33RD CONGRESS
OF THE INTERNATIONAL COUNCIL
OF THE AERONAUTICAL SCIENCES
STOCKHOLM, SWEDEN, 4-9 SEPTEMBER, 2022

ICAS 2022 SWEDEN

TOWARDS CORRECT-BY-CONSTRUCTION DESIGN OF SAFETY-CRITICAL EMBEDDED AVIONICS SYSTEMS

Ingo Sander[1], Ingemar Söderquist[2], Mats Ekman[2], Rodolfo Jordao[1], Fahimeh Bahrami[1], Rui Chen[1] & Anders Åhlander[3]

[1]KTH Royal Institute of Technology, Kistagången 16, 164 40 Kista, Sweden
[2]Saab AB, Bröderna Ugglas Gata, 581 88 Linköping, Sweden
[3]Saab AB, Solhusgatan 10, 412 89 Göteborg, Sweden

Abstract

New methodologies are needed for the development of avionics systems to meet today's software explosion in complexity and related cost due to the increased functionality in the aircraft. Current design flows for software-intensive systems do not have a clear path from the functional specification to the final implementation and cannot provide real-time guarantees. The situation will become even more difficult because, in the future, more and more applications will share the same computation nodes and the network in a distributed hierarchical network-based system. In order to overcome the present situation, a novel methodology for a correct-by-construction design of safety-critical embedded avionics systems has been created and formulated within the Vinnova NFFP7 project CORRECT. Correct-by-construction design is a radical departure from current design practice, with the potential to decrease the verification costs for future systems significantly. The paper presents the underlying foundation of the methodology, its carefully selected ingredients, and discuss available results and existing tool support. The methodology is based on a disciplined system modelling environment grounded on a sound formal foundation, a design space exploration technique, and a clear path to hardware and software synthesis. An industrial case study investigates the potential of the methodology.

Keywords: Correct-by-Construction Design, Integrated Modular Avionics, System Modelling, Design Space Exploration, System Synthesis

1. Introduction

Today it is broadly recognised that future aircraft systems will have an increased level of automation, including autonomous operation and management of air vehicles (manned or unmanned). Also, the size and complexity of embedded software will increase drastically. The degree of automation in the aircraft will directly depend on the computing power in the avionics system and the efficiency of available design processes and tools used to map the overall functionality onto the platform architecture. With its electronics and software, the avionics system is a prerequisite for an aircraft's general functions, e.g. cabin entertainment systems and tactical functions. There is a continuous growth in the number of functions and their complexity in an aircraft. The dominant architecture is Integrated Modular Avionics (IMA) [1], where functions with different levels of safety, according to ARP4754 [2] and ARP4761 [3], can utilise a single processor (node), but are separated by the support of the electronics architecture and software layers, such as ARINC653 [4]. Several nodes connected through a network build up an IMA system [5]. The European aeronautics industry shares the roadmap for IMA and the need for common architectures, modules, and design processes. They are well described based on first-generation IMA in [5, 6, 7, 8]. The second generation, IMA2G, is the outcome of the project SCARLETT [9]. It has been further developed to IMA2.5G as an outcome of the project ASHLEY [10]. One important conclusion is the foreseen complexity explosion and the identified need for new methods and tools to handle complexity. The design space, i.e. the possible combinations of platform

architecture, components and possible mappings of the aircraft functionality, increases in complexity. Another critical driver for the need of new design methods is to reduce the development costs for future products, which has to be addressed by the software and system design methodology. The use of more efficient tools can achieve this in the same way as the electronic system industry successfully has dealt with complexity increase, see the IEEE International Roadmap for Devices and Systems (IRDS) [11].
Natale and Sangivanni-Vincentelli [12] view the move from federated to integrated architectures, including IMA, as a paradigm shift, which requires the development of new methods, models and tools. Sifakis [13] analyses how the success of electronic design automation (EDA) can be adapted to general mixed hardware/software systems through the application of four principles, which have enabled the success of EDA: (1) separation of concerns, (2) component-based design, (3) semantic coherency, and (4) correct-by-construction. He states that “formal verification can be applied to systems whose criticality justifies relatively high development costs” and advocates a shift towards rigorous system design with an emphasis on correct-by-construction. The platform-based design (PBD) paradigm [14] addresses system design complexity by a meet-in-the-middle approach, which is a combination of a top-down and a bottom-up effort. The platform is composed of a set of library components with associated performance abstraction (bottom-up), while the functionality is mapped on an instance of the platform with constraint propagation (top-down).

