Early exploring design alternatives of smart sensor software with Model of Computation implemented with actors

Jean-Philippe Schneider Zoé Drey Jean-Christophe Le Lann
UMR 6285 Lab-STICC, ENSTA Bretagne
jean-philippe.schneider@ensta-bretagne.fr

Abstract

Cabled sea floor observatories are used to study the oceans. Not only the amount of data they generate is too large to be treated manually, but also a lot of irrelevant data degrades the treatment efficiency. Treating the acquired data at the source reduces the amount of data. But this leads to the design of complex sensors mixing data acquisition and treatments.

Simulation can help supporting the design activity. However, there is no way to bypass accurate modeling of concurrent systems while ensuring a strong semantic during the model execution.

In this paper, we present an experimentation using Models of Computation to model the concurrency in the program of a smart sensor. Besides, we introduce an actor-based simulation framework.The simulation framework is used to quickly and cheaply test alternatives of architecture in the early stages of the design process. Some actors implement the computation behavior, some others implement the communication behavior. This dinstinction promotes agility during the exploration phase. Several MoCs are investigated among which, Kahn Process Network, Communicating Sequential Process and Synchronous Data Flow.

The implementation relies on SCALA at first, while Smalltalk and the Biniou framework are used to speed up the process, through aleviating the need for software simulator and compiling to hardware platforms instead.

Keywords Actor, Model of computation, smalltalk, scala

1. Introduction

In 2008 the European Council published the directive 2008/56/EC which obliges European states to check the water quality in their coastal area on the long term. This raises the interest for coastal sea floor observatories. An example of such a project is the MeDON (Marine eData Observatory Network) project an observatory deployed near Brest in France[13]. The usual structure of a cabled sea floor observatory is a network of sensors linked to one or more servers that perform the computations. The drawback of cabled sea floor observatories is the creation of huge amount of data to process and to store. Besides, the data exhibits a lot of redundancy and noise.

According to Spencer et al.[28] one way to reduce this issue is to introduce smart sensors. A smart sensor is a small system that contains, beside a sensing device, a microprocessor to perform computations on the acquired data. The behavior of a smart sensor can be broken down into a set of tasks responsible for the acquisition, processing and dissemination of data. For example, a smart sensor built around an HD camera may be made of a task to acquire images, several tasks to classify forms and a task that manage the network communications. In the case of a camera, the acquisition of data should be continuous. So acquisition and treatments are process that should occur simultaneously. Likewise, the dissemination of data over the network should not pause the acquisition of data. It implies the need to implement concurrent processes in the software of a smart sensor. Designers need to take into account the execution of each process and the inter-process communications (IPC). In the example of the HD camera, designers must decide how many process are devoted to treatments and how data are shared between acquisition and treatments and between treatments and dissemination. A typical question is having only one image processing chain or multiple ones that work in parallel on different part of an image. In addition the number of possible architectural solutions is increased with the number of possible uses of IPC. The set of combinations of breakdown in processes and IPCs make the design space of a smart sensor. Designers must explore this design space. This raises the issue of modeling the concurrency and the communication between the processing elements of the smart sensor and how to execute the obtained model.

In this paper, we present a framework to cheaply and quickly test different alternatives of software architecture. We conform to the actor model described by Agha[2] as the modeling and execution framework. Our goal is implement models of concurrency and communication from an higher level of abstraction also called Model of Computation. Our prototype supports modeling data exchange and synchronization between processes using either Kahn Process Network (KPN)[15], Communicating Sequential Process (CSP)[12] or Synchronous Data Flow (SDF)[20] viewed as Models of Computation. In order to easily implement MoCs, we need a programming structure that natively describes concurrency and communication. The actor model is such a programming structure. We implement KPN, CSP and SDF as actors having the different semantics of communication and synchronization. We obtained an actor based framework of simulation for the functional design of Smart Sensors. Our framework enables reuse by using actors with a well defined functional boundary. Besides, our framework supports the testing of different configuration and communication or synchronization by implementing several Models of Computation. Our prototype is used to:

[Copyright notice will appear here once 'preprint' option is removed.]

- help designers understand the required functions of a smart sensor;
- simulate different alternatives of software architecture;
- refine the alternatives of software architecture until they are stable and precise enough to be analyzed by more precise tools like Ptolemy or Forsyde.

From a practical point of view, we made a first implementation of our framework in Scala. Scala has the advantages of running on the Java Virtual Machine and of being able to integrate existing pieces of code written in Java. Besides, Scala is used by major industrial actors. We also made an implementation in Smalltalk using the Actalk library. Smalltalk has been used in the context of multi-agent systems[23]. An agent formalism describes a network of smart sensors as it can be seen as a network of autonomous entities that communicate with each other in order to perform a mission. Smalltalk is also used to address the issue of programming a virtual machine able to support multiple concurrency mechanisms [22]. In both cases, concurrent processes that have to communicate are involved. The communications between the different processes are an issue. Our framework implemented in Smalltalk can help in testing different solution for the communication.

Another benefit of using Smalltalk lies in the compatibility with some legacy work we been leading since mid 2000s, referred as Biniou [18]. Biniou is a framework that support describing applications as a set of concurrent processes, prior to synthesize the application onto a reconfigurable device. This process, often referred as high level synthesis (HLS), can either rely on global scheduling on distributed scheduling, with either a known behavior or inter modules arbitrary synchronizations. The benefit of using Biniou is to easily stress some design options, while providing a hardware speedup compared to pure software execution. Besides, as Biniou embeds debugging (observability, controllability) features within the generated hardware, this speed up comes at no cost in term of exploration and analysis.

