Towards Runtime Adaptivity by using Models of Computation for Real-Time Embedded Systems Design

Denis S. Loubach[1], Eurípedes G. O. Nóbrega[1], Ingo Sander[2], Ingemar Söderquist[3], Osamu Saotome[4]
{*dloubach, egon*}*@fem.unicamp.br, ingo@kth.se, ingemar.soderquist@saabgroup.com, osaotome@ita.br*

[1]Advanced Computing, Control & Embedded Systems Lab, University of Campinas - UNICAMP, 13083-860, Campinas, SP, Brazil

[2]Electronics and Embedded Systems Department, KTH Royal Institute of Technology, Stockholm, Sweden

[3]HMI & Avionics Department, Aeronautics, Saab AB, Linköping, Sweden

[4]Electronics Engineering Division, Aeronautics Institute of Technology - ITA, São José dos Campos, SP, Brazil

Abstract

Considering the aeronautics trend to the second generation of integrated modular avionics adoption, dynamic adaptive real-time embedded systems become more and more important. In this context, reconfiguration capability has posed itself as a new challenge for future avionics design. Modern field-programmable gate array (FPGA), which also comes in system on chip (SoC), offers runtime partial reconfiguration. This capability can be used to adapt at runtime to changes in an uncertain environment or due to internal faults, for instance. Also, these SoCs are capable to deliver significant computational power, including hard/soft multiprocessors, memories, and hard multipliers that can be combined to implement complex designs. That makes FPGA-based systems interesting as processing nodes for next generation of avionics systems. Addressing the aforementioned issues, this paper shows that runtime adaptivity is feasible to be performed through a case study implementation on both software and hardware. To capture and deal with the complexity of adaptivity, the system is modeled by using the theory of formal models of computation (MoC), mainly the synchronous MoC. By the synchrony hypothesis (instantaneous computation and communication) outputs are synchronous to inputs. Besides, this MoC has similarity with clock-driven digital circuits, making it attractive for hardware synthesis. Adaptivity may have two facets: (i) reconfiguration triggered by performance/power consumption constraints; and (ii) regeneration, which is also a reconfiguration, but triggered by the fault handling mechanism thus allowing the system to keep with a minimum functional level.

INTRODUCTION

Considering aeronautics current trend towards the second generation of integrated modular avionics (IMA2G[1]) adoption, runtime adaptive real time embedded systems with mixed criticality[2] become of

[1]The literature also presents the term ”Distributed Integrated Modular Avionics” (DIMA) for the second generation of IMA (IMA2G)

[2]From the IMA1G concept, *i.e.*, multiple software functions with different criticality levels on a single avionic computing module. That was to keep the weight, volume and costs within reasonable limits

fundamental importance. Devising new architecture and system avionics concepts are urgent matters, exploring recent advances on digital systems and aiming to significantly improve the balance between performance and power consumption.

Avionics systems have unavoidably to deal with mandatory constraints such as safety and dependability. Critical subsystems may be completely handled by software, and it is required that those subsystems work correctly in all situations, complying with the set of constraints. A catastrophic failure has to be avoided, and the system is expected to deliver its functionality even in a degraded mode.

In this context, reconfiguration capability has posed itself as a new challenge for future avionics architectures. For IMA2G, reconfiguration means the reallocation of functions to safe modules. Nevertheless, our paper deals with a model for reconfiguration on both software and hardware.

The introduction of the first generation of integrated modular avionics (IMA1G), based on centralizing and reutilization of computer power, implied in increasing complexity and difficulty to comprehensive data analyses, when compared to the federated architecture where every function was performed by an exclusive processing system [Wolfig and Jakovljevic, 2008, Bieber et al., 2012].

IMA1G comprises basically two complementary principles. The first one is about integrating multiple software functions, with possible different criticality levels, on a single avionic computing resource system. Nevertheless, this resource sharing principle brought some side effects and non-functional dependencies among software functions. Proving system safety requires the global knowledge of the system. It cannot be achieved by pieces in separate ways. As a second principle, it was intended to simplify the design process and receive certification. As a consequence, the partitioning concept was introduced, allocating to each function its own non-shared virtual resources to prevent function interference on each other. Then, functions are partitioned considering space (resources partitioning) and time (temporal partitioning) [Bieber et al., 2012].

IMA1G was intended to provide flexibility, interoperability, and integration, and consequently less equipment and cost, and easy of certification. But fault tolerance and containment, both easily achieved on the federated architecture, and also reconfigurability, became focal points for the next version of the concept [Andrillon and Aviation, 2013]. Distributed IMA became the natural solution, mixing both approaches for a better solution, leading to IMA2G that is now concentrating researches and development efforts worldwide (*i.e.*, Boeing, Airbus, Saab).