2. Correct-by-Construction Design Methodology

The correct-by-construction approach presented in this paper builds on models at several levels, fulfils the above principals of Sifakis [13], and is consistent with platform-based design [14]. Furthermore, efforts have been made to be in line with aeronautic industry practice regarding development steps and terminology for integrated modular avionics as specified in RTCA DO-297 [5].

2.1 IMA-based Design Methodology

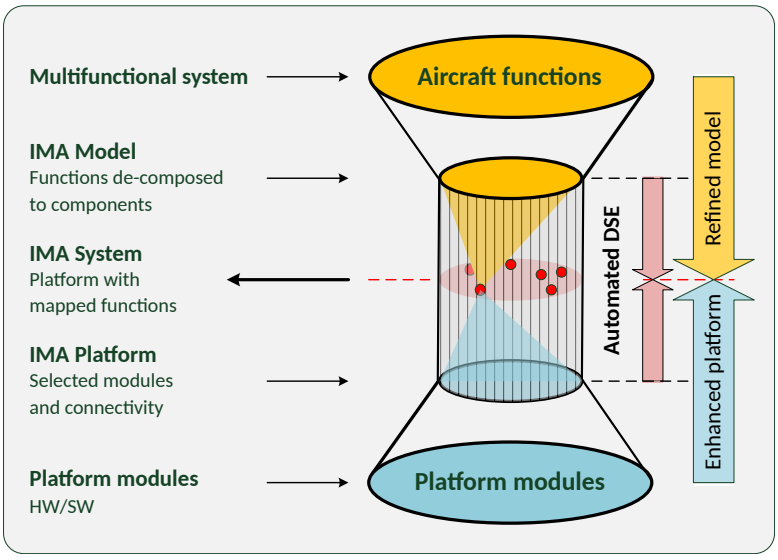

Figure 1 – Hour glass illustrating the IMA-based design methodology and abstraction levels proposed for mapping multiple aircraft functions on one IMA platform consisting of platform modules.

The paper proposes a conceptual IMA-based design methodology, which is illustrated in Figure 1. The approach uses several levels of abstraction, which cleanly separate the function models from the platform modules, and supports top-down and bottom-up design. The IMA system is established by a mapping of function models to platform modules. This separation of concerns enables independent function and platform design and enables many possible mapping combinations for the final IMA system. The resulting design space should ideally be explored by an automated tool to create a correct and efficient IMA system.

The following shortened definitions in RTCA DO-297 [5] are considered the most essential supporting IMA design.

- **Aircraft Function**: The capability of the aircraft that may be provided by the hardware and software of the systems on the aircraft. Functions include flight control, autopilot, braking, fuel management, etc.
- **Application**: Software and/or application-specific hardware with a defined set of interfaces that, when integrated with a platform, performs a function.
- **Component:** A self-contained hardware part, software part, database, or combination thereof that is configuration controlled. A component does not provide an aircraft function by itself.
- **IMA System:** Consists of an IMA platform(s) and a defined set of hosted applications.
- **Platform:** Module or group of modules, including core software, that manages resources in a manner sufficient to support at least one application. IMA hardware resources and core software are designed and managed in a way to provide computational, communication, and interface capabilities. Platforms, by themselves, do not provide any aircraft functionality.
- **Module:** A component or collection of components. A module may also comprise other modules. A module may be software, hardware, or a combination of hardware and software, which provides resources to the IMA-hosted applications.

The IMA methodology addresses concurrent multifunctional system design starting at the aircraft level. Each aircraft function and the corresponding application are well-defined with functional and non-functional requirements and overall design constraints, e.g. safety (separation), energy usage, and peak power. Then, the functions are decomposed into components and defined as an IMA model. The approach aligns perfectly with the platform-based design paradigm [14]. A design space exploration tool shall identify a feasible mapping of the IMA model components to the IMA platform that satisfies the design constraints. The set of feasible solutions are indicated in Figure 1 by the red dots.