The rest of the paper is organized as follows. Section 2 first describes some related work and summarizes the definitions used in the paper for the main concepts. Section 3 provides more details on the context and our motivations. Section 4 delves into the choices made for the implementation of our framework. Section 5 demonstrates the usage of our framework on a simplified example of Smart Sensor.

2. Related Work

The modeling of embedded systems in general and modeling of sensors in particular has been studied in different ways. This section provides background and focus on the notion of Model of Computation and Actors, as they both drive our modeling.

2.1 Background

SensorML[3] has been used for describing the specification of sensors with a XML format. The XML Schema of SensorML is standardized by the Open Geospatial Consortium. In SensorML a sensor has a series of attributes and may be composed of processes linked together. Robin and Botts[24] demonstrated the use of SensorML to describe chains of processes to analyze acquired data. SensorML describes the relations between processes but does not provide a description of the communication or synchronization mechanisms between processes. In [6], Diallo and al. show the ability of MoCs to describe the communication semantic in models. They define a modeling language called Cometa that enables to model the communication and synchronization mechanism defined by a MoC.

ThingML[8] is a Domain Specific Language (DSL) to model resource-constrained systems such as smart sensors. Fleurey and al. promote a Model Driven Engineering approach using nested state machines to model the behavior. Several model-to-code transformations are defined to target the Java programming language, the Arduino family of board and the Atmel AVR and TI MSP chips. As the final aim of ThingML is to generate code, the designers of the system must be highly confident in their model and transformation engines. As any generated code, the result here is not human readable, and specific tools must be considered to meter the impact of high level (modeling) changes that the designer would operate. Several tools or language libraries implement particular MoCs. Communicating Sequential Process is implemented in the Java Programming Language with JCSP or in Scala with Communicating Scala Objects[30]. A SDF based approach can be found in industrial tools like Scade Suite[31]. However, this different tools are restricted to one Model of Computation. As we want to explore architecture that may mix different MoCs we need a tool that implement different MoCs. Ptolemy[7] is an analysis tool for heterogeneous systems based on Models of Computation. Ptolemy is made of a graphical modeling tool that enables to model the system. The model is executable through an implementation of the Models of Computation in Java. Heterogeneity is handled through nested components. Ptolemy is intended to analyze models of systems. The analysis of the models is only significant if the models are stable. Forsyde[25] is both a tool and a methodology for the design based on Models of Computation of system on chips (SoC). MoCs in ForSyDe are implemented in Haskell. The suggested methodology is based on the use of modeling during the whole design cycle. The model is refined incrementally until the designer gets a model ready for implementation on a SoC.

Our approach is synergetic with these works, as we describe concurrency between processes and to unambiguously explicit their communications and synchronizations. We are addressing designers in charge of design space of the whole system. We offer them facilities to simulate several alternatives of architecture to select those best suitable for the system. Our environment intents to serve

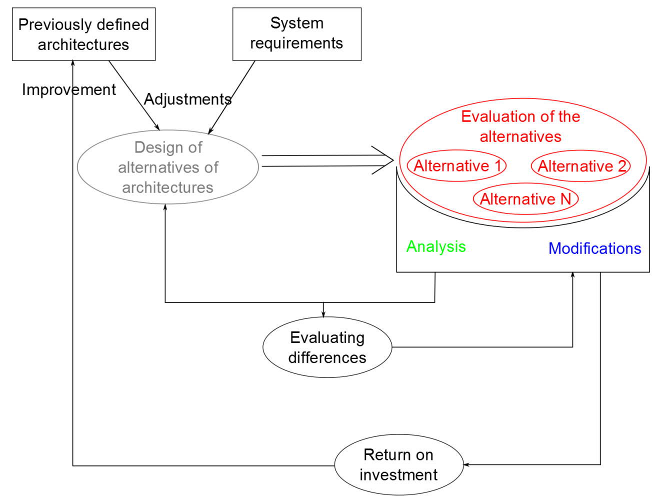

Figure 1. Tool flow overview

as an exploration step, with fast but coarse grained evaluations. Simulating architecture alternatives (central left bubble in figure 1) helps the designer to fully capture the functional requirements. Once done, architecture models are stabilized and refined, and third party tools such ForSyDe or Ptolemy can be controlled to offer more accurate metrics and analyze features. For example, in our framework an entity can use different MoCs at the same level. Our

framework enables to verify that the functionality of the entity is well performed. However, mixing MoCs raises heterogeneity issues [7]. Ptolemy is specifically designed to deal with these issues and enables to validate the entity.

2.2 Models of Computation

A Model of Computation (MoC) for concurrent applications defines[21]:

1. the composition of the concurrent components of the application including the description of how to perform computations;
2. the concurrency mechanisms that govern the execution of the components;
3. the communication mechanisms.

MoCs are used in the design of embedded systems[14] such as smart sensors. They provide an abstraction of the concurrency that enables to design a concurrent system without having to take into account the implementation of concurrency on the actual system. This ability is really useful in the first phase of the design cycle when the final platform may not have been chosen yet.

2.3 Actor Model

The chosen semantics for actor is concurrent entities called actors that run in parallel[2]. An actor is able to send messages to other actors and to receive messages from them. Each actor has its own buffer to receive messages. When an actor sends a message to another it is never blocked. On the contrary, when reading data from its buffer, an actor can be blocked if the buffer is empty.
In some languages such as Erlang[1] or Scala[10], actors are a language features. In other languages such as Java (for example Kilim[29]) or Smalltalk (Actalk[4]), actors are provided through libraries.

