Towards Dynamic Model Driven Architectures

Morgan Ericsson, Welf Löwe, and Jesper Andersson

Software Technology Group, MSI, Växjö universitet
{mogge | wlo | jesan}@msi.vxu.se

Abstract Scientific applications using data from networks of sensors must be both highly flexible and high performing. A service-oriented architecture makes sense for modelling such applications, but not for implementing them, due to performance issues and architectural mismatch. In this paper we present an architecture that aims at solving these problems. Applications are modelled as services in a service-oriented architecture, mapped to high-performance, data-driven architectures. Each component is a parallel application. This mapping is done using a MDA approach and is changeable at runtime due to a dynamic Architecture Pattern.

1 Introduction

With recent advances in sensor, communications and computer technology it is now possible to construct huge arrays of geographically distributed heterogeneous sensors connected by a high-speed network. Such a Wide Area Sensor Network (WASN) can be used for a multitude of purposes, for instance to help observe, understand and shape the world around us and sky above us.

LOFAR (LOw Frequency ARray),\(^1\) is the first large-scale attempt at constructing such a WASN. It consists of geographically distributed digital receptor units connected to computing facilities with a high-speed network. Approximately 13,000 such units are distributed over an area 400 km across, and any unit will produce data at a rate of 2 Gbits/s, resulting in a total system data rate of 25 Tbits/s. This data-rate can be compared to streaming 731 DVD-movies per second. The Swedish initiative LOIS (LOFAR Outrigger In Scandinavia)\(^2\) aims, among others, to enhance the IT infrastructure.

Researchers that want to use the WASN submit applications to it. These applications must perform well enough to deal with the enormous data-rates, yet the development process must be simple enough to allow for non-expert developers and flexible enough to allow for external resources to be integrated. This paper presents a solution using MDA, that transforms between a conceptual design-level and a physical, high-performance implementation-level.

The paper is organized as follows: Section 2 introduces the architecture and the requirements of the system, Section 3 shows how MDA can be be used, and section 4 introduces our Dynamic Service Architecture and its instantiation to Dynamic MDA. Section 5 concludes the paper.

\(^1\)\url{www.lofar.org}
\(^2\)\url{www.lois-space.org}
2 System Architecture

A sensor network is, in the context of this paper, defined as a set of sensors generating input, a set of computational nodes transforming the input and a set of interconnecting links transporting the data. All these entities are assumed to be heterogeneous, i.e. there can be different types of sensors scattered around the network, and the capacity of the links can vary as can the computational power of the different computational nodes. We further assume that the sensor network is closed in the sense that all resources are under our control.

The software is an integral part of a sensor network. The principle tasks of this software are to control the sensor network and its applications, as well as performing the computations necessary to produce the desired data output. These computations can be divided into two categories, transformation and aggregation. Computations in the transformation category deal with any kind of transformation that is necessary to produce usable data. This can, for instance, be error-corrections. Aggregation reduces the data rate, for instance by removing or accumulating data.

A sensor network must be very resilient to change. We assume a event-driven change model, were events are used to signal changes to the sensor network or the applications running on it. The result of any of these events should be a reconfiguration of the software in order to deal with the change signaled. There exist three classes of events:

**User events** are triggered by user interaction, such as adding or removing software or manually changing the parameters of a running application.

**Application events** are triggered by the application, as a response to some condition being fulfilled, for instance by some pattern in the data stream.

**System events** are triggered by the system in order to signal a change in the environment. Things that could cause system events to be triggered could, for instance, be hardware failures or a need for load-balancing in order to satisfy the globally desired level of quality of service.

As noted in previous works, e.g Bass et.al [2, p 32.], an architecture “inhibits or enables” quality properties in an application. In the process of finding a suitable candidate, these properties are used for evaluating different candidates.

2.1 Service-Oriented Architecture

In order to provide users with data from a sensor network, we choose to view a sensor net as a collection of services, where each service provides a view of the data produced by the sensor network. A view can be any combination of transformation and aggregation operators applied to the data stream. A service, in this context, is self-contained and communication is message-oriented. This is consistent with the Service-Oriented Architecture style (SOA).