2.2 Correct-by-Construction Design Flow

The correct-by-construction methodology presented in this paper uses the ForSyDe (Formal System Design) [15] framework as base for the conceptual design flow of Figure 1. Figure 2 shows a more detailed view of the correct-by-construction design flow, where the application model corresponds to the decomposed functions in the IMA model of Figure 1. The design process starts with capturing top-level requirements. Then follows the system modelling phase, where the functionality of each aircraft function is modelled as application model (Section 3) using the ForSyDe system modelling framework [15]. ForSyDe application models are executable and have a sound formal foundation based on the theory of models of computation [16]. ForSyDe provides powerful modelling elements to yield a structured and well-defined application model, which can be simulated to validate the functionality. The nature of this well-defined system model is a key property of the correct-by-construction design process and is exploited in the following phases of the design flow. Each application must fulfil individual top-level requirements and additional design constraints, such as timing constraints. In addition, the complete system consisting of several applications has to fulfil global design constraints, e.g. energy, reliability, security, cost or safety constraints. The platform is modelled as hierarchical architecture consisting of predictable computation and communication components with well-defined deterministic behaviour and quality of service (Section 4). An essential step in the correct-by-construction design process is the design space exploration (Section 6), which aims at identifying an efficient implementation for the set of applications on the shared target platform, so that each application satisfies its individual design constraints and also the global design constraints are satisfied. The design space exploration tool exploits the well-defined application models and platform models to efficiently search the design space by mapping and scheduling the application models to platform components. Solutions will give the mapping and schedules and contain performance numbers such as utilisation,

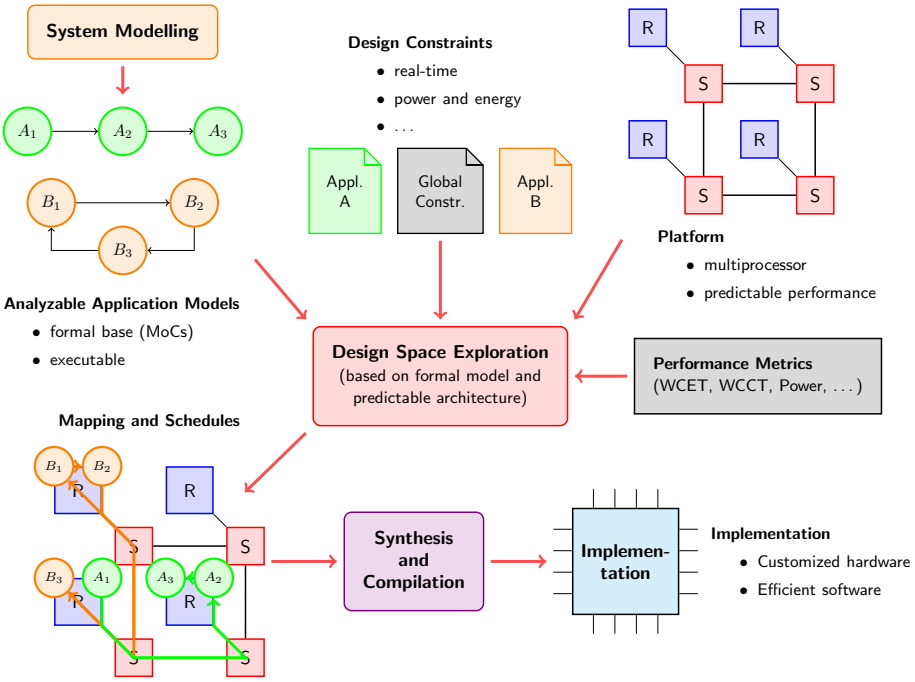

Figure 2 – Correct-by-construction design flow.

throughput, or design costs. The outcome of the design space exploration forms the entry point to the final part of the design flow, the synthesis and compilation phase (Section 5), where the final implementation is generated. The process for code generation exploits the well-defined and structured nature of the application models. It is important to point out that the methodology assumes that the application models have been validated as correct with respect to the functional requirements. The correct-by-construction methodology aims to generate a correct implementation of the application models on a shared target platform with respect to the modelled functionality of the application model, the non-functional design constraints for each application, and the overall design constraints imposed on the system.