3. Context and Motivations

Contrary to standard sensors, smart sensors have a microprocessor that makes them intelligent[28]. A smart sensor is able to acquire data, to perform computations on these data and to send the result of the computations over a network. A smart sensor is also able to modify its behavior according to data sent by other smart sensors. In the context of sea floor observatories, smart sensors are studied because their embedded intelligence enable them to automatically register into the sensor network of the observatory[32]. So the smart sensor can be seen as a plug and play component of the deployed observatory. They are also able to reduce the amount of data that is sent on the network.
A smart sensor is a complex system that mix hardware and software. Faults and errors in such a system come either from software or hardware or their interactions between software and hardware. In the context of sea floor observatories recovering from a failure may require an on-site intervention requiring expensive materials. Besides the underwater environment is very hostile. So hardware failures due to unexpected causes have a high rate of appearance. As the software of a smart sensor might be the most manageable part, a lot of pressure is put on the software engineers. They have to reduce the risks of pure software failures. The software of a smart sensor is made of concurrent processes that communicate and synchronize. This concurrency hides complexity, however, it can be the source of a lot of software failures. In order to ensure the overall quality of their softwares, software engineers use strict development methodologies supported by tools. The INCOSE details a system engineering methodology in [11]. One of the steps is the design of the architecture of the system. It consists in defining different candidates of architecture and then validate them. The different validated architectures are compared to choose the one that best fits to the system requirements. In order to validate the candidate architectures, simulation can be used. We are interested in the concurrent behavior of the system, the communication and synchronization. Simulation requires a way to ease the modeling of these aspects.
In the case of small size systems, it is tempting to develop the software of the smart sensor directly on the final platform. So the software is both a prototype and the final application. This mix may contain portion of code of different quality levels that can cause the failure of the system. Besides, according to the maturity of the project, the final hardware platform is not known in advance. So it is not possible to develop directly on it. This leads to the need of using a prototyping platform that is able to simulate concurrent applications. The concurrency is well defined by Models of computation. MoCs enable reasoning about the system at a higher level, abstracting the low level (platform specific) details. This additional level of indirection has the following advantages as shown in Figure 2:

- it enables to generate software for different platforms from the same model;
- it enables to perform simulation of the system on a platform agnostic simulation framework to validate functional properties;
- it ensures the coherence of the different generated software as communication and synchronization are well-defined.

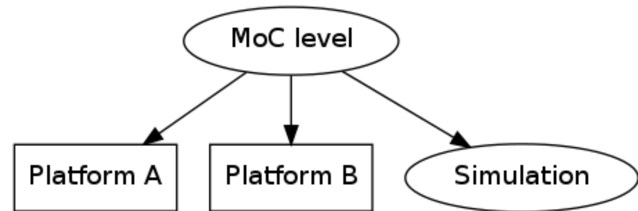

Figure 2. Benefits of using Models of Computations

Besides, design tools of embedded systems such as ForSyDe[25] make a use of Models of Computation. In order to define the architectural alternatives, designers must analyze the functions of the system. A tool that enable to fast prototype the system helps to understand the main functions of a system. It is also a way to communicate with a client to check for the well understanding of his needs. Besides, communication and synchronization mechanisms have a high impact on the architecture of a system. It is mandatory to identify and analyze this impact as soon as possible. So, designers must be able to do experiments with several Models of Computations.
From our point of view, our platform should enable to quickly modify a prototype to test a new version of the code. The platform should also be easily extended by adding new Models of Computation. The Ptolemy project lists twelve different MoCs[5]. However, the list of existing MoCs is not restricted to those listed by the Ptolemy project. So the process of making the implementation of a new MoC compatible with our framework should be made as easy as possible. Last but not least, the platform should enable to create modular prototypes. So processes already developed to prototype one smart sensor should be reused to prototype another smart sensor of the same kind.

4. Architectural choices

4.1 Framework Architecture

The architecture of a concurrent system is made of several computing blocks that execute simultaneously and that exchange data. The choice of the breakdown into blocks and the combination of communication ways between blocks is an architectural alternative. The breakdown in blocks may be made according to the goal of the system. However, the choices of communication mechanisms are wide. Most of the problems such as deadlocks or inconsistency of shared variables. So we choose to focus on the communication mechanisms. In our framework, we did not implement a scheduler to simulate the concurrency. We rely on the concurrency mechanisms provided by the implementation language. We focus on the implementation of different communication mechanisms each of them are defined by a Model of Computation. We need a model that natively describes concurrent processes and their communication. This is the case of the actor model. Besides, in [27] we showed that each concurrent processes of a smart sensor can be made of a thread with a FIFO to store received data. This implementation is similar to the actor model. So we choose to use the actor model as the underlying concurrency mechanism to implement our prototyping platform. The actor model provides modularity. Actors are building blocks that encapsulate their behavior. Besides, actors do not share variables. As the communication with an actor is only based on message exchanges it is possible to replace an actor by another one that is reactive to the same messages.
An actor has a single FIFO to receive the messages from the other actors. Karmani and Agha[17] points out that this mechanism does not guarantee the order of reception of messages by an actor. They suggest the creation of dedicated communication channels between the actors. For modularity reasons, we use the solution suggested by Karmani and Agha. Each channel of communication can be associated to a Model of Computation to formally describe how the messages should be exchanged. Each Model of Computation describes a particular communication and synchronization mechanism.

In order to take into account a variety of behaviors, we capture the functional properties of a given MoC into an actor. This solution offers the benefit of separating the functional behaviors of the application from the behaviors that rule communications.