A system designed using a Service-Oriented Architecture style achieves a high degree of flexibility and programmability. A client can bind or re-bind to any given service at any given point in time. A client can further discover services at run-time, using a locator service. A message-passing scheme moves the semantics of communication into
the format of the message, which achieves platform independence, as long as the message format is platform independent.

When it comes to performance, the SOA-style is a problem. Moving to message orientation on that level has a high overhead. In order to keep the services platform and client independent, they must be design so that the message format is self-describing. They must further use a message format that is universally parseable. This implies large messages and costly parsers. A common way of implementing SOA is by using Web Services [3], and here the cost of SOAP (XML) messages is very visible. See for instance [4] for more information.

Another issue is the architecture mismatch. A service-oriented architecture assumes message-orientation, i.e. a pull-style, while the sensor network is data-driven, i.e. push-style. While this is something that can be converted between, it introduces further performance issues.

2.2 Data-Driven Architectures

Given that the sensor network is data-driven, a data-driven architecture style, where transformations and aggregations are implemented as (parallel) filters on a data stream may be a more suitable implementation. The data-driven architecture will deliver the required performance, but is very limited with respect to flexibility. In order to maximize the performance, static scheduling is used, which rules out any change to the execution environment. If dynamic scheduling is used, performance will be worse, due to the limited knowledge of the application structure, but will allow for a certain degree of flexibility when it comes to changes in the execution environment. For both cases, interoperability is a problem. Any external inputs or outputs must be defined at the deployment of the system, and conversions to a suitable data-format must be performed. Another issue with data-driven architectures is that the development of applications in such a scenario requires expert knowledge of the parallel-programming environment and good knowledge of the structure of the distributed system. The typical researcher developing applications for the sensor network can not be expected to possess this knowledge.

2.3 Requirements and Approach

The software running on the sensor network must adhere to four non-functional requirements.

- The sensor network must be open, i.e. it must be able to interoperate with external applications, both in terms of providing and using data.
- The sensor network must be able to deal with changes introduced by changing conditions in the real world that is to be observed, changing hardware resources or a changing software environment, for instance due to new software being deployed.
- The performance of the sensor network must be able to deal with the data rate and the quality of service requirements of all deployed software.
- The development of applications that should run on the sensor network must be intuitive and have a short time-frame from idea to finished application.
These requirements call for both high performance and a high degree of flexibility. The development process should, as far as possible, be driven by reuse of existing entities and at high-level as possible.

The requirements on development process and flexibility clearly suggest a service-oriented approach, while the performance requirements do not. We therefore differ between a conceptual level, used to design and annotate applications, and physical level used for the actual running system. This way, a SOA-style can be used to design applications, and describe and reason about the system, while a data-driven approach is used for the implementation.

3 Model Driven Architecture

As described before, researchers want to deploy new experiments online. Since they are rather interested in their respective experiments, the development, integration, and deployment of the corresponding software components should be easy to understand, and manageable not only by computer scientists. We have argued, that a SOA is an appropriate system view. In this view, the system consists of a number of services composed to higher-level services by orchestration.

Some of these services are delivered by third parties and are not under our control, others have low performance requirements. These two classes of services remain service-oriented components also in the physically running system. Other services have high performance requirements making it necessary to translate them to a data-driven architecture on the physical level.

To provide an easily manageable system with high performance, we need to provide both a conceptual and a physical view of the system, and a mapping between the two. Let’s ignore the fact that the two levels need to be changed at system runtime for a while, and concentrate how to establish the mapping.

Model Driven Architecture (MDA) provides us with a framework for such a mapping [7,5]. In MDA, one defines a Platform Independent Model (PIM) of the system. The PIM is annotated with stereotypes and tagged values introducing special model entities and their parameters, respectively. From the PIM, a Platform Specific Model (PSM) is generated. Generators may be specific for the different stereotypes and are controlled by tagged values.

