SoLOMon: Monitoring End-User Service Levels

S. Frølund, M. Jain, and J. Pruyne
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
{frolund, jainm, pruyne}@hpl.hp.com

Abstract

To manage distributed applications, we need to accurately monitor end-user service levels. There are two key challenges in monitoring end-user service levels: expressiveness and scalability. There is a big semantic gap between the metrics that administrators want to monitor and the metrics offered by commercial measurement systems. Moreover, it is hard to apply the same kind of metrics to different applications because different applications are likely to offer different types of instrumentation points. We want to apply the same metrics to very large numbers of instrumentation points, which makes scalability a key issue.

In this work we present the Activity Monitoring Language (AML) for declaratively specifying metrics, and a run-time system, called SoLOMon (Service Level Objective Monitor), that implements the concepts in AML. Expressiveness is a result of AML, which allows the high-level specification of user-defined metrics in an application-neutral way. SoLOMon's scalability is a result of reducing events and measurements as close to their physical source as is possible without the loss of accuracy.

1 Introduction

The technology for monitoring enterprise-scale systems has not kept pace with the deployment of large scale, distributed, heterogeneous applications. This has introduced several challenges for the IT departments of enterprises:

- There is a large semantic gap between the business metrics that the IT department would like to monitor and the type of measurement that is actually available.
- Monitoring solutions are application specific because each application presents different types of measurements and different measurement interfaces. There is no uniform way of carrying over the solution for monitoring one application to another, or of correlating metrics across applications.

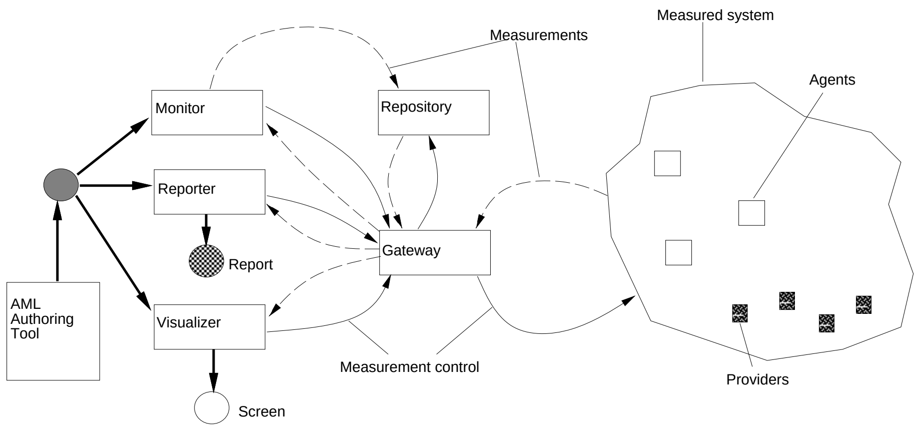

Figure 1: Architecture of The SoLOMon Monitoring System

- There is no easy, ubiquitous, scalable mechanism for gathering, correlating, and transporting the data from distributed sites to a central location.
- Most measurement systems do not have access to end-to-end metrics, and thus do not provide complete information.

To address these challenges, we introduce the Activity Monitoring Language (AML) for specifying measurement reduction according to user-defined metrics in an application-neutral way. An AML program declaratively specifies the computation of high-level business metrics from low-level events. In addition to measurement reduction, AML also facilitates measurement control. The bottom-level event sources are selected based on an attribute filtering mechanism.

SoLOMon (Service Level Objective Monitor) is a distributed measurement system that can execute AML specifications. There is a well-defined, and extensible, interface between SoLOMon and the native measurement points in the applications under measurement. Rather than provide a proprietary instrumentation format, SoLOMon provides an integration framework for existing instrumentation formats. AML contains an "interface defintion language" to facilitate the integration of existing instrumentation formats. The architecture of SoLOMon is illustrated in Figure 1.