A naive example of synchronization of concurrent processes is illustrated by table 1. This code creates two processes that communicate through a SharedQueue. As reading is blocking and writing is not, this implicitly describes a Khan Process Network synchronization scheme. The issue is to let the designer change at will the semantic of the synchronization, while keeping constant the topology of the processes network.

```
|channel|
channel := SharedQueue new.
[[true] whileTrue:[channel nextPut:
        self produceData]] fork.
[[true] whileTrue:[self process:
        channel next]] fork.
```

Listing 1. A naïve implementation of Khan Process network

In order to be modular and to enforce reusability, we use the Adapter design pattern[9]. We implement a MoC as an actor that is reactive to two messages:

- Put: message sent by a writer process to send data on the channel;
- Get: message sent by a reader process to read data on the channel.

Figure 3 shows the principles of the architecture of our prototype.

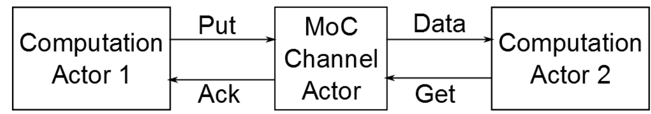

Figure 3. Principle of the implementation

The interesting part is that, once this partition done, evolution becomes easy, bringing a fundamental basis for exploration. A simple implementation is proposed, based on three main classes: Actor, Channel and MocActor. The code in the Actor class does not depend on the used Moc as shown by listing 2.

```
read
"Get"
(self channels at: #input) postRequest: #req.
"Data"
^(self channels at: #input) getAck

write: aData
"Put"
(self channels at: #output) postRequest:
        aData.
"Ack"
(self channels at: #output) getAck
```

Listing 2. The Actor's read and write operations

As illustrated, we have defined a common protocol for data exchange between a concurrent block and a MoC block. To send data, a concurrent block should send a *Put* message to the MoC block and wait for an *Ack* message. To read data, a concurrent block should send a *Get* and wait for a *Data* message. In the case of MoCs that describe non-blocking write operations such as KPN or SDF, the MoC actor performs the sending of the *Ack* message just after the reception of a *Put* message. As the MoC actor does not wait for the reception of a *Get* message, we can simulate a non-blocking write operation. Otherwise, the *Ack* message is sent when the synchronization between the writer and the reader processes can occur.

4.2 Implementation

In the former section, the principles for the implementation have been explained. The next subsections illustrate the implementation of Kahn Process Networks, Communicating Sequential Processes and Synchronous Data Flows. A special focus has been set on the particularities of these MoCs.

4.2.1 Implementing Communicating Sequential Process

Communicating Sequential Process (CSP) has been introduced by Hoare[12]. CSP also describes a network of concurrent processes. In CSP communications are rendez-vous based. Both write and read operations are blocking. It also has a conditional rendez-vous mechanism that introduces indeterminism in the Model of Computation. To achieve that, multiple rendez-vous are started concurrently. But only the first one that can be made, goes to the end. The other ones are canceled once the first one succeeded. Besides, a guard is used to select which conditional rendez-vous must be started.

Communicating Sequential Process are well suited for the modeling of systems requiring tight synchronization between processes. An example is the dining philosopher problem that can be easily solved with CSP.

In our implementation, the rendez-vous between a producer and a consumer is managed by a dedicated Actor called *CspRendezVous*. the *CspRendezVous* actor implements several methods to handle incoming messages and to execute a rendez-vous. Each method follows the same principles:

- they are called on the reception of a trigger event;
- methods called after reception of a message from a consumer wait for a message from a producer and vice-versa;
- if a conditional rendez-vous is canceled then the original message must be put in queue again;

Figure 4 is a extract of the state machine implemented in the *CspRendezVous* actor. It details the reception of a *Get* message from a reading process. A method named *executeGet* is called after the reception of a *Get* message and it waits either for a *Put* message or for a *CondSnd* message.

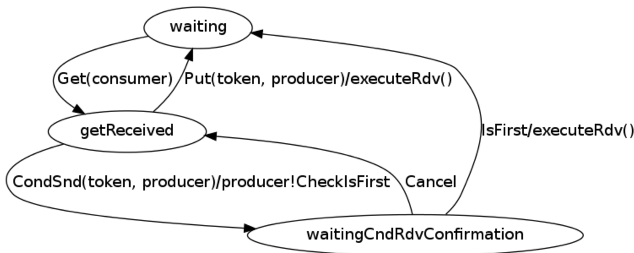

Figure 4. Part of the state machine of *CspRendezVous* to manage the reception of a *Get* message

When a *Put* message is received the method *executeRdv* is called. This method performs the rendez-vous by sending an *Ack* message to the producer and a *Data* message to the consumer.
When a *CondSnd* message is received, a *CheckIsFirst* message is sent to the producer. This message informs the producer that the rendez-vous is possible. The producer checks if it is the first rendez-vous that is able to complete. If that is the case, it sends an *IsFirst* message to the *CspRendezVous* actor which calls the *executeRdv* method. In the other case, the *CspRendezVous* actor receives a *Cancel* message from the producer and simply sends to itself a *Get* message to restart the rendez-vous as the consumer tries a non-conditional rendez-vous. An example of the sequence of messages sent during a conditional communication is shown Figure 5.
Conditional rendez-vous are initiated by a specific actor which is able to start several actors which are competing to perform a rendez-vous. This actor reacts to a *Get* or a *Put* message sent by an application actor. It also checks which rendez-vous is successful at the first place and cancels the others.