3.1 Platform Independent Model (PIM)

This PIM corresponds to a SOA-view on the system, i.e. to the conceptual view. Here, we have different specifications each providing us with a certain aspect on the system:

**Service Aspect:** Gives us the global system architecture. Its elements are classes, interfaces and their aggregation. We introduce a stereotype «exported service» to indicate that some classes might be exposed for external access. Another stereotype, «imported service» indicates that some services are not under our control. The third stereotype «data service» indicates a data source that is not under our control either. In contrast to the «imported service», «data
service» is a data pump pushing data into our services. All services with stereotype «exported service» and «imported service» preserve a services interface even on physical level. Depending on the performance requirements, other services may or may not be translated to data-driven components. Annotations of services with performance requirements are tagged values and added to this view, as well. We also annotate the input and output sizes of the service using tagged values.

**Data-Driven Aspect**: defines the data-parallel implementation of a service. Therefore it may use other services and, hence, this view also defines the orchestration of services to larger ones. We assume a High-Performance Fortran (HPF)-like programming model, with data parallel synchronous program but without any data distribution.

### 3.2 Platform Specific Models PSM\textsubscript{task} and PSM\textsubscript{schedule}

There are two Platform Specific Models (PSMs) we would like to separate; we refer to them as PSM\textsubscript{task} and PSM\textsubscript{schedule}, respectively. PSM\textsubscript{task} makes explicit, which service of PIM remain service even in the running system and which is further translated to a data-driven application. PSM\textsubscript{schedule} lowers the level of description even more. It schedules processors to the tasks in the data-driven applications and communications between them.

**Mapping PIM to PSM\textsubscript{task}**: The mapping PIM to PSM\textsubscript{task} consists of two generation processes. The first decides, which service of PIM remains a service on physical level and which is further translated to a data-driven application. For a first try, everything that is an «exported service», an «imported service», or a service with a required data rate below a certain boundary remains a service. This step also includes the generation of buffer processes to adapt the conflicting architectural styles where necessary. We need two basic adapters for dealing with expected architectural mismatches:

- A push-to-pull adapter that converts the data stream pushed out from a data-driven application to a service that delivers data on request. This is basically implemented by a buffer of the data stream. From outside, a push-to-pull adapter appears like a service on physical level.
- A pull-to-push adapter that converts the service providing some data on request to a data-driven application. It request the data and pushes it into the data-driven application. This basically implemented by calls the services and subsequently sending the received data to the input of a data-driven application. From outside, a pull-to-push adapter appears like a data-driven application.

These two adapters are introduced as new stereotypes on the PSM\textsubscript{task} level. The stereotype «exported service» disappears, we just distinguish «service» and «imported service» from «data service».

Note that the semantics has changed compared to the PIM. In the PSM\textsubscript{task}, a «service» and an «imported service» is guaranteed to remain a service on
physical level and a «data service» is guaranteed to become a data-driven application.

The second part of the PIM to PSM task mapping is actually already preparing the mapping PSM\text{task} to PSM\text{schedule}. It computes the task-graphs of the data services. We can model a data-driven program on an input $x$ by a family of task-graphs $G_x = (V_x, E_x)$. The tasks $v \in V_x$ model local computations without access to the shared memory, and there is a directed edge from $v$ to $w$ iff $v$ writes a value into the shared memory that is read later by task $w$. Therefore, task-graphs are always acyclic. $G_x$ does not always depend on the actual input $x$. In many cases of practical relevance it only depends on the problem size $n$. We call these program oblivious and denote their task-graphs by $G_n$. We write $G$ instead of $G_n$ if $n$ is arbitrary but fixed. Scientific applications can automatically be compiled to such a family of task-graphs. Many of them are oblivious or iterations over oblivious loop bodies, e.g. Matrix Multiplications, Fast Fourier Transformations (FFT), CG-Methods, Finite-Element-Methods etc. [8].

For any service that should be translated to a data-driven application, the data-driven aspect, i.e. the data-parallel program, of that service is mapped to its task-graph for the given input and output sizes.