Modern field-programmable gate arrays (FPGAs) are capable to delivery significant computational power, including hardware/software multiprocessors, memories, and hardware multipliers, which can be combined to implement complex system design solutions, especially if it demands reconfigurability [Sterpone and Ullah, 2013]. More important, this reconfigurability may be *fast enough* to be used in runtime, which makes FPGA-based systems an interesting component for future avionics systems.

Nevertheless, it is still needed to perform further investigations aiming to cover heterogeneous embedded systems considering features such as reconfigurability.

In this context, our paper shows that runtime adaptivity is feasible to be performed through a case study implementation on both software and hardware. To capture and deal with the complexity of adaptivity, the system is modeled by using the theory of formal models of computation (MoC), mainly the synchronous MoC. Within this MoC, a process reads its inputs and computes the outputs at each event cycle (discrete time). By the synchrony hypothesis (instantaneous computation and communication) outputs are synchronous to inputs. Besides, this MoC has similarity with clock-driven digital circuits, making it attractive for hardware synthesis.

The remainder of this paper is organized as follows. Next section presents the main concepts and definitions used along with our research. After, the case study section introduces the system specification, in a high-level of abstraction, implementation specification, and the implementation details of our modeled system. Next, we present the results related to the models implementation on both software and hardware considering a heterogeneous SoC. Finally, we present the paper conclusions and some possible future works.

BACKGROUND

This section introduces the main models of computation (MoC) concepts used in this paper, along with ForSyDe modeling framework, real-time embedded systems and adaptivity.

Models of Computation

Different kinds of models exist to address different purposes. For instance, functional modeling is used to address the functional behavior of a system. On the other hand, design and synthesis refine into implementation details [Jantsch, 2003].

Definition 1. *Model. An abstraction or simplification of an entity that can be physical system or even another model [Jantsch, 2003].*

Definition 2. *Abstraction. Comprises a way for choosing which information or aspects to consider when modeling a system [Jantsch, 2003].*

A **model of computation (MoC)** is an abstraction of a physical computing device, and so, different MoCs serve different objectives. The purpose of a MoC drives how it is designed and what are the properties exposed [Jantsch, 2005]. That author also claims that a MoC has to support implementation independence, composability, and analyzability.

In this context, MoCs are based on *processes*, *events* and *signals*. The following definitions are derived from [Jantsch, 2003, Sander and Jantsch, 2008].

Definition 3. *Event. Elementary information unit exchanged between processes.*

Definition 4. *Signal. Processes communicate to each other by writing to and reading from signals. A signal is a sequence of events. Signals preserve the order that events are entered. Each event has a tag and a value. Tags can be used to model physical time, the events order and other key properties of a MoC.*

Definition 5. *Process. Receives and send events. The activity of a process is comprised of evaluation cycles, which stands for a function application. Then, in each evaluation cycle the process receives inputs, computes, and sends outputs. Generally, a process can have internal states or not.*

Definition 6. *Process Constructor. Parameterizable templates for instantiating processes, as a higher-order function that receives a function as input and returns a function as output.*

Then, processes constructors are used to create processes. New processes can be created to form a hierarchical concurrent process network.

Definition 7. *Model of Computation. Set of process and process networks that can be implemented by a set of processes constructors.*

ForSyDe Modeling Framework

Formal System Development (ForSyDe) [Sander and Jantsch, 2004] is a transformational design methodology based on the functional programming paradigm. ForSyDe targets heterogeneous embedded systems [Jantsch, 2003].

A system is modeled as a hierarchical concurrent process network in ForSyDe. Processes communicate with each other by signals. ForSyDe supports several different MoCs.

In ForSyDe we model a *signal* as a list of *events*, where the event's *tag* is implicitly given by the event's position in the list. The semantics of a tag is defined by the used MoC, *e.g.*, an identical tag of two events in different signals does not imply that these events happen at the same time instant. All events in a signal must have values of the same type [Sander and Jantsch, 2008].

Synchronous Model of Computation

Synchronous MoC split the time domain into slots, where everything inside a slot occurs at the same time. The evaluation cycle of processes lasts exactly one time slot in synchronous MoC [Jantsch, 2003].

Definition 8. *Perfect synchrony hypothesis. Neither computation nor communication takes time.*

Following this hypothesis, the *timing behavior* is simply defined by the arriving of input events considering the system handles input samples in zero time and waits until next input arrives.

Synchronous processes consume/produce exactly one event on each input/output in each evaluation cycle. This implies *total order* of all events in any signal from a synchronous MoC. Events with the same tag appear at the same time instant.