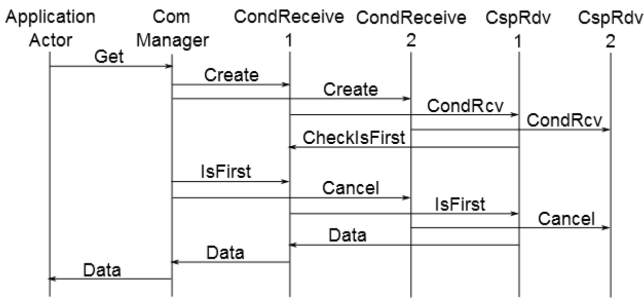

Figure 5. Example of sequence of messages sent during a Conditional communication

4.2.2 Implementing Kahn Process Networks

Kahn Process Network is a Model of Computation that describe a network of communicating parallel process[15]. The process network is an oriented graph. In KPN, the communication are made through non-blocking write operations and blocking read operations. The communications are made through channels made of an infinite FIFO. Processes in KPN can be created during the execution of the process network.

Kahn Process Network is useful in the modeling of systems based on data flows. Signal processing or scientific computing applications are example of data flow applications.

Our implementation is based on the proposal of Kahn and McQueen[16]. The FIFO of a channel is managed by a dedicated actor called KpnChannel. The attributes of this actor are:

queue The FIFO managed by the actor.

hungryConsumer A consumer of data that is blocked on a read operation on the FIFO.

isFinishReceived The flag that indicates that no more data will be received from the producer.

monitor A specific actor that manages the different FIFOs.

The KpnChannel actor may receive three different messages:

- Get,
- Put,
- Finish.

Figure 6 describes the state machine implemented in the actor *KpnChannel*. When receiving a *Get* message, the *KpnChannel* actor checks if its *queue* is empty. If there are data in the *queue*, the *KpnChannel* sends the head of the *queue* to the consumer in a *Data* message. Otherwise, the *KpnChannel* checks if the *isFinishReceived* flag is set. In such a case, the *KpnChannel* sends the *Finish* message to the consumer and the *KpnChannel* kills itself. Otherwise, the attribute *blockedConsumer* is set and a message *ReadBlocked* is sent to the *monitor*. Then the *KpnChannel* waits for new messages.
When receiving a *Put* message, the *KpnChannel* first sends an *Ack* message to the producer. Then it checks if there is an *hungryConsumer*. In such a case, the *KpnChannel* sends a message *Data* to the *hungryConsumer* and a message *ReadUnblocked* to the monitor. The attribute *hungryConsumer* is set to *null* and the *KpnChannel* waits for new messages. Otherwise, the data received from the producer are enqueued and the *KpnChannel* waits for new messages.
When receiving a *Finish* message, if there is an *hungryConsumer*, the *KpnChannel* sends to it a *Finish* message and kills itself. Otherwise, the flag *isFinishReceived* is set to true and the *KpnChannel* goes on waiting for new messages.
The attribute *monitor* is a reference to an actor of kind *KpnMonitor*. Its role is to detect deadlocks. A deadlock occurs when all active process are blocked into a read operation. So the *KpnMonitor* keeps a count of the active process and of those blocked in a read operation.

4.2.3 Implementing Synchronous Data Flow

Synchronous Data Flow (SDF[20]) describes communication with non-blocking write and blocking read operations as KPN. SDF adds the constraint that the production or consumption rates on the communication links are constant and well known. As a result SDF offers the ability to pre-determine a scheduling of the different processes. A SDF process begins by reading the required amount of data on each of its inputs, then performs its processing and finally

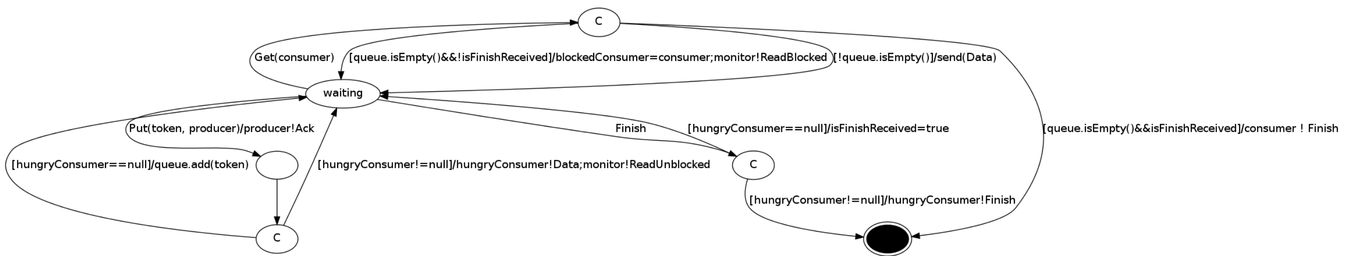

Figure 6. State machine of the actor *KpnChannel*

writes a given amount of data on each of its outputs.

As KPN, Synchronous Data Flow is also well suited to model data flow applications. Unlike KPN, SDF can only be used when the data production rate of the different processes are known. It is the case in most of the signal processing applications.

Our implementation use a multithreaded dynamic schedule as described by Schaumont[26]. The different actors in the application work concurrently. The scheduling of the actors is left to the underlying virtual machine. The scheduling is not made explicit in a dedicated actor that activates the others in a predefined order.
The actor *SdfChannel* has an attribute *nbElemToRead* which defines how much data must be read by the consumer at each read operation. If there is not enough data in the FIFO then the read operation is blocked until the required amount of data is reached.
The *SdfChannel* actor receives either a *Get* or a *Put* message. The content of a *Put* message is a list of data. This list is dequeued to be inserted in the FIFO.
When a Get message is received, the size of the FIFO is compared to the *nbElemToRead* attribute. If it is superior or equal, *nbElemToRead* elements of the FIFO are dequeued to populate a List of data that is sent to the consumer. Otherwise, the consumer is blocked until enough data is available.