Mapping PSM\text{task} to PSM\text{schedule}. The mapping PSM\text{task} to PSM\text{schedule} computes the actual schedule for the whole application. A schedule of a task graph is a mapping of its task to actual processors and inserts explicit communication operations. Let $p$ be the maximum number of linearly independent tasks in a task graph $G_n$; i.e. the maximum parallelism in a service for a given input and output size $n$. Then we compute schedules for that service for $P = 1 \ldots p$ processors. These pre-computed schedules are inputs for scheduling services $G'$ using $G_n$ etc. This scheduling is called malleable task graph scheduling and is described in detail in [9]. We may assume that the resulting schedules guarantee that all tasks of task graphs are computed at least once and that the data-dependency constraints induced by the task graphs are met by the communication operations.

The next section serves as an extended example explaining the different models and their mappings.

3.3 Example 2D Image Convolution

The following example computes a 2D convolution of images (2D matrices of complex values). Figure 1 shows the service aspect of the application. The convolution service refers to a 2D Fast Fourier Transform (fft2d) service and an inverse 2D FFT service (invfft2d), which in turn refer to FFT and inverse FFT services, respectively. Convolution is the only «exported service». Note that, in our example setup, the first image is provided by «data service», i.e. comes from a data-pump. The second image comes from an «imported service» providing a constant filter.

Figure 2 shows the data-driven aspect, i.e. the data-parallel programs, of the example. The pardo command executes the enclosed statements in parallel. The for $i = 0 \ldots n$ do in parallel executes the enclosed statements for all array elements $a[0], a[1], \ldots, a[n]$ in parallel.
Lets assume that all but the external services providing the constant second image are to be translated to a data-driven application. We would then need two adapters for dealing with the architectural mismatch:

- A push-to-pull adapter that converts the data stream pushed out from the convolution application to a service that delivers data on request. This is basically a buffer of resulting images.
- A pull-to-push adapter that converts the service providing the second image and pushes it into the data-driven convolution. This basically calls the service and sends the received image as input to the convolution.

Figure 3 shows the service aspect on the PSM$_{task}$ of the convolution application including the new adapters.

Now we are ready to compute the task-graph of the application. Note that the matrix convolution is an oblivious program since the data-dependencies are only dependent on the input size, not the input data.

Figure 4 shows the task-graphs of FFT and inverse FFT, respectively. Figure 5 (left) shows the task graphs of the 2D FFT and 2D inverse FFT, respectively. The boxes represent FFTs and inverse FFTs, respectively. Each arrow connects an array of 8 complex output values with the corresponding input array. Figure 5 (right) shows the task graphs of the 2D matrix convolution. The boxes represent 2D FFT and 2D inverse FFT, respectively. Again, each arrow connects an array of 8 complex output values with the corresponding input array.
fun fft(v : array[n] of complex, ω : complex) : array[n] of complex
  //r(i) denotes the value of the reversed bit representation of i.
  for i = 0, . . . , n − 1 do in parallel v[i] := v[r(i)]; end;
  for j = 0, . . . , log n − 1 do
    for k := 0, . . . , n/2^{j+1} − 1 do in parallel
      pardo
      v[k·2^{j+1} + i] := v[k·2^{j+1} + i] + ω^{i·n/2^{j+1}}·v[(2·k + 1)·2^{j} + i];
      v[(2k + 1)·2^{j} + i] := v[k·2^{j+1} + i] − ω^{i·n/2^{j+1}}·v[(2·k + 1)·2^{j} + i];
    end
  end
end

fun fft2d(a : array[n, n] of complex, ω : complex) : array[n, n] of complex
  for i = 0, . . . , n − 1 do in parallel
    a[*, i] := fft(a[*, i], ω);
  end
  for i = 0, . . . , n − 1 do in parallel
    a[i, *} := fft(a[i, *], ω);
  end
  return a end

fun invfft(v : array[n] of complex, ω : complex) : array[n] of complex
  //analogous to fft
end

fun invfft2d(a : array[n, n] of complex, ω : complex) : array[n, n] of complex
  // analogous to fft2d
end

fun convolution(a, b :: array[n, n] of complex, ω : complex) : array[n, n] of complex
  pardo
  a := fft2d(a, ω); b := fft2d(b, ω)
  end
  for i, j = 0, . . . , n − 1 do in parallel
    c[i, j] := a[i, j] · b[i, j]
  end
  return invfft2d(c, ω)
end