One may conclude that the hypothesis from Definition 8 does not hold for physical systems. However, the hypothesis holds for simulated models and if a model is based on that hypothesis it should behave in the very same fashion as a physical model implementation provided the physical systems responds *fast enough*. In this context, **fast enough** means to compute outputs before the next input arrives, *i.e.*, to apply a function on an input and get its result ready within the same evaluation cycle. This way we can handle functional and temporal behavior separately from each other. Then, we focus on system functionality first (working based on perfect synchronous hypothesis), and so during the implementation step we have to validate the *fast enough* criteria.

The $mapSY_n$ ForSyDe combinational process constructor, required to model a system according to the synchronous MoC, is illustrated in Figure 1 [Sander and Jantsch, 2008].

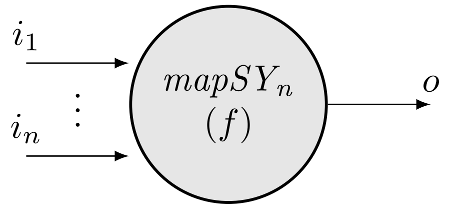

Figure 1: Process constructor for $mapSY_n$ [Sander and Jantsch, 2008]

$$p = mapSY(f)$$

where:

$$o = p(i_1, \ldots, i_n)$$
$$o[k] = f(i_i[k], \ldots, i_n[k])$$

The $mapSY_n$ process constructor takes a function $f : D_1 \times \ldots \times D_n \rightarrow E$ as argument and returns a process $p : S(D_1) \times \ldots \times S(D_n) \rightarrow S(E)$ with no internal state.

SY stands for the synchronous MoC.

For the sake of clarity, we are using the index n for processes inputs in ForSyDe definitions.

Real-Time Embedded Systems

A number of works can be found in the literature considering reconfigurable real time embedded systems. Just to mention some, there are the application of heterogeneous CPU/FPGA for avionics test application [Afonso et al., 2013]; ERA project (Embedded Reconfigurable Architectures) [Wong et al., 2011]; MORPHEUS project (Multi-purpOse dynamically Reconfigurable Platform for intensive HEterogeneoUS processing) [Voros et al., 2013]; and ACROSS Project (ARTEMIS CROSS-Domain Architecture) [Salloum et al., 2012].

Adaptivity

In this section we present the definitions related to adaptivity used along with this paper.

Definition 9. *Adaptive. An adaptive system possesses a certain level of intelligence to actively change its configuration based on the state of the system itself or its environment. Adaptiveness can be enabled by reconfigurable computing or runtime reconfiguration. Adaptive computing is stated as one of the reconfigurable computing research fields.*

Reconfigurable computing comprises computation studies aiming at the use of reconfigurable devices. Then, for a specific application and a specific time frame, the reconfigurable devices' spatial structure is changed to comply with a given objective [Bobda, 2007, Koch, 2013]. Reconfigurable computing is intended to fulfill the distance between hardware and software, aiming higher performance than software and keeping higher level of flexibility compared to hardware [Compton and Hauck, 2002].

Definition 10. *Runtime Reconfiguration (RTR). Basically, runtime reconfiguration is the possibility to change a reconfigurable device's functionality while that device is still executing something and without stopping it. One can reconfigure a partition without affecting others by using partial reconfiguration.*

Definition 11. *Partial reconfiguration (PR). A reconfigurable processor or reconfigurable device supports partial reconfiguration if it allows its programmable area to be divided into one or more partitions and each partition can be reconfigured/changed independently from each other and without stopping others.*

Definition 12. *Full reconfiguration (FR). A reconfigurable processor or reconfigurable device supports full reconfiguration when it considers the whole programmable area as just one reconfiguration area. In this case, the functionality can be changed at the cost of stopping the entire programmable area.*

Modeling of Adaptivity

We use the concept for modeling adaptivity introduced by [Sander and Jantsch, 2008], where the key is to use *functions* in the same fashion as regular data types variables, then signals can also carry functions.

It is possible to create an *adaptive process* with the concept of signals carrying functions (s_f). In this case, function adaptivity is feasible to be implemented through the use of a *reconfigurable device*, therefore functions can be loaded into the partitions in runtime. Moreover, we assume that adaptation is instantaneous at the highest level of abstraction.

The definition of the adaptive process $reconfigSY$ is given next.

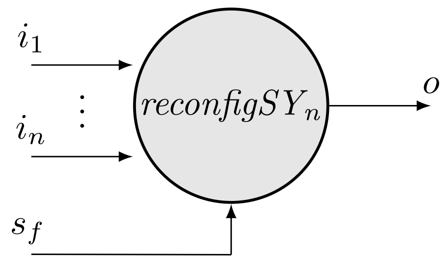

Figure 2: Process constructor for *reconfigSY* [Sander and Jantsch, 2008]

$$ap_f = reconfigSY_n$$

where:

$$o = ap_f(s_f, (i_1, \ldots, i_n))$$
$$o[k] = s_f[k](i_1[k], \ldots, i_n[k])$$

reconfigSY synchronous reconfigurable process can be modeled by using the *mapSY* ForSyDe process constructor together with the function application operator (\$)[3]:

$$reconfigSY = mapSY(\$) \tag{1}$$

Besides, *reconfigSY* can also be modeled by using the *zipWithSY* process constructor along with the same function application operator (\$):

$$reconfigSY = zipWithSY(\$) \tag{2}$$

The *zipWithSY* applies a pairwise function f onto two synchronous signals.

