, IOD is executed and the resulting ephemeris element is output. This results in a very steep decrease of the lag. After that, the detections of the second tracklet begin to arrive at IOD operator, starting the cycle again. These results are as expected. Interestingly though, IOD and output operator lags never become 0 or 1 again, but 2 directly after both IOD executions. However, this can be explained with IOD execution supposedly requiring an extra second execution time directly after the last detection of a tracklet has arrived. The overall IOD lag L_{IOD} is 2 seconds for both input tracklets which is way beneath the threshold of 30 seconds. So, this is an acceptable result.

Due to the stated benchmark test case algorithm and very similar results to the run with 1 ips, the input rates of the following runs were $\iota_{1..5} = (2, 4, 8, 16, 32)$. In these runs, a blip in both queue loads becomes more and more visible directly after the first tracklet has been input into the system. The performance results of run 5 (32 ips) are given in figure 8 to show this. The data shows that the load first decreases and then suddenly increases above the normal level, before it decreases to the normal level again. The reason for this is assumed to lie in two factors:

- The source operator simulates a buffering of the detections that arrive at the system.
- IOD operator "blocks" the system because of SMART execution.

IOD operator executes SMART synchronously which means that STREAM execution is halted as long as SMART is running, since Generic Client only utilizes one CPU core. Furthermore, execution between STREAM and SMART at this time is done via text files, which is a rather slow way of communication. Because of this, the source operator does not get the opportunity to inject new detections into the system for a small amount of time. As soon as it is called again, more buffered elements than normal are emitted.

The operator lags show that the source operator lags a little bit more than before which can be explained by the fact that a magnitude more detections than before need

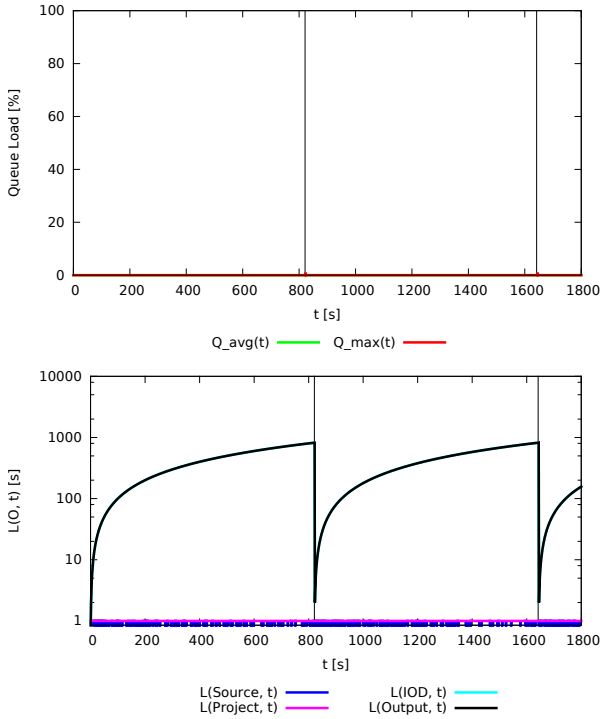

Figure 7. Results for benchmark test case run 0 (1 ips). Top: Average and maximum relative queue load of the system over execution time. Bottom: Operator lag over execution time. The lower boundary of $L(O, \tau)$ shown is 0.8 to make a change between a lag of 0 and 1 visible without taking up too much space in the plot. The two black vertical lines mark the two timestamps at which the tracklet has been completely input into the system ($\tau_{tracklet\ 0}$ and $\tau_{tracklet\ 1}$).