The gateway component converts AML specifications into streams of measurements. Streams of measurements may have several consumers: graphical measurement visualizers, report generators, notifiers that apply thresholds to streams and produce events and alarms, and monitors that feed streams of measurements to other tools. These consumers may be implemented by anybody, and are not part of SoLOMon. In figure 1, we depict a monitor, reporter, and visualizer.

The gateway component itself can receive measurements from two sources: agents in a measured system or a measurement repository. The measurement repository contains historical data whereas the agents provide current data. Agents are part of the SoLOMon system, and perform measurement reduction according to mobile dataflow graphs. AML specifications are compiled into dataflow graphs that are loaded by agents at runtime. These graphs program the reduction engine in agents. Agents are organized in tree structures, which gives hierarchical measurement reduction. Moreover, since an agent can be collocated with a measurement point, our agent concept allows measurement reduction close to the measurement source—a key to scalable measurement collection.

The repository has a database in which it stores historical measurement data. We populate the repository through the monitor tool. The monitor tool obtains a live measurement feed from the gateway component and directs this live feed into the repository. The repository only contains the measurement that the user explicitly redirects to it—we can only present historical views on measurements that were explicitly collected.

Providers represent the instrumentation points in the application being managed. The provider abstraction gives a uniform interface to the heterogeneous instrumentation points in the various applications managed by SoLOMon.

The remainder of the paper is organized as follows. Section 2 describes AML in more detail, and section 3 describes an approach for implementing SoLOMon. Section 4 describes work related to SoLOMon. Conclusions and future work are presented in section 5.

2 AML

In this section we describe the AML concepts. The constructs of AML are designed with scalability in mind; we want to perform as much measurement reduction as close to the physical measurement source as possible. In AML, this goal manifests itself in constructs that can be calculated in a distributed manner.

2.1 Language Constructs

Each construct in AML is designed to perform one step in the process of transforming local, event-based measurements into values that represent a time-averaged view on multiple events from multiple instrumentation points. At the lowest level, we define *providers* and *provider types* which encapsulate different instrumentation points in a system. Above these, we define *metrics* which are procedures to convert events from these instrumentation points into values. The

```
provider CCMS {
  attributes
    initiator: string; application: string;
  events
    responseTime(eventID: uuid, time: long);
};

provider ARM {
  attributes
    user: string; application: string; activity: string;
  events
    start(tranID: uuid, timestamp: long);
    stop(tranID: uuid, timestamp: long, status: int);
};

metric responseTime: float msec;

responseTime for ARM {
  val = stop(status == "ok",tranID == t).timestamp -
    start(t = tranID).timestamp;
};

responseTime for CCMS {
  val = responseTime.time;
};

reducer mean(source S: float, period: long) {
  val = sum(S,period) / count(S,period);
};

source avgRTSrc: float;
avgRTSrc = mean(responseTime, 10) from filter { user == ‘‘joe’’}
```

Figure 2:

combination of a metric and a number of providers creates a *source*. Providers generate events and sources generate values. A source can be controlled (e.g. turned on or off). The highest level construct is the *reducer* which transforms one set of sources into another. We use reducers to aggregate values, such as computing the mean value over some time period. In the following sections, we describe each of these constructs in more detail by showing examples of how they are used. We will refer to figure 2 to show how each of the constructs is defined in AML.

2.1.1 Providers and Provider Types

Provider types define the instrumentation in the system being monitored. Provider types are user-defined as the type of information (events) may vary widely from one domain to another, and new types are likely to arise in the future. Provider types are the key abstraction that allow SoLOMon to work with heterogeneous information sources.

In Figure 2, lines 8-15 show an example of the AML syntax that defines an ARM [HP97] provider type. ARM is an application instrumentation API that gives a method of marking the beginning and ending of a transaction. A provider type definition has two sections: an attribute section and an event section. In the example, providers have attributes that capture the user of the process in which the instrumentation is embedded, the name of the application, and the name of the activity (transaction) initiated by the enclosing process. Provider instances bind values to these attributes. The binding of values to attributes happens at run-time and is not part of the AML specification.

The event section declares the different kinds of events that ARM providers can produce. In the example, ARM providers produce `start` and `stop` events. Events also have attributes. `timestamp` is an attribute of the `start` event. `timestamp` is of type `long`, and is the time at which the `start` event occurred. As we see in the next section, event attributes are used in defining metric procedures.

To further illustrate the notion of provider type, we give the AML definition for a hypothetical provider type that captures CCMS providers in Figure 2 lines 1-6. CCMS is the name of the monitoring system in the SAP/R3 environment. In this example, a CCMS provider produces events called `responseTime`. These events correspond to the completion of a business transaction.

We use the concept of an event as the common denominator for heterogeneous measurement data. Producing events imposes minimal requirements on participating instrumentation points because the concept of an event does not require any processing, such as time intervalization, of measurement data. Having events at the base layer gives rise to a "push" model for measurement collection and processing. Some legacy systems are likely to support a "pull" model instead. We can integrate such systems into SoLOMon by wrapping them with code that periodically polls the legacy measurement system and produces an AML event.

2.1.2 Metrics

The next level of abstraction in AML is the metric. A metric is a procedure that computes values from provider events. Figure 2 lines 16-25 define a metric to compute response time values. We use the keyword **metric** to define the name, type, and unit, of the metric. We then use overloading to define, for each provider, the procedure for computing the metric from events. The higher level constructs can thus be written in terms of metrics without regard to the underlying providers that generate the values. The definition in Figure 2 contains procedures for providers of type ARM and CCMS. The values computed by the response time metric are of type **float**. In general, metric values can be composite entities, such as pairs or records over other values.

The procedure to compute response time values from ARM events is complex because these values are computed based on two correlated events. We need the `start` and `stop` events from the *same* transaction to compute a valid response time. We therefore need to correlate events over a transaction. We use pattern matching over event attributes to describe event correlation. A pattern specifies acceptable values for (some of) the attributes in an event. For example, in lines 19 and 20, we describe a pattern that matches all `stop` events whose attribute status has a value of “ok”:

```
stop(status == "ok")                                              (1)
```

Notice that in (1) we do not specify values for all attributes. If a pattern does not specify a value for an attribute, it trivially matches the attribute.

For the purposes of event correlation we need to describe patterns whose attribute values depend on attribute values in other events. For example, to correlate ARM `start` and `stop` events, we need to construct a pattern for the `stop` event that contains the `tranID` value from the `start` event. We use the following syntax to express this:

```
stop(status == "ok",tranID == t).timestamp -
  start(t = tranID).timestamp                                     (2)
```

In (2), the `==` is a comparison operator and the `=` establishes a binding. The expression `t = tranID` establishes a binding for `t`. We can then use `t` in specifying patterns. Thus, when we write `tranID == t` we construct a pattern based on the value of `t`.

Notice that expression (2) may be evaluated only when all patterns are matched. Here there is only one pattern: the pattern for `stop` events. This pattern is matched whenever there is a successful `stop` event that has a `tranID` equal to that of a previous `start` event. We can access the attributes of the resulting matched event. To use the value of event attributes we use a dot (“.”) notation as follows:

```
stop(status == "ok",tranID == t).timestamp                    (3)
```

The result of evaluating (3) is the value bound to the attribute `timestamp` of the `stop` event that matches the contained pattern. We can then perform arithmetic expressions over these event attribute values. In the `response time` metric our operator is subtraction.

The computation of metric expressions over multiple events requires that events be stored. When a new event occurs, it may trigger the evaluation of a metric expression. For example, the occurrence of a `stop` event may cause the metric `response time` to be evaluated if a corresponding `start` event has occurred previously. In order to determine that such a `start` has in fact happened, we need to store old `start` events. We impose the semantics that the evaluation of a metric expression causes deletion of all the participating events. As measurement systems typically operate in resource constrained environments, forcing this semantics in the specification allows for efficient implementations. Although events are removed after they have been used in a computation, we still need a policy to deal with events that are never used in computations. For example, we may never see a stop event that matches a given start event. The runtime system must ensure that we do not store such start events forever. We choose to have an expiration time for events.

Consider the expression for computing response time values for CCMS providers:

```
val = responseTime.time;                                      (4)
```

There are no patterns involved in expression (4). So we can evaluate this expression for each `responseTime` event. The result of the expression is the value bound to the event attribute called `time`. The `responseTime` metric definitions for ARM and CCMS provide us with a single notion of response time for both CCMS and ARM though the underlying events produced by the corresponding instrumentation are very different.

2.1.3 Sources and Filters

A source is a logical instrumentation point that, when activated, can produce a stream of values. Underlying a source is a set of providers and a metric. The metric defines a procedure for computing values based on events from the providers. An example of a source is shown in lines 31-32 of figure 2.

In line 31, we declare a source `avgRTSrc`. In line 32, `responseTime` is the name of a metric. The expression "**filter** { `user == ''joe''`}" instantiates a filter that matches on an attribute `user` in providers. The filter will select all providers where this attribute is bound to the value "joe." The **from** keyword associates the filter with the metric. The second statement thus binds the source `avgRTSrc` to the result of associating the `responseTime` metric with all providers that have a `user` named "joe."

2.1.4 Reducers

The association of a filter and a metric is a primitive source. AML treats sources as first class entities, and defines operators over sources so that new sources can be constructed from old sources. A reducer is an operator over sources.

We allow the usual set of arithmetic operators over sources. In addition, we provide a number of built-in reducers to provide intervalization of sources. We pre-define the built-in reducers are **sum**, **count**, **min**, and **max**. The significance of these will be described in Section 3.2.

Built-in reducers provide time intervalization of measurement data. For example, **sum** is a built in reducer that takes a source S and a number T, and returns a source S'. Each value in S' is a sum of the set of values in S that appear over the interval T. A value in S' appears every T time units. If over a particular interval no values appear in S, a special null value appears in S' as the value for that particular interval.

As illustrated in lines 27-29 of Figure 2, we use the built-in **sum** reducer and a built-in **count** reducer to construct a user-defined reducer `mean`. Both **sum** and **count** behave as expected, and the `mean` reducer uses them to compute the time-averaged mean for the values in a source `S`. The `mean` reducer also takes a number `period`, that represents the time interval over which values in `S` are averaged. `mean` returns a source that contains the mean of `S` values computed on `period` time boundaries.

In this example, we compute arithmetic expressions over sources. We use a division operator "/" on the sources returned by **sum** and **count**. The result of merging two sources according to a binary operator is a source whose values are constructed from a per-value application of the operator on the values of the two merged sources.

Let a stream of values represent the runtime behavior of a source. We use the following notation to represent sources:

```
s = (t,v1 v2 v3 ...)                                    (5)
```

(5) represents a source `s` that is time intervalized on an interval of length `t`. `s` produces the values `v1`, `v2`, `v3`, and so on one after each interval `t`. With this notation, the semantics of a binary source operation, such as division, are as follows:

```
(t,v1 v2 v3 ...) / (t,v4 v5 v6 ...) = (t,v1/v4 v2/v5 v3/v6 ...)
```

Operators can only work on sources that are either all unintervalized or that are intervalized with the same period. For unintervalized sources, the division operator works as follows:

```
(,v1 v2 v3 ...) / (,v4 v5 v6 ...) = (,v1/v4 v2/v5 v3/v6 ...)
```

Time intervalization of data requires state in the reducer that performs the time intervalization. For example, the **sum** reducer must store a sum value that is updated whenever the input source produces a value. The sum value is set to 0 on interval boundaries, and it is written to the output source before it is set to 0. It is hard to efficiently implement user-defined reducers with state in a distributed manner. Therefore, user-defined reducers cannot have state. In particular, it is only built-in reducers that can time intervalize values.

The values produced by sources do not have any associated attributes, implicit or explicit. In particular, they do not have associated implicit timestamps. A value in a measurement stream may be the result of two events that ostensibly occur at very different times. Also, we do not assume clock synchronization or bounded clock skew between machines. Thus semantics of a time-stamp for this value is unclear. A side-effect is that unintervalized data stored in a repository cannot later be intervalized. However, the intervalization period of stored intervalized data can be increased.

The `mean` reducer in lines 27-29 of Figure 2 provides a general way to describe time averaging of measurement data. For example, we can use the `mean` reducer to measure the average response time for the user "joe" as shown in line 32 of the example.

3 Implementing SoLOMon and AML

The implementation of SoLOMon and AML should be low overhead, extensible, and timely in terms of the delivery of measurement data. Moreover, the implementation should be scalable in terms of the number of measurement points supported. We are currently prototyping the SoLOMon runtime system to satisfy these goals. In this section, we describe how to implement SoLOMon in a distributed JAVA environment. We use JAVA as an example platform because it supports dynamic code loading over a network. We could also implement SoLOMon in other distributed middleware environments such as COM or CORBA.

We illustrate the overall structure of the SoLOMon runtime in Figure 3. We describe the mapping from AML to an implementation language, in our case JAVA. Furthermore, we show how the runtime representation of an AML specification can be used to activate instrumentation points and generate measurement data according to user-defined metrics.

3.1 The AML to JAVA Mapping

We represent AML metrics and reducers as dataflow graphs in the SoLOMon runtime system. A graph that represents a metric has leaf nodes that correspond to provider events. The evaluation of the metric is triggered by event

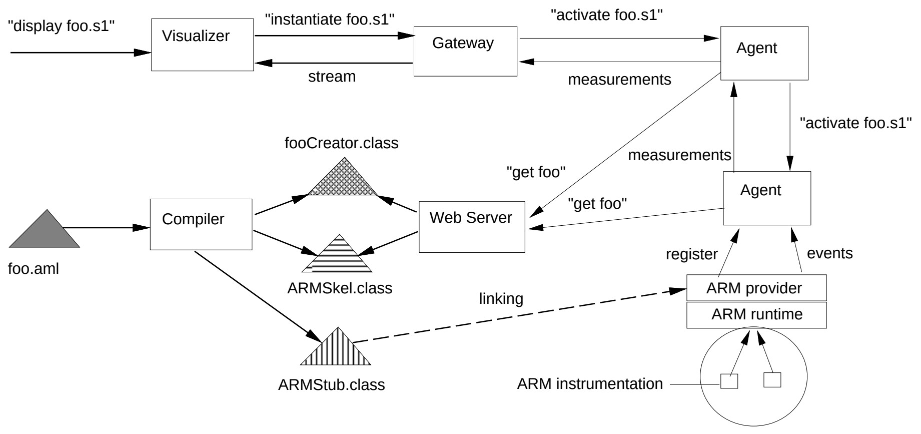

Figure 3: An overview of the AML to JAVA mapping and the SoLOMon runtime structure

occurrences at the leaf nodes. In the case of a reducer, the leaf nodes represent the sources being reduced, and the root represents the source which is the result of the reducer. The evaluation of a reducer graph is triggered by values being produced by the leaf sources. For an introduction to dataflow in general, refer to any of [Den74, AGP78, GKW85]; for a more indepth description of the implementation of metrics and reducers in terms of dataflow graphs, refer to [FJP98].

Given an AML text file, `foo.aml`, the AML compiler maps the metrics, reducers, and sources in `foo.aml` into a JAVA class called `fooCreator`. The class `fooCreator` has a method called `createGraph` that takes the name of a source and returns a dataflow graph that performs measurement reduction as prescribed by that source.

At runtime, measurement providers are the entry points into the SoLOMon runtime system. Providers are the runtime incarnation of provider types declared in AML files. The AML compiler generates provider stubs that can be used by a native measurement system to generate providers. For example, suppose that an application process contains ARM instrumentation. This ARM instrumentation can then use the provider stubs emitted for the ARM provider type to instantiate an ARM provider. The ARM instrumentation use the provider to register against the SoLOMon runtime and to push events into the SoLOMon runtime. At registration, a provider informs a SoLOMon agent about its attributes, the values bound to those attributes, and the events that it may generate. The stubs generated for a particular provider type will define the data format for registration and event generation for providers of that type.

In addition to provider stubs, the AML compiler also generates provider skeletons from a provider type. The provider skeletons define the dataflow leaf nodes that receive provider events. The skeletons are loaded dynamically by SoLOMon agents as they instantiate dataflow graphs.

In Figure 3, we assume that `foo.aml` only contains a single provider type called `ARM`. The stubs for this provider type are contained in the file called `ARMStub.class` and the skeletons are in `ARMSkel.class`. The stubs are linked into ARM providers, and the skeletons are loaded dynamically by agents that need to operate on ARM events.

3.2 Instantiation of Sources

A frontend tool such as a visualizer instantiates the dataflow graphs at runtime. For example, a user may ask the visualizer to display measurements as defined by a source, say `s1`. The tool then asks the gateway to instantiate the source into a stream of measurements. In response, the gateway returns a measurement stream. To feed measurement values into this stream, the gateway asks the SoLOMon agents to activate the specified source. Agents then load the dataflow creator `fooCreator` through a web server. They then call the method `createGraph` on `fooCreator`, with the source name `s1` as a text string. In return, they get the appropriate dataflow graph. Agents then ask their sub-agents to also activate the source, and perform the wiring between their own dataflow graph and those of their children. The bottom-level agents perform this wiring against one or more providers that have registered with the agent. The bottom-level agent only wires the dataflow graph against providers that match `s1`'s filter. This filter can be instantiated as JAVA objects through the `fooCreator` class.

We want to perform as much measurement reduction at the bottom-level agents as possible. Thus, ideally, we want lower-level agents to always wire their root node to the leaf nodes of higher-level agents. However, this wiring is not always possible. For example, if we have a reducer that counts all events in a distributed system, we cannot compute this value at any local agent. Counting is a global operator that can only be performed by the top-most agent. In order to correctly wire dataflow graphs, we have to "cut" them so that global operations are not distributed.

Our first step in performing the cut is to define a tree-structured hierarchy among agents. By restricting this hierarchy to a tree structure, we limit each agent to having exactly one parent agent that it reports to. We also use the observation that there are well defined methods of distributing the built-in reducers: **sum, count, min, max**. For example, a global **sum** is simply the **sum** of a set of partial **sum**s. Likewise, a global **count** is the **sum** of a set of partial **count** operations. Similar strategies are apparent for **min** and **max** operations. Because we can decompose these operations, they allow us to perform our cut.

The tree hierarchy and built-in reducer properties allow us to use the following algorithm to distribute a dataflow graph. When the "root agent" receives a reference to a dataflow graph from the gateway, it uses this reference to download the graph from the web-server and uses the `createGraph` method to instantiate the graph. The graph is then traversed by the agent looking for one of the built-in reducers. If none of these are found, the graph cannot be cut, and the agent sends the graph, in its entirety, to each of its children. The output of this graph in each of the children is sent to the root. The root does no processing on these events, and simply passes them through to the gateway.

If the root does discover one of the built-in operators, it performs a cut operation as follows. It instantiates the proper global reducer for this operation. As described above, for **sum** the global operation is **sum**, for **count** the global operation is also **sum**, and so on. The root then passes the portion of the graph below and including the reducer to each of its children. The output of the children will be directed to the global reduction operator instantiated by the root. The root will therefore perform the global reduction step before passing the result on to the gateway.

The process described here was in terms of two levels: a root and its children. The algorithm generalizes easily to multiple levels. All non-leaf level agents implement the cut algorithm described above, and pass the appropriate subgraphs to their children. Leaf agents do not need to run the algorithm, and will simply load and run each graph they receive.

4 Related Work

HP's Data Source Integration (DSI) language [Hew96] is used in conjunction with the HP Measureware collection agent [Hew95]. DSI allows external measurement providers to use the Measureware collection infrastructure. A DSI specification defines the format of a particular class of measurement data that is produced by an external measurement provider. In contrast to AML, DSI only addresses the interaction between measurement providers and a single collection agent. DSI does not support distributed and hierarchical measurement reduction, not does it support control of measurement providers based on their attributes.

The InfoVista system [Cor97] contains a language for describing metrics. It is possible to describe composite metrics in terms of simpler metrics. However, InfoVista does not support distributed computation of metrics: composite metrics are computed at a centralized measurement server. Moreover, InfoVista uses explicit, rather than attribute-based, naming of measurement sources.

The Distributed Measurement System (DMS) [RJTS95, JFS95] has per-computer measurement agents that collect, intervalize, and transport measurement data to a central location. An agent The notion of threshold, or intenisty level, in DMS allows control of individual measurement sources. They can be turned on and off and instructed to operate at different levels of data aggregation. However, DMS only provides a fixed set of intensity levels, there is no way to define custom thresholds and use them for existing DMS measurement sources. Moreover, DMS measurement sources have unique names, there is no support for attribute-based naming.

The stream concept in dataflow languages, such as Lustre [NPPD91] and ESTEREL [BG92], is similar to our notion of a source. Where AML provides reducers to perform computation of sources, these languages provide operators to perform computation over stream values. However, the data operators in Lustre and ESTEREL do not support any notion of intervalization. Furthermore, Lustre and ESTEREL do not support attribute-based naming of data sources, nor do they provide operators to specify the computation of values based on events.

5 Conclusion

We have introduced SoLOMon, a distributed monitoring framework designed to provide scalability and expressiveness. SoLOMon's scalability is primarily achieved by reducing measurements as close to their physical source as possible. Expressiveness in SoLOMon is a result of using a high-level language, AML, as a declarative front end to the measurement system, making the system programmable.

The SoLOMon framework is built around the notion of reducing events to values. A topic for future work would be to extend the framework to provide ways to generate events from values. Another extension is to provide operators that reduce events to other events.

6 Acknowledgements

The work reported in this paper has benefitted greatly from interactions and discussions with a number of individuals. We want to acknowledge the contributions of Muthusamy Chelliah who participated in the early design meetings. Joe Martinka continually supported and encouraged our endeavors into distributed measurement systems. We thank Joe Sventek for his comments on a draft version of this paper. Finally, we thank Brad Askins and Pankaj Garg for their feedback on our work.

References

[AGP78] Arvind, K.P. Gostelow, and W. Plouffe. An asynchronous programming language and computing machine. Technical Report UCI-

TR114a, U.C. Irvine, Dept. of Information and Computer Science, UC Irvine, December 1978.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, and implementation. *Science of Computer Programming*, 19:87–152, 1992.

[Cor97] InfoVista Corporation. It quality of service management solutions. Technology white paper from http://www.infovistacorp.com/, March 1997.

[Den74] J.B. Dennis. *First Version of a Data Flow Procedure Language*, volume 19 of *Lecture Notes in Computer Science*. Springer-Verlag, 1974.

[FJP98] S. Frolund, M. Jain, and J. Pruyne. Solomon: Monitoring end-user service levels. Technical Report HPL-TR98-153, Hewlett-Packard Laboratories, 1998.

[GKW85] J.R. Gurd, C.C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[Hew95] Measureware agent: User's manual, 1995.

[Hew96] Measureware agent: Data source integration guide, June 1996.

[HP97] Hewlett-Packard. Application response management. http://www.hp.com/-openview/rpm/arm/index_f.html, 1997.

[JFS95] J.Martinka, R. Friedrich, and T. Sienknecht. Murky transparencies: Clarity through performance engineering. *Proc. of the Intl. Conf. on Open Distributed Processing (ICODP'95)*, February 1995.

[NPPD91] N.Halbwachs, P.Caspi, P.Raymond, and D.Pilaud. The synchronous dataflow programming language lustre. *Proc. of the IEEE*, 79(9), September 1991.

[RJTS95] R.Friedrich, J.Martinka, T.Sienknecht, and S. Saunders. Integration of performance measurement and modeling for open distributed processing. *Proc. of the Intl. Conf. on Open Distributed Processing (ICODP'95)*, February 1995.