CASE STUDY

We implemented an encoder/decoder system [Sander and Jantsch, 2008], illustrated in Figure 3, as our real-time embedded adaptive system case study using the synchronous MoC.

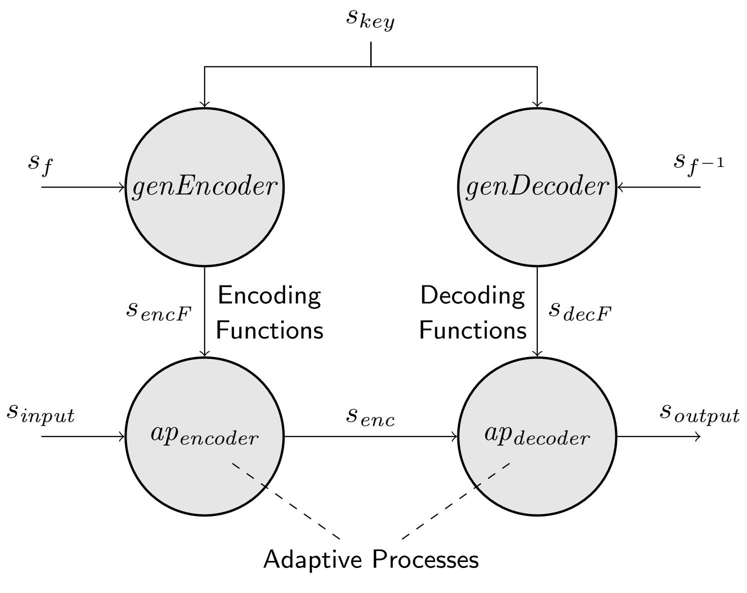

Figure 3: Encoder and decoder system, adapted from [Sander and Jantsch, 2008]

[3]The function application operator allows functions applications on signal values. This is defined as f \$ x = f x.

In that system (Fig. 3), we have two adaptive processes: $ap_{encoder}$ and $ap_{decoder}$. A signal $s_{input}[k]$ is encoded with and encoding function $s_{encF}[k]$ given by $s_f[k](s_{key}[k])$, which is the $genEncoder$ process output signal. Following this hierarchical processes network, the encoded signal $s_{enc}[k]$ is taken by the $ap_{decoder}$ process together with the decoding function $s_{decF}[k]$ given by the $s_{f^{-1}}[k](s_{key}[k])$. Finally, the $ap_{decoder}$ produces the decoded signal $s_{output}[k]$.

Next, we present the summary of signals definition.

- s_{key} : signal comprising the keys used in the encoding/decoding process
- s_f : signal containing the encoding functions f
- s_f^{-1} : signal containing the decoding functions f^{-1}, which stands for the inverse of encoding function
- s_{encF} : signal comprising the encoding functions along with the respective keys
- s_{decF} : signal comprising the decoding functions along with the respective keys
- s_{input} : plain-text-like signal to be encoded
- s_{enc} : signal comprising the encoded data
- s_{output} : signal comprising the decoded data

The processes definition is given next.

$$genEncoder : s_f[k](s_{key}[k]) = s_{encF}[k] \tag{3}$$

$$ap_{encoder} : s_{encF}[k](s_{input}[k]) = s_{enc}[k] \tag{4}$$

$$genDecoder : s_{f^{-1}}[k](s_{key}[k]) = s_{decF}[k] \tag{5}$$

$$ap_{decoder} : s_{decF}[k](s_{enc}[k]) = s_{output}[k] \tag{6}$$

Follow we present the listing with the ForSyDe implementation related to the model illustrated in Figure 3.