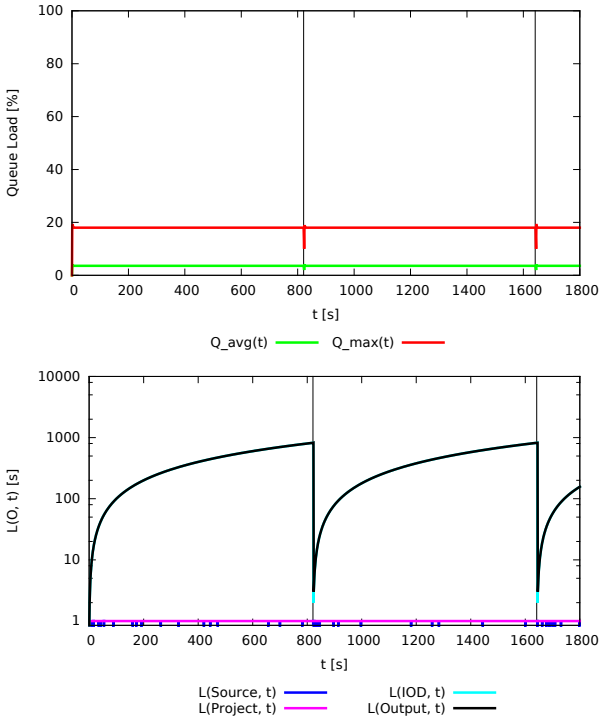

Figure 8. Results for benchmark test case run 5 (32 ips). Similar plots to the ones in figure 7.

to be processed by all operators (26282 instead of 822) which requires each operator to use more CPU time than before. Another thing that becomes visible is the difference between IOD and output operator lags directly after IOD execution. IOD lag improves, of course, a little bit earlier than output lag, since IOD comes before output in the processing chain.

In runs 1 to 5 the state vector remains, overall, the same. It changes only by very small amounts due the different number of detections. This was the case for all the following test case runs which is why the state vector is not mentioned in the following anymore.

Runs 1 to 5 furthermore show only a small increase in overall IOD lag from 2 to 3 seconds. In all these runs, the lag remains constant. This means that a realistic $\iota \approx 30$ ips case for a radar in tracking mode states no problem for executing IOD with STREAM.

Runs 6 and 7 were executed with 64 and 128 ips. At 128 ips the system reaches its limits for the first time, alas only for a small amount of time (cp. figure 9). In fact, the data shows that this is only the case for one second. Namely, the maximal queue load reaches 100 %. The reasons for this should be the same as for the blips mentioned before. Only, this time, at least one queue becomes full which leads to a short rise in source and project lags. The overall IOD lag reaches 9 seconds, but remains constant.

With run 8 and 256 ips the system is not able to cope with the input data rate anymore (cp. figure 10). $Q_{\max}$ becomes

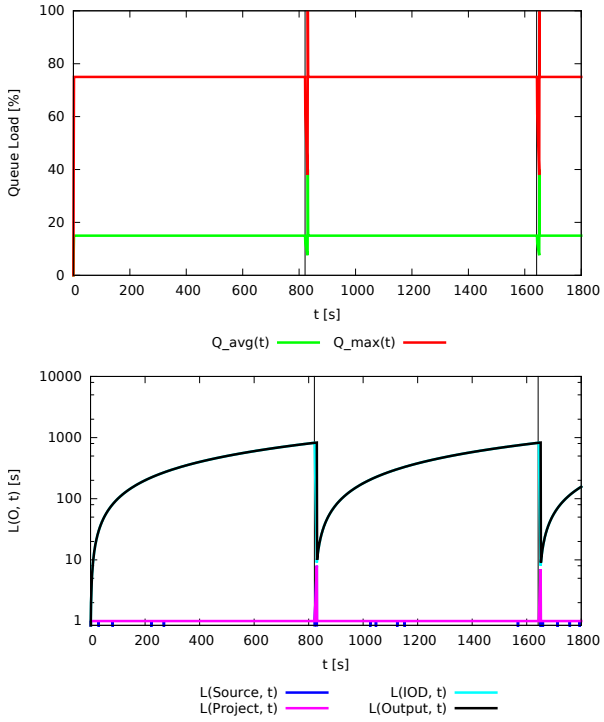

Figure 9. Results for benchmark test case run 7 (128 ips). Similar plots to the ones in figure 7.

100% at once, $\bar{Q}$ 30%. Interestingly, the operator lags show that the system is able to handle this kind of input rate until $\tau = 253$, correlating with an increase in $\bar{Q}$ at $\tau = 251$ to 40% after a series of small local increases to 30.2%. An easy explanation would be that ι changes over time. However, this is not the case. Currently, this aspect of the results can't be explained.

Operator lags also show that the system is unlikely to recover after one tracklet has been processed. Only L_{IOD_0} could be obtained in test case run execution time. It amounted to 649 seconds which is why ips were reduced for the next test case run for the first time.

Thus, run 9 was executed with 192 ips (cp. figure 11). The maximum queue load stays at 100% over the whole execution time. The sudden $\bar{Q}$, $L(\text{Source}, \tau)$ and $L(\text{Project}, \tau)$ increases are visible again, but the time at which the increases happen is with $\tau = 448$ later.