4.3 Using Scala and Smalltalk

We made two full implementation, an implementation in Scala and one in Smalltalk. Scala and Smalltalk both provide interesting programming mechanisms. An example is the collection manipulation methods such as *map* and *filter* in Scala and *collect* and *select* in Smalltalk. These methods ease the manipulation of data structures. However, the type system is the big difference between Scala and Smalltalk. Even if a type inference mechanism is implemented in Scala, the Scala programming language remains a statically typed language. It has the advantage of raising some errors at compile time, however, the type of each variable has to be either inferred from the context or made explicit by the programmer. It has an impact on the use of generic messages for the communication between actors.

```
process

[[| ack data |
"read request, may be blocking"
ack :=(self channels at: #output) getRequest.
"read data, may be blocking"
data :=(self channels at: #input) getRequest.
(self channels at: #output) postAck: data.
(self channels at: #input) postAck: ack]
repeat] fork
```

Listing 3. The CSP MoC in Smalltalk

For example, the *Put* message as an argument with the generic type *Data* (equivalent of *Object* in Smalltalk). The processing of the *Data* message in a consumer process requires type checking and casting before a clean use of the value carried by the *Data* message. On the contrary, Smalltalk is dynamically typed. So there is need of type checking and casting. This produces a more readable code. The code does not contain programming constraints due to the programming language. Besides, dynamic typing seems more suited for fast prototyping purposes as a change of type of a variable does not have effects on the whole code.

```
process
"Two processes"
[[
"read data, may be blocking"
self fifo nextPut:
    (self channels at: #input) getRequest.
"automatic acknoledge"
(self channels at: #input)
    postAck: #acknoledge
]repeat] fork.

[[
"read request, may be blocking"
(self channels at: #output) getRequest.
"fifo access request, may be blocking"
(self channels at: #output) postAck:
      self fifo next
]repeat] fork
```

Listing 4. The KPN MoC in Smalltalk

The listings 3 and 4 illustrate how the bahavior divergence is implemented. The MocActor object (MocActor being an abstract common super class to CSPMocActor and KPNMocActor) implements the process method, which schedules the get/ack operations over the IOs channels. Every channel has two SharedQueues and support posting and requesting operations over them. Besides, the MocActor owns an additional SharedQueue named FIFO. This shared queue supports temporary storage of data for KPN and SDF Moc. It's useless for CSP, though.

5. Experimentation

5.1 Description of the Experimentation

In the MeDON project[13], we deployed a high definition camera. This camera produces a lot of images that are manually processed. The major drawbacks of this solution are:

- it requests manpower to analyze the acquired images;

- non-relevant data are unnecessarily stored waiting for processing.

Our solution is creating a smart sensor using the HD camera as the sensing device. The smart sensor will embed image processing algorithms. These algorithms are used on the acquired data before they are sent to a ground-based server. In the following, we simplified the example. The acquisition part consists in reading an image file on the disk. An example of image processing algorithm is the Sobel edge detection algorithm. The data sending consists in writing the result of the Sobel algorithm on the disk.

5.2 Exploring the Logical Architecture Alternatives

The exploration phase intends to determine the best solution when describing the architecture as a set of communicating processes. Not all of the processes have a direct mapping with a physical device or sensor. Instead, the analysis focuses on functions being executed, with a tradeoff to be found out, between simplicity of simulation and accuracy of the modeled behavior.

5.2.1 Presentation of the Possible Architectures

Multiple architectures may be used for this example. The simplest alternative makes each block of the functional architecture become a concurrent entity. Another possible breakdown is to decompose the Sobel algorithm into multiple concurrent entities. This architectural alternative is shown Figure 7.

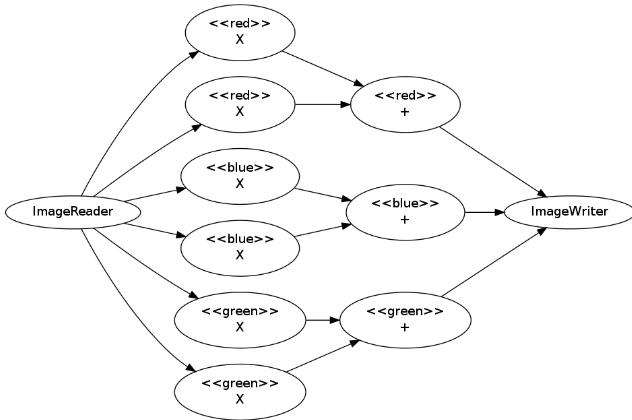

Figure 7. Alternative for the logical architecture

The *ImageReader* is responsible for reading the image from a file. For each pixel and for each color plane, the *ImageReader* creates a list of nine elements containing the different values associated to a pixel and its neighbors.
For each color plane, two convolution and a sum computations are required. Each of them are transformed into concurrent entities The *ImageWriter* is responsible for creating an image from the values coming from the sums.
In the rest of the paper, we focus on the second alternative. As there are several concurrent entities, we need to ensure the communication and synchronization of these entities. For example, we need to ensure that the sum is performed on data that concerns the right pixel.