Listing 1: ForSyde code for the encoder and decoder system

```
module EncoderDecoder where

import ForSyDe.Shallow

-- example of signals
s_keys_example  = signal [1, 4, 6, 1, 1]
s_input_example = signal [1, 2, 3, 4, 5]

sub :: Num a => a -> a -> a
sub x y = y - x

```

```
add :: Num a => a -> a -> a
add x y = x + y

s_f = signal [(add),(add),(add),(add),(add)]
s_f_inv = signal [(sub),(sub),(sub),(sub),(sub)]

reconfigSY = zipWithSY ($)

genEncoder s_f s_keys = zipWithSY ($) s_f s_keys

genDecoder s_f_inv s_keys = zipWithSY ($) s_f_inv s_keys

ap_encoder s_encF xs = reconfigSY s_encF xs

ap_decoder s_decF xs = reconfigSY s_decF xs

system s_keys s_input = (s_enc, s_output)
       where s_enc = ap_encoder s_encF s_input
             s_output = ap_decoder s_decF s_enc
             s_encF = genEncoder s_f s_keys
             s_decF = genDecoder s_f_inv s_keys

-- to test, just type:
-- system s_keys_example s_input_example
-- result should be:
-- ({2,6,9,5,6},{1,2,3,4,5})
```

The code from Listing 1 brings examples of keys and input signals. One can test the system by typing:

```
*EncoderDecoder> system s_keys_example s_input_example
({2,2,9,5,6},{1,2,3,4,5})
```

The output is a tuple comprised of the encoded signal and the decoded signal computed by the system.

Model Implementation Details

We implemented this encoder/decoder system in a heterogeneous system on chip (SoC) following the design proposed by [Loubach, 2016].

Two different areas comprise the SoC hardware architecture we used. A hard processor area named **programmable device (`ProgDev`)**, and a reconfigurable area named **reconfigurable device (`ReconDev`)**, as illustrated in Figure 4. A partition implementation is named **`prosopon`**.

In general, the implementation of a design is controlled by configuration bits, stored in a configuration memory (CRAM) inside the FPGA.

According to [Loubach, 2016], the `ProgDev` has control over the `ReconDev` to manage which `prosopon` will be programmed/used in a given time. To achieve this, we implemented a software part named **reconfiguration manager (`ReMan`)** in C programming language.

Also, we instantiated a PR control block `prblock`, based on `cyclonev_prblock`, in the `ReconDev`. It is a specific implementation considering the SoC used in this case study.

The main steps of the runtime reconfiguration design are [Loubach, 2016]:

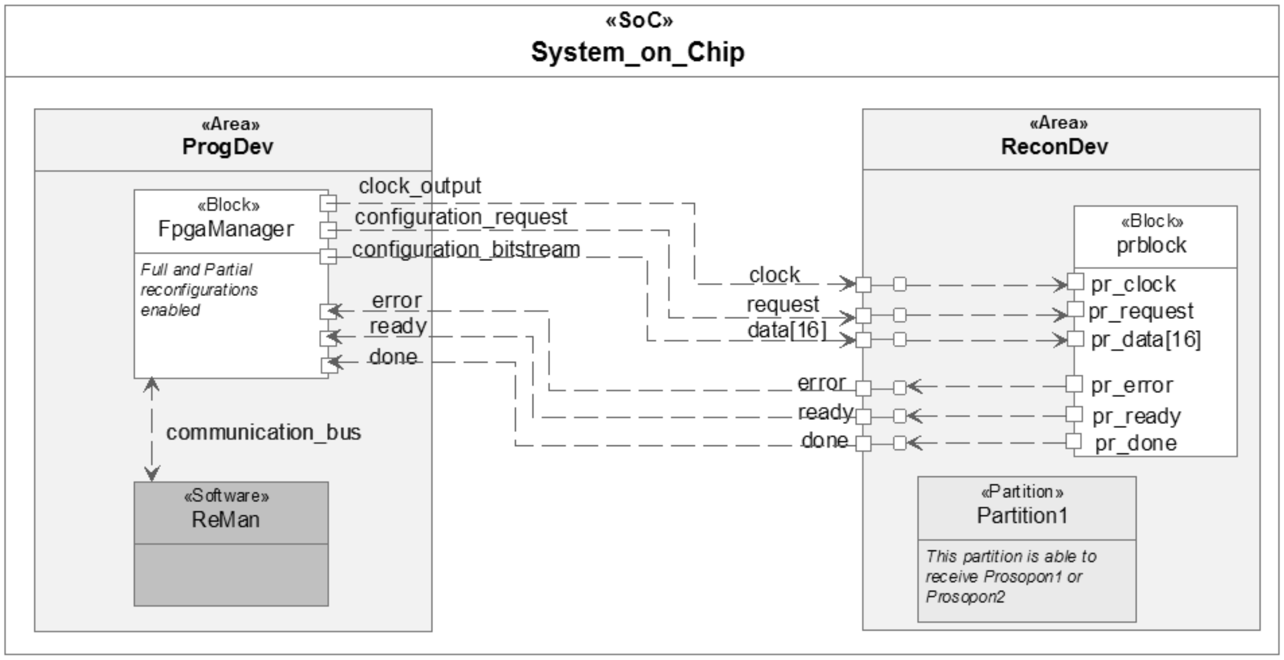

Figure 4: SoC with the `ProgDev` and `ReconDev` areas overview. It also contains the `ReMan` software application inside the `ProgDev` [Loubach, 2016]

1. Implement the `ReconDev`
   (a) Describe the hardware without partitions;
   (b) Identify the parts to be partitioned;
   (c) Define the logical area and physical area for the identified partitions;
   (d) Implement and test each `prosopon`;
2. Generate the bitstream for full reconfiguration (FR) and for each `prosopon`; and
3. Implement the `ReMan` in the `ProgDev`.