What's interesting here is also that, despite at least one of the queues being full the whole time, the source and project operator lags completely recover after IOD execution. What's more, L_{IOD_1} (which has been obtained through a longer execution time than normal) is roughly the same as L_{IOD_0} (164 and 163). This was not expected and means that, even though the system lags significantly, it seems "stable": It can be assumed that the system is able to handle this kind of input rate over longer times without, in the long run, falling behind.

This effect even becomes more apparent with run 10 us-

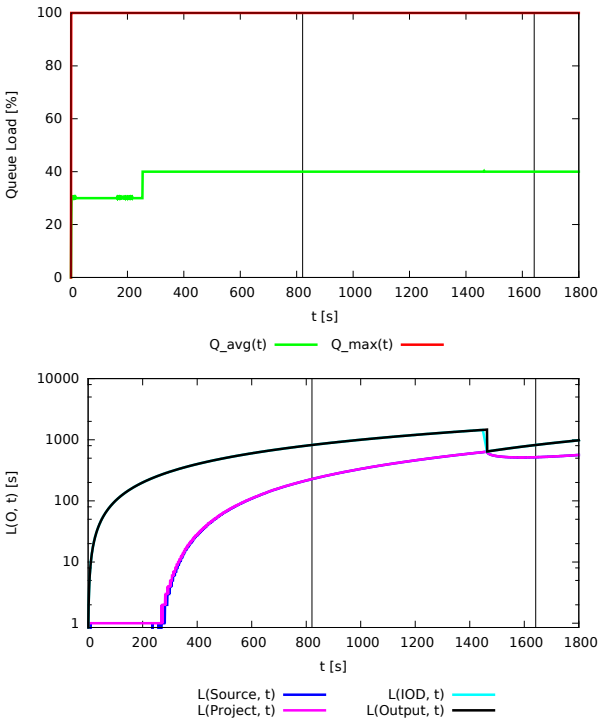

Figure 10. Results for benchmark test case run 8 (256 ips). Similar plots to the ones in figure 7.

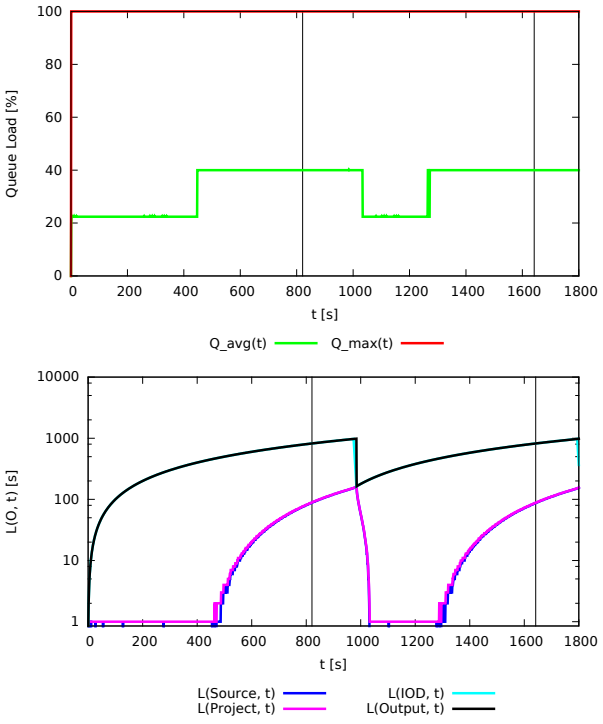

Figure 11. Results for benchmark test case run 9 (192 ips). Similar plots to the ones in figure 7.