Figure 2. Convolution of 2D Images – Data-Driven View of PIM.
Finally, PSM\textsubscript{schedule} is generated from PSM\textsubscript{task} using malleable task graph scheduling. The maximum parallelism in the whole task graph is 32 (two 2D FFTs on the two input images each consisting of 2 FFTs on an array with input size 8. Hence we pre-compute FFT and inverse FFT schedules for 1..8 processors, 2D FFT and 2D inverse FFT schedules for 1..16 processors and a schedule for the whole convolution for 1..32 processors. At the price of getting the minimum completion time of the schedule but in order to save pre-computation time, we could reduce the numbers of schedules we pre-compute, e.g. we could schedule for 1, 2, 4, 8, 16 and 32 processors.

Figure 6 shows two alternative PSM\textsubscript{schedule} instances coming from the same PSM\textsubscript{task} instance depending on decisions in the scheduling phase. Figure 6 (left) shows a schedule where the 2D FFTs are executed in parallel, each on 4 processors. Figure 6 (right) shows a schedule where the 2D FFTs are executed in sequence, each on 8 processors. It is not the scope of this paper to present the malleable task-graph scheduling algorithm or the cost model it bases its decisions on. For this, we refer to [9].

## 4 Dynamic Service Architecture

So far, we defined an offline design-process mapping a SOA-view on our system to its physical implementation. Now we need to allow online changes of both the SOA-view and its physical implementation. In order to introduce this dynamism, we define a more general solution that we instantiate in our scenario.
We define a so-called Dynamic Service Architecture (DSA), a composite pattern influenced by architectural patterns proposed by [1]. The resulting architecture separates a description of the running system from control-entities manipulating the descriptions and thereby the running system.

### 4.1 General Dynamic Architecture

This conceptual architecture has two aspects: control and processing, as depicted by Figure 7. The control architecture is concerned with monitoring and, if needed evolution of the processing architecture to suit new requirements and/or constraints. The processing architecture captures the current configuration of the the running system, that is it serves as a (meta-level) description of the running system.

The **processing architecture** contains three basic component types; *Component*, *Connector*, and *Configuration*. These components and their relationships are depicted on the right-hand side of Figure 7. The whole application is a *Component*. A *Component* is either basic or complex. In the latter case it contains further *Components* communicating via *Connectors*. A *Connector* captures component interactions. The typed
end-points of a Connector connect to Component’s interface points. Both Components and Connectors are parameterized by Configurations.

The control architecture consists of five component types; Probe, Actuator, Generator, Validator, and Coordinator. The Probe and Actuator are the bridging point where control connects to processing. A Probe is a configurable monitoring process that monitor processing components and generate events that are communicated back to its Coordinator. The Coordinator is the decision maker responsible for coordination and delegation of control tasks. It may employ different Generator and Validator components, for instance for creating new application configurations or rule-sets for coordination control and monitoring. The meta-data is provided by the processing architecture elements described above. The Coordinator use Actuator components to directly affect application configurations and/or application component instances.

4.2 General Dynamic Model Driven Architecture

We are now ready to combine the concepts of Dynamism and Model Driven Architecture. Figure 8 sketches this combination.

In MDA, a Platform Independent Model (PIM) is stepwise refined to a Platform Specific Model (PSM) and, finally, translated to the running system. Each model describes the finally running system. It contains definitions of special components, special connectors and special configurations. Therefore, each model is a special processing architecture of the Dynamic Service Architecture (DSA). Corresponding to each model, we need special conceptual architectures of the DSA with special generators, actors and coordinators. Note, that only the physically running system is monitored for system and
application events. Hence, only the special conceptual architecture of this level contains a probe component.

User driven change-events are processed by the top-level coordinator, c.f. Figure 8 (1). Employing the corresponding actuator and generator, this coordinator triggers changes in the top level model (PIM), c.f. Figure 8 (2). This, in turn, requires changes on the lower-level models. Accordingly, the coordinator on next level receives an event from the coordinator on the previous level and gets the model of that level as input, c.f. (3). This continues down to the physically running system.