Then, we considered the *adaptive processes* implementations (Fig. 3) as one *partition* implementation allowing runtime *partial reconfiguration* (PR) to take place.

Hardware Components Used

The device we used was the Cyclone V SX SoC (part number 5CSXFC6D6F31C6N) within the Cyclone V SoC development kit.

The `ProgDev` from this SoC integrates a dual-core ARM Cortex-A9 MPCore running at 925 MHz, which is the maximum supported frequency. The `ReconDev` has 41,509 adaptive logic modules (ALM)[4].

[4]ALM is considered a basic building block.

Software Ecosystem

To write the hardware description in a hardware description language (HDL), synthesize the hardware, create partitions and generate the configuration bitstreams, for the `ReconDev`, we used the Quartus Prime Version 16.0.0 Build 211 04/27/2016 SJ Standard Edition.

To write the C language code and generate the executable file to embed in the `ProgDev` we used the ARM Development Studio 5 (DS-5) Altera Edition Toolkit version: 5.23.1, build number 5231008, `arm-altera-eabi-` toolchain version 4.9.1.

Implementation Model

Instead of `(+)` and `(-)` functions, we used two well-known symmetric-key cryptographic algorithms: advanced encryption standard (AES) [Daemen and Rijmen, 2002b], and data encryption standard (DES) [Daemen and Rijmen, 2002a] for our implementation model, considering that the encoder/decoder functions can vary.

For AES, we used a 128-bit block size and 128-bit key size implementation based on [Strömbergson, 2014]. On the other hand, DES works with 64-bit block size and 64-bit key size (actually 56-bit represent the key). We used a DES implementation based on [Usselmann, 2001].

Then our signal containing the encoding and decoding functions are defined as follows:

$$s_f = \langle (enc_{AES}), (enc_{DES}), \ldots \rangle$$
$$s_f^{-1} = \langle (dec_{AES}), (dec_{DES}), \ldots \rangle$$

Our implementation model is shown in Figure 5. For the sake of simplicity and due to hardware resources sharing, without the loss of generality, the $genEncoder$ and $genDecoder$ processes were unified into the $genCrypto$ process. Also, the s_f and s_f^{-1} signals turned in $s_{f'}$.

$$s_{f'} = \langle (f_{AES}), (f_{DES}), \ldots \rangle \quad (7)$$

where:
f_{AES} stands for AES cryptography (encoder/decoder) function;
f_{DES} for the DES cryptography (encoder/decoder) function

In the same sense, the s_{encF} and s_{decF} signals became s_{crytoF}, which carries the cryptography (encoder/decoder) functions along with the respective keys. The s_{crytoF} signal is actually the **reconfiguration signal** generated from `ProgDev` to `ReconDev` to change the functionality of $ap_{encoder}$ and $ap_{decoder}$ processes. Together, those processes form a partition identified as `Partition1`, which is subject to runtime partial reconfiguration in hardware.

As mentioned before, a partition implementation is named `prosopon` in our design. As we have identified just one partition and two possible implementations for that partition (*i.e.,* f_{AES} and f_{DES}), then:

$$prosopon_{AES} = f_{AES} \quad (8)$$
$$prosopon_{DES} = f_{DES} \quad (9)$$

By following the methodology steps we:

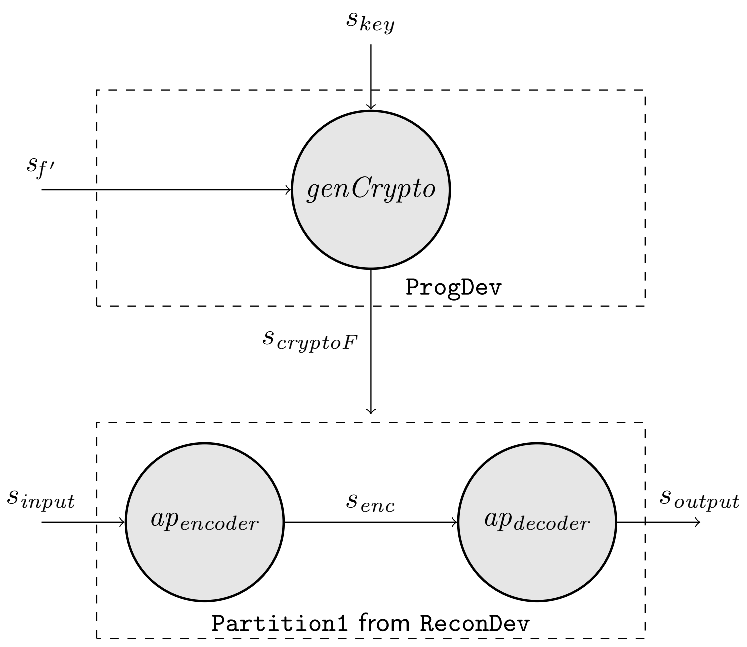

Figure 5: Encoder and decoder implementation model showing both SoC areas: `programmable device (ProgDev)` and `reconfigurable device (ReconDev)` with one partition named `Partition1`