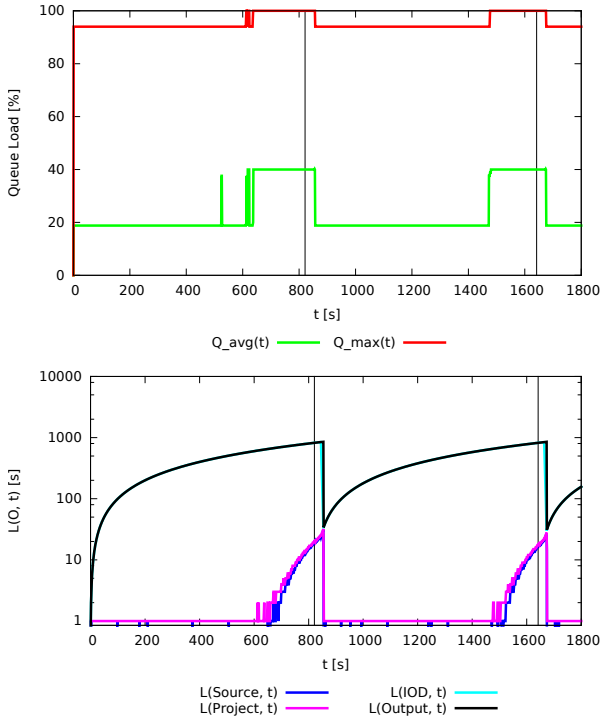

Figure 12. Results for benchmark test case run 10 (160 ips). Similar plots to the ones in figure 7.

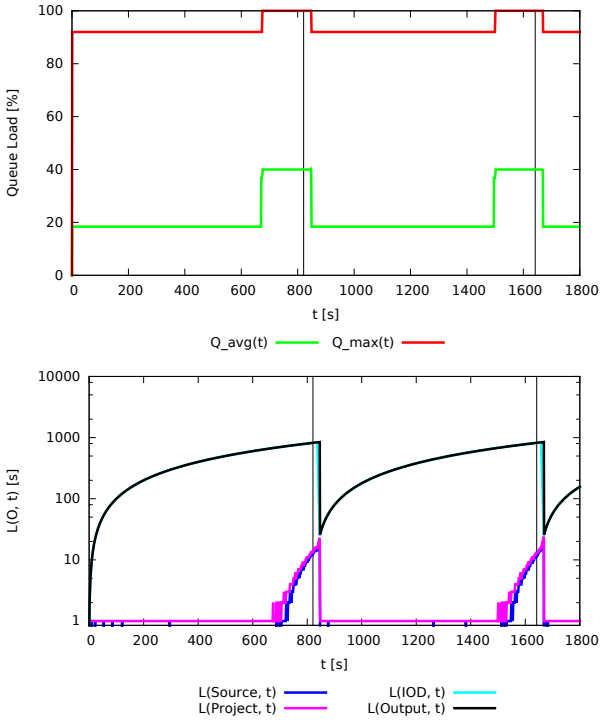

Figure 13. Results for benchmark test case run 15 (157 ips). Similar plots to the ones in figure 7.

ing 160 ips. Here, clearly $L_{\text{IOD}_0} > L_{\text{IOD}_1}$ with 34 and 31 seconds. Although these values remain above the threshold set in section 5, the system is not only "stable", but it seems to "catch up".

The system performance (cp. figure 12) shows a combination of the effects described for run 7 and 8: On the one hand, the queue loads are higher than normal for a certain time after a complete tracklet is input. On the other hand, there is a timestamp at which the average queue load increases to the same level before complete tracklet insertion and after a period of local blips. However, there is no blip downwards in queue loads directly after tracklet completions as in 7. It seems probable that the effect described for run 8 overlays the blip. Also, the maximal queue load shows the same behaviour as the average queue load which was not visible in run 8 as, supposedly, the input rate was too high to make this visible.

Run 11 produced results which were comparable to run 10. Consequently, run 12 was executed with 152 ips. 156 and 158 ips were used for runs 13 and 14, because the overall IOD lag remained nearly constant and beneath the threshold in runs 12 and 13. The performance results of runs 12 and 13 look quite similar and are comparable to the ones of 10 and 11, with the exception that no short blips are visible in the queue loads before the effect described for run 8 kicks in.

The performance results of run 14 look similar to run 13. However, $L_{\text{IOD}_0} < L_{\text{IOD}_1}$. The difference only amounts to 1 second and both lags are below the threshold, but, to be sure, 158 ips were determined as too much for the system.

Test case run 15 was the final run, since $\Delta_{15} = 1$ ips. The results are shown in figure 13. Here, $L_{\text{IOD}_0} = L_{\text{IOD}_1} = 27$ seconds resulting in a final $\iota = 157$ ips.

ι, L_{IOD_0} and L_{IOD_1} for all the bechmark test case runs are listed in table 1.

8. CONCLUSION

The main conclusion that can be drawn from the work this paper is about is that the DSMS STREAM is capable of executing IOD and that it is able to handle a realistic input data rate of a radar sensor in tracking mode. In fact, the benchmark test case results suggest that roughly 150 input data elements per second can be handled with a lag of maximal 30 seconds indefinitely. In cases where a higher lag of the system is tolerable, an even greater input data rate may be coped with (minimal 192 ips). These results were obtained on a widely available computer system indicating that higher rates are possible on high-end systems.