5.2.2 Implementation using only CSP

The topology described in Figure 7 can be implemented directly using actors. However, the order of the messages received by the different actors is not guaranteed.
One possible solution to this issue is to use a highly synchronized system (rendez-vous based). This ensures that producer and consumer work at the same rate. As no data is produced until the previous one is not consumed there will be no inversion of data. However, this imposes strong constraints on the real system. It limits the number of processes able to run in parallel.
Each ellipse in Figure 7 is implemented as an actor with a computation role. Each arrow is implemented by a *CspChannel* actor. Each *CspChannel* actor handle the rendez-vous between two computation actors as described by CSP. The code of a computation actor contains instructions to send messages to and receive messages from *CspChannel* actors. The only impact on computation actor of the use of our framework rather than a purely actor based implementation is the lines of code to handle the communication with the MoC actors. The modified topology of the application, modeled based on CSP, is shown in Figure 8. The *CspChannel* actors that are added are represented in black.

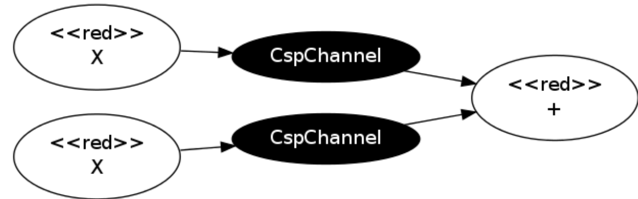

Figure 8. Topology of the application using CSP

The use of Csp prevents an actor from acting on data before the tasks of the previous actors in the flow are not completed on their own input data. As a result, the order of the data at the inputs of the *SumActor* is kept.
The use of our implementation does not affect the logic of the actors composing the system but only the way data are exchanged. This agility only requires that for each sending of data we add code to receive the acknowledge from the MoC actors. In the case of receiving data, we have to add code to send a get request to the MoC actor and the code to wait for its answer.

5.2.3 Alternative using only KPN

In our previous implementation, we used CSP to make the different actors communicate. It ensures the correct treatment of the pixels making the input image. CSP puts strong constraints on the system. As the synchronization between the application actors is obtained through rendez-vous, the number of actors that are able to work concurrently is limited. This may reduce the overall performance of the application.

Be these constraints useful when developing an application, especially for simplifying the debugging process through offering a more sequential scheme to the designer, this level of constraints may not be necessary in the final application. Exploration - as previously states - remains one of our more critical motivations beyond this work. The ability to switch between different MoCs is a key facility to support agility and incremental refinements at no cost in term of readability and understanding versus final performance tradeoff.

Another alternative to CSP channels, is to use a FIFO per required communications between two computation actors. This reduces the risks of reading data coming from the same sender twice. Besides, blocking read operation ensures that the consumer process will wait for incoming data. This corresponds to the KPN Model of Computation.
Contrary to CSP, KPN enables a process that produces data to keep running while the consumer of its data is also running. This allows

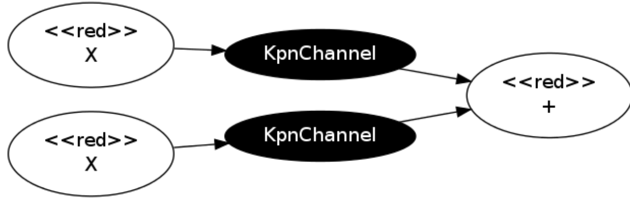

Figure 9. Modification of the topology using KPN channels

to have more processes running in parallel than with CSP. However, there is a lost of control over the communications. It is not possible to ensure that the system will have enough memory for the different FIFOs. We made an implementation of this alternative with our framework using *KpnChannel* actors. No modification is required on the application actors as we defined a common interface for all implementation of MoCs. The only changes occurs in the description of the topology of the application to obtain the topology shown in Figure 9. New instances of kind *KpnChannel* has to be created to replace instances of *CspChannel*. We are able to quickly change the type of communication or synchronization between actors in the application we would like to prototype.

5.3 Benefits of Smalltalk's debugger

From our point of view, in these experiments the main advantage of Smalltalk over Scala is its debugger.
Firstly the Smalltalk's debugger is integrated in the development environment. When a fault occurs in an application the debugger is started. It enables to perform a post-mortem analysis of the application. It is possible to check the state of the different components of the application. On the contrary, Scala only provides a message corresponding to the exception that occurs in the running program. The advantage of Smalltalk is the ability to analyze deeply the reason of a failure as all information are available.
Secondly the Smalltalk's debugger enables to make live modification on the code of the running application and on the values of the different variables. The Scala debugger only to modify the values of the variables. The Smalltalk's debugger provides the ability to make a correction on the application and to continue the execution. It is an asset when performing fast prototyping as there is no need to perform the *Code - Compile - Run* cycle again.

Besides, the Redpill [19] environment reproduces most of the smalltalk debugger features at a hardware level. This is critical as only hardware emulation can support scalability, and the medium term aim is to address massive sensor networks. As Redpill has been developed using cincom Visualworks, and is Moc oriented - despite only CSP is supported at this time - it offers a sound path to integrate our modeling and evaluation framework with hardware synthesis. Not only extending the set of supported MoCs makes sense, but it is part of our strategic research plan and is at the heart of several research projects.

6. Conclusion and Future Work

When programming an application, the source code is the primary interest of engineers. In the context of developing the software of a smart sensor, attention must be drawn on sensing, processing data and networking. Thus the complexity of the software of a smart sensor is higher than the one of the software of a simple sensor. Having a higher level of abstraction to consider the programming of a smart sensor is useful to resolve early problems like these associated to IPCs. This statement points out the need to define a generic model for IPC and synchronization mechanisms. These mechanisms are many, and this model will be the key to flexibility, evolutivity and ease of domain space exploration.