1. Implemented the `ReconDev` by describing the hardware without partitions; then we identified the $ap_{encoder}$ and $ap_{decoder}$ as one partition (Fig. 5); next we defined the logical and physical areas for that partition in the `ReconDev` area; and finally we implemented and tested each `prosopon` (*i.e.,* $prosopon_{AES}$ and $prosopon_{DES}$);

2. Generated the bitstream for full reconfiguration (FR) and for each `prosopon` (*i.e.,* $prosopon_{AES}$ and $prosopon_{DES}$); and

3. Implemented the `ReMan` in the `ProgDev` (*genCrypto*), which actually triggers the functional reconfiguration.

RESULTS

In this section we present the static and runtime measuring of our implementation model (Fig. 5).

Static Data

We generated a bitstream without data compression for FR. This bitstream is required to configure the `ReconDev` for the first time. The number of ALM needed to implement the complete system and the bitstream file size (*i.e.,* raw binary file .rbf) are shown in Table 1.

Table 1: Full reconfiguration static data

FR	Measure	Number
	ALM	2,896
	Bytes	7,007,204

Related to the `Partition1`, we generated two different PR bitstreams modes (*i.e.,* "and/or" and "scrub" [Loubach, 2016]) for $prosopon_{AES}$ and for $prosopon_{DES}$. The required number of hardware resources to implement each `prosopon` is presented on Table 2. Table 3 shows the bitstream sizes.

Table 2: `Prosopons` and the number of ALM needed to implement them

`Prosopon`	**ALM**
$prosopon_{AES}$	2,849
$prosopon_{DES}$	1,019

Table 3: `Prosopons` bitstream modes and sizes. (*)Data given in Bytes

Bitstream mode	$prosopon_{AES}$*	$prosopon_{DES}$*
Scrub	1,873,272	1,873,272
And/or	3,081,512	3,000,628

There is no difference in size between $prosopon_{AES}$ and $prosopon_{DES}$ for scrub mode, considering that in this mode all CRAM bits related to a given partition are overwritten with new data, despite its previous content.

Runtime Data

Once the FR and PR bitstreams were generated, we collected runtime data related to the encoder/decoder functions executions and also the time for reconfiguration taken in the PR and FR.

We applied a 50 MHz base clock (20 nano seconds period) for the encoder/decoder functions in the `ReconDev` area. The execution time measuring are shown in Table 4. `ProgDev` clock frequency was configured to 925 MHz.

Table 4: Functions and sub functions runtime clock cycles and computation time

Main function	**Sub function**	**Clock cycles**	**Time [ns]**
f_{AES}	encode	63	1,260
f_{AES}	decode	63	1,260
f_{DES}	encode	18	360
f_{DES}	decode	19	380

The PR time measuring, that is, the time taken to perform the partial reconfiguration, are given in Table 5. On the other hand, Table 6 shows the time need for one FR.

Table 5: Partial reconfiguration measured times. (*)Time is given in mili seconds [ms]

Bitstream mode	$prosopon_{AES}$*	$prosopon_{DES}$*
Scrub	7.76	7.76
And/or	12.7	12.4

Table 6: Full reconfiguration with no data compression measured time. (*)Time is given in mili seconds [ms]

Bitstream	Time*
FR	29.6

CONCLUSION

We introduce in this paper a runtime adaptivity case study using formal models of computation (MoC) for real-time embedded systems design.

A literature review covering adaptivity concepts, MoC definitions, events, signals, and processes was presented. We also considered the utilization of ForSyDe modeling framework and the synchronous MoC. Within this MoC a process reads its inputs and computes the outputs at each event cycle (discrete time). Besides, this MoC has similarity with clock-driven digital circuits, making it attractive for hardware synthesis.

To show that runtime adaptivity is feasible to be performed, we modeled an encoder/decoder system in a high-level of abstraction (*system specification*) using ForSyDe. Next, we refined that model and manually transformed it into an *implementation specification* to be designed in software (`ProgDev` using C programming language) and hardware (`ReconDev` using hardware description language) parts.

The implementation specification used a heterogeneous system on chip (SoC) as hardware platform.