3. Application Model

Defining a suitable application model is key for a successful design since it is the starting point for all further design steps. Thus, the application model should both capture the system's functionality and support the succeeding activities of the design process. A good application model

- defines the functionality the application has to perform,
- enables the simulation or formal verification of the functionality of the system so that designers can confirm that the intended functionality is correctly modelled,
- uses a formal language with standardised semantics so that the model has one clear meaning,
- abstracts from low-level details so that the final implementation can use different implementation techniques and at the same time allowing the designer to focus on the functional aspects during modelling, and
- enables at the same time a clear path to the final implementation.

Errors at an early stage of the design process can be avoided if the system model can be simulated or formally verified, preventing high unnecessary design costs. Another important aspect is that the modelling language should support the development of the tools required at different stages of the design process, such as tools for simulation, performance analysis and design space exploration, compilation and synthesis, or formal verification. The simpler the modelling language is and the

cleaner the formal semantics are defined, the easier it is to develop powerful tools. Thus, an expressive modelling language, with many language components and complex formal semantics, allows expressing many different models but makes it very difficult to develop the necessary tools for the design process. Hence, it is essential to find a suitable trade-off between the expressiveness and the simplicity of the modelling language.

3.1 Formal Base of ForSyDe Models

The ForSyDe modelling framework is based on the *tagged-signal model* [16] and the *functional programming paradigm*. The tagged-signal model enables the formal description of different models of computation (MoCs), which are relevant for system design and for which powerful analysis methods exist. The functional programming paradigm ensures that application models have clean semantics and enables simulation and powerful modelling constructs, which the following design activities can exploit. This underlying foundation is powerful enough to express different MoCs, such as the synchronous MoC, data-flow MoCs, and continuous-time MoCs. ForSyDe offers modelling libraries for different models of computation, which are publicly available on the ForSyDe web page [17].

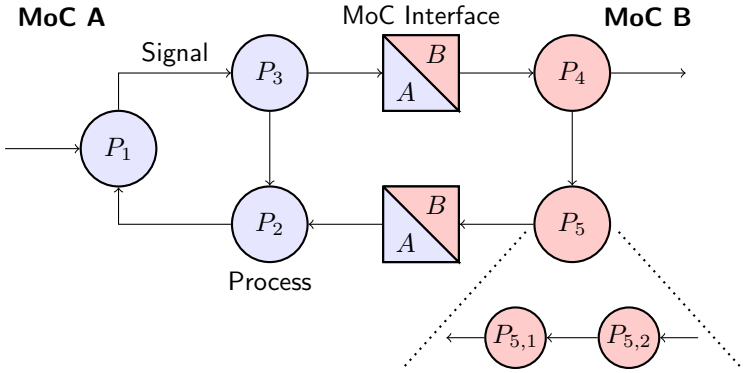

Figure 3 – A ForSyDe application model is a hierarchical concurrent process network, where processes belonging to different models of computation (MoCs) communicate via MoC interfaces.

Applications are modelled in ForSyDe as heterogeneous hierarchical concurrent process networks, where *processes* communicate with each other via *signals* as illustrated in Figure 3. Following the tagged-signal model [16], a *signal* s is defined as a set of *events* $s = \{e_0, e_1, e_2, \ldots\}$, where each event e_i has a tag $t_i \in \mathbf{T}$ and a value $v_i \in \mathbf{V}$, thus $e_i = (t_i, v_i)$. The tag system is used to relate events. Depending on the set of tags $\mathbf{T}$, tags can be used to express (a) physical time, where $\mathbf{T} = \mathbb{R}$; (b) discrete time, where $\mathbf{T}$ is a totally ordered discrete set; or (c) precedences, where $\mathbf{T}$ is a partially ordered discrete set. A process P is a function of input signals to output signals, i.e. $P = (s_{i,1}, \ldots, s_{i,m}) \rightarrow (s_{o,1}, \ldots, s_{o,n})$. Processes can be composed of other processes to create hierarchical process networks. The process P_5 in Figure 3 is composed by means of the processes $P_{5,1}$ and $P_{5,2}$. Processes belonging to different models of computation cannot directly communicate with each other, instead special processes called *MoC interfaces* are used to give a well-defined semantics for the conversion of signals between two models of computation as illustrated in Figure 3.

3.2 The ForSyDe Modelling