In this paper we explored the use of different Models of Computations for modeling smart sensors. We create dedicated actors for the behavior of the communication described by the MoCs. This enables to separate the computations from the communications and synchronizations. Besides, we define a common protocol of data exchange between the computation actors and the MoC actors. This enables to have a modular simulation framework for concurrent applications defined with Models of Computations. This framework can be used to help to define candidate architecture for concurrent applications. Future works include realizing hardware emulation of such smart sensors, with no loss in term of observability and controllability of the execution. This will offer both faster execution and scalable modeling. This direction takes a direct benefit from the RedPill framework. Next, system integration will be considered. Multiple abstraction layers and on-demand refinements will support addressing (smart) sensor networks. It's a second dimension for scalability, with qualitative enhancements in addition to quantitative scaling. This second direction will benefit from previous work that we have led on the Cometa and Biniou frameworks.

Acknowledgments

This work has been done with the financial support of the French Délégation Générale de l'Armement and of the Région Bretagne.

References

[1] Erlang programming language. URL `http://www.erlang.org/`.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[3] M. Botts and A. Robin. Opengis sensor model language (sensorml) implementation specification. *OpenGIS Implementation Specification OGC*, pages 07–000, 2007.

[4] J.-P. Briot. Actalk: A testbed for classifying and designing actor languages in the smalltalk-80 environment. In *Proceedings ECOOP*, volume 89, pages 109–129, 1989.

[5] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains). Technical report, EECS Department, University of California, Berkeley, Apr 2008.

[6] P. I. Diallo, J. Champeau, and V. Leilde. An approach for describing concurrency and communication of heterogeneous systems. In *Proceedings of the Third Workshop on Behavioural Modelling*, BM-FA '11, pages 56–63, 2011. ISBN 978-1-4503-0617-1.

[7] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. In *Proceedings of the IEEE*, pages 127–144, 2003.

[8] F. Fleurey, B. Morin, A. Solberg, and O. Barais. Mde to manage communications with and between resource-constrained systems. In *Model Driven Engineering Languages and Systems*, pages 349–363. Springer, 2011.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[10] P. Haller and F. Sommers. *Actors in Scala. Concurrent Programming for the multi-core era.* Artima, 2012.

[11] C. Haskins, K. Forsberg, and M. Krueger. Systems engineering handbook. *INCOSE. Version*, 3.2, 2010.

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL `http://doi.acm.org/10.1145/359576.359585`.

[13] Interreg IVA. Marine edata observatory network, 2013. URL `http://medon.info/`.

[14] A. Jantsch and I. Sander. Models of computation in the design process. 2005.

[15] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[16] G. Kahn and D. Macqueen. Coroutines and Networks of Parallel Processes. Rapport de recherche, 1976. URL `http://hal.inria.fr/inria-00306565`.

[17] R. K. Karmani and G. Agha. Actors. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1–11. Springer, 2011. ISBN 978-0-387-09765-7.

[18] L. Lagadec and D. Picard. Software-like debugging methodology for reconfigurable platforms. In *IPDPS*, pages 1–4. IEEE, 2009.

[19] L. Lagadec and D. Picard. Smalltalk debug lives in the matrix. In *International Workshop on Smalltalk Technologies*, IWST '10, pages 11–16, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0497-9. doi: 10.1145/1942790.1942792. URL `http://doi.acm.org/10.1145/1942790.1942792`.

[20] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987. ISSN 0018-9219. doi: 10.1109/PROC.1987.13876.

[21] E. A. Lee and S. A. Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Lee & Seshia, 2011.

[22] S. Marr. *Supporting Concurrency Abstractions in High-level Language Virtual Machines*. PhD thesis, Software Languages Lab, Vrije Universiteit Brussel, January 2013.

[23] R. Robbes, N. Bouraqadi, and S. Stinckwich. An aspect-based multi-agent system. *ESUG 2004 Research Track*, page 65, 2004.

[24] A. Robin and M. E. Botts. Creation of specific sensorml process models. *Earth System Science Center-NSSTC, University of Alabama in Huntsville (UAH), HUNTSVILLE, AL*, 35899, 2006.

[25] I. Sander and A. Jantsch. System modeling and transformational design refinement in forsyde [formal system design]. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23 (1):17–32, 2004.

[26] P. R. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2012.

[27] J.-P. Schneider, J. Champeau, D. Kerjean, O. K. Zein, Y. Auffret, and L. Dufrechou. Domain specific modelling applied to smart sensors. In *OCEANS, 2011 IEEE-Spain*, pages 1–6. IEEE, 2011.

[28] B. Spencer Jr, M. Ruiz-Sandoval, and N. Kurata. Smart sensing technology: opportunities and challenges. *Structural Control and Health Monitoring*, 11(4):349–368, 2004.

[29] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP 2008–Object-Oriented Programming*, pages 104–128. Springer, 2008.

[30] B. Sufrin. Communicating Scala Objects. In P. H. Welch, S. Stepney, F. Polack, F. R. M. Barnes, A. A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson, editors, *Communicating Process Architectures 2008*, pages 35–54, sep 2008. ISBN 978-1-58603-907-3.

[31] E. Technologies. Scade suite, 2013. URL `www.esterel-technologies.com/products/scade-suite/`.

[32] D. M. Toma, T. O'Reilly, J. del Rio, K. Headley, A. Manuel, A. Broring, and D. Edgington. Smart sensors for interoperable smart ocean environment. In *OCEANS, 2011 IEEE-Spain*, pages 1–4. IEEE, 2011.