Results showed that one partial reconfiguration ($prosopon_{AES}$ in scrub mode = 7.76 ms) takes less time to complete than a full reconfiguration (29.6 ms). Therefore, this enables the possibility to runtime reconfiguration using partial reconfiguration techniques, then leading to runtime adaptivity feasibility.

Our paper has impact on both industry, by enabling future efficient adaptive avionics, and academia, by addressing a design methodology with high-level abstraction for runtime reconfigurability.

As future work, our design could take into account the use of systematic and automated transformations instead of manually/*ad-hoc* transformations among the different abstraction levels and still preserving semantics.

ACKNOWLEDGMENT

This research work is partially supported by the Swedish-Brazilian Research and Innovation Centre (CISB) – Project CISB ID 68-2015-A, Regular Research Awards grant #2014/24855-8 from São Paulo Research Foundation (FAPESP), and the Altera University Program.

REFERENCES

[Afonso et al., 2013] Afonso, G., Baklouti, Z., Duvivier, D., ben Atitallah, R., Billauer, E., and Stilkerich, S. (2013). Heterogeneous CPU/FPGA Reconfigurable Computing System for Avionic Test Application. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 260–267.

[Andrillon and Aviation, 2013] Andrillon, B. and Aviation, D. (2013). Contribution of integrated modular avionics of second generation for business aviation.

[Bieber et al., 2012] Bieber, P., Boniol, F., Boyer, M., Noulard, E., and Pagetti, C. (2012). New challenges for future avionic architectures. *Journal AerospaceLab*, 4.

[Bobda, 2007] Bobda, C. (2007). *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. Springer.

[Compton and Hauck, 2002] Compton, K. and Hauck, S. (2002). Reconfigurable computing: A survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210.

[Daemen and Rijmen, 2002a] Daemen, J. and Rijmen, V. (2002a). *The Data Encryption Standard*, pages 81–87. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Daemen and Rijmen, 2002b] Daemen, J. and Rijmen, V. (2002b). *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Berlin Heidelberg.

[Jantsch, 2003] Jantsch, A. (2003). *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation*. Morgan Kaufmann, 1st edition.

[Jantsch, 2005] Jantsch, A. (2005). *Embedded Systems Handbook*, chapter Models of Embedded Computation. CRC Press.

[Koch, 2013] Koch, D. (2013). *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications*. Springer-Verlag New York, 1 edition.

[Loubach, 2016] Loubach, D. S. (2016). A runtime reconfiguration design targeting avionics systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, Sacramento, USA.

[Salloum et al., 2012] Salloum, C., Elshuber, M., Hoftberger, O., Isakovic, H., and Wasicek, A. (2012). The ACROSS MPSoC – a new generation of multi-core processors designed for safety-critical embedded systems. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 105–113.

[Sander and Jantsch, 2004] Sander, I. and Jantsch, A. (2004). 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.

[Sander and Jantsch, 2008] Sander, I. and Jantsch, A. (2008). Modelling Adaptive Systems in ForSyDe. *Electronic Notes in Theoretical Computer Science*, 200(2):39 – 54. Proceedings of the First Workshop on Verification of Adaptive Systems (VerAS 2007).

[Sterpone and Ullah, 2013] Sterpone, L. and Ullah, A. (2013). On the optimal reconfiguration times for TMR circuits on SRAM based FPGAs. In *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, pages 9–14.

[Strömbergson, 2014] Strömbergson, J. (2014). https://github.com/secworks/aes.

[Usselmann, 2001] Usselmann, R. (2001). https://github.com/freecores/des.

[Voros et al., 2013] Voros, N. S., Hübner, M., Becker, J., Kühnle, M., Thomaitiv, F., Grasset, A., Brelet, P., Bonnot, P., Campi, F., Schüler, E., Sahlbach, H., Whitty, S., Ernst, R., Billich, E., Tischendorf, C., Heinkel, U., Ieromnimon, F., Kritharidis, D., Schneider, A., Knaeblein, J., and Putzke-Röming, W. (2013). MORPHEUS: A heterogeneous dynamically reconfigurable platform for designing highly complex embedded systems. *ACM Trans. Embed. Comput. Syst.*, 12(3):70:1–70:33.

[Wolfig and Jakovljevic, 2008] Wolfig, R. and Jakovljevic, M. (2008). Distributed IMA and DO-297: Architectural, communication and certification attributes. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1.E.4–1–1.E.4–10.

[Wong et al., 2011] Wong, S., Brandon, A., Anjam, F., Seedorf, R., Giorgi, R., Yu, Z., Puzovic, N., Mckee, S., Sjalander, M., Carro, L., and Keramidas, G. (2011). Early results from ERA - embedded reconfigurable architectures. In *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pages 816–822.