However, it should be noted that the execution of the test cases involved IOD only. In actual SST systems multiple functions are executed in parallel. Nevertheless, the STREAM program used is single-core and no distributed approach has been tested. Also, no performance tweak-

Table 1. Benchmark test case run results.

TC#	ι	L_{IOD_0}	L_{IOD_1}
0	1	2	2
1	2	2	2
2	4	2	2
3	8	2	2
4	16	3	3
5	32	3	3
6	64	5	5
7	128	10	9
8	256	643	-
9	192	164	163
10	160	34	31
11	144	25	23
12	152	16	16
13	156	29	27
14	158	27	28
15	157	27	27

ing has been done. Maximal queue sizes, for example, could be modified. Text file usage could be eradicated, as well. So, overall, the results look very promising regarding an application of STREAM in SST context justifying further research.

Future analyses could, for example, include a "drop" approach for the `FileSource`. Also, another interesting idea is to take advantage of the fact that IOD execution (and possibly the execution of other SST functions) takes a very small amount of time compared to the time that it takes to receive all detections of one tracklet: IOD could be executed several times based on the same tracklet, as long as new detections come in in order to have orbit data quicker. This would also be in line with the general idea of DSMS to constantly update all internal views on the data (like the view on orbit data) as soon as new data elements arrive at the system.

Quite some work presented in this paper has the potential to be reused in further research at IRAS. Processing operators can be easily and quickly added for other SST functions like Precise Orbit Determination (POD). Also, among others, the formal processing operator definition, the metrics and the testing procedure may serve well in upcoming developments and analyses.

ACKNOWLEDGMENTS

The work shown in this paper was achieved during the research project "RSS" (Fkz.: 50 LZ 1404) funded by German Federal Ministry for Economic Affairs and Energy. All responsibilities for the contents of this publication resides with the authors.

REFERENCES

1. S. Powers, "Ibm infosphere streams and the uppsala university space weather project," *Linux J.*, vol. 2009, no. 187, Nov. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1653247.1653248
2. M. Ali, "An introduction to microsoft sql server streaminsight," in *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Application*, ser. COM.Geo '10. New York, NY, USA: ACM, 2010, pp. 66:1–66:1. [Online]. Available: http://doi.acm.org/10.1145/1823854.1823929
3. R. J. Smith, "The oracle platform for real time streaming event driven architecture based solutions," in *Proceedings of the ACM SIGSPATIAL International Workshop on GeoStreaming*, ser. IWGS '10. New York, NY, USA: ACM, 2010, pp. 3–3. [Online]. Available: http://doi.acm.org/10.1145/1878500.1878503
4. C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 647–651. [Online]. Available: http://doi.acm.org/10.1145/872757.872838
5. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "Telegraphcq: Continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 668–668. [Online]. Available: http://doi.acm.org/10.1145/872757.872857
6. A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, *Data Stream Management: Processing High-Speed Data Streams*. Springer, 2004, ch. 1 of Part IV (STREAM: The Stanford Data Stream Management System), pp. 317–336.
7. Y. Ahmad, B. Berg, U. Cetintemel, M. Humphrey, J.-H. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, W. Xing, Y. Xing, and S. Zdonik, "Distributed operation in the borealis stream processing engine," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 882–884. [Online]. Available: http://doi.acm.org/10.1145/1066157.1066274
8. *STREAM: The Stanford Stream Data Manager – User Guide and Design Document*.
9. D. Geesen and M. Grawunder, "Odysseus as platform to solve grand challenges: Debs grand challenge," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '12. New York, NY, USA:

ACM, 2012, pp. 359–364. [Online]. Available: http://doi.acm.org/10.1145/2335484.2335523

10. A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1007/s00778-004-0147-z
11. S. Flegel, "Final report - maintenance of the esa master model," Institute of Aerospace Systems, Tech. Rep., 2011.
12. V. Braun, "Providing orbit information with predetermined bounded accuracy," Ph.D. dissertation, TU Braunschweig, December 2016.
13. D. A. Vallado and W. D. McClain, *Fundamentals of Astrodynamics and Applications*, 3rd ed. Microcosn Press and Springer, 2007.
14. U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '04. New York, NY, USA: ACM, 2004, pp. 263–274. [Online]. Available: http://doi.acm.org/10.1145/1055558.1055596