System and application driven change events take the opposite direction. They have their origin in the running system, the runtime environment and the running application, respectively, c.f. Figure 8 (a). The probe component is responsible for these events. Employing the actuator and generator, the coordinator on the lowest level triggers changes in the running system, c.f. Figure 8 (b). This may or may not trigger changes on higher levels. If the original and the changed running systems correspond to the same model on the next higher level (PSM), no further actions are taken. If, in contrast, the PSMs are different, the changes in the running systems need to be reflected on PSM level, as well. In that case, the corresponding coordinator is informed (c) and its model updated (d). This continues until a level where the original and updated models are identical, or the top level is reached.

4.3 Our Instance Dynamic Model Driven Architecture

This section instantiates the general DMDA introduced in Section 4.2 for the purpose of our application scenario. It is sketched in Figure 9.

Corresponding to our PIM, PSM_{task}, PSM_{schedule}, as introduced in Section 3, and the finally running system, we define four specializations of the general DMDA. The
generators on each of these levels correspond to the mapping from a model to the next lower model. Note, that the service-oriented view on the system (our PIM) is directly defined by the user using a model editor. The next levels are generated by a task-graph generator (PSM\textsubscript{task}) and scheduler (PSM\textsubscript{schedule}), respectively. Finally, the program needs to be compiled and deployed.

Remark 1. A special kind of system-driven change-events indicate performance bottlenecks, that may lead to changes of the running system for load-balancing. Note, that moving a task to another processor (load-balancing) needs to be reflected on PSM\textsubscript{schedule} but wouldn’t have any effects on PSM\textsubscript{task}. Hence load balancing changes would be propagated only to the PSM\textsubscript{schedule}-level.

In our PIM, components are UML-style service definitions (Service-Oriented Aspect) and their data-parallel implementation (Data-Driven Aspect). Component configurations are induced by stereotypes on that level and tagged values describing quality of service. Connectors are UML-style service aggregations; their configurations are the tagged values describing input / output sizes and quality of service, as well.

On PSM\textsubscript{task} level, components are UML-style service definitions (coarse-grained components) and tasks (fine-grained components), respectively. Connectors are the are UML-style service aggregations (coarse-grained connectors) and the data-dependencies between the tasks (fine-grained connectors). Configurations are just transferred from the PIM level.

On PSM\textsubscript{schedule}-level, components are sequential programs assigned to specific processors. Connectors are explicit communication-calls in the components. This level
corresponds to an explicitly distributed program in a high-level programming language using communication libraries for explicit data exchange (like MPI or PVM).

This PSM \textsubscript{schedule} can directly be compiled into a executable system. Replacing a running system with an updated one can be seen as:

1. Shipping the code to the respective processor,
2. Connecting the output of unchanged data services and unchanged external services with the corresponding input nodes of the new system,
3. Connecting the unchanged exported services with the corresponding output Nodes of the new system,
4. Disconnect input and output nodes of the old system,
5. Remove the code of the old system.

This simple updating routine is possible since the data-driven applications have no state but are rather filters of the input stream. The state is only captured in the external and imported services that remain unchanged or are completely removed.

In [6] we introduced a more fine-grained updating routine that tries to update only deltas and leaves unchanged parts of the system untouched. The paper further discuss a look-ahead scheduling which conservatively foresees certain updates before the triggering events actual occur. This reduces the respons-time to system and application-driven change-events.

5 Conclusions

In a concrete application scenario, the LOIS sensor network, we faced the problem of achieving programmability, flexibility and high performance at the same time, three
usually conflicting requirements. Instead of solving the problem directly, we generalized it to systems with both data-parallel and service-oriented subsystems.

For these kinds of systems, programmability and performance issues could be achieved with a Model Driven Architecture (MDA) approach. The system is given as a Platform Independent Model which simple to define, and then stepwise and automatically refined to a high-performance Platform Specific Model.

Flexibility was introduced by merging the MDA approach with the Dynamic Architecture pattern leading to what we call a Dynamic Model Driven Architecture (DMDA). DMDA specializes the Dynamic Architecture pattern and shows how the different Platform Independent and Specific Models can be updated according to user, application and system triggered change events.

DMDA defines now a framework for solving the original problem. Its concrete implementation and evaluation are matters of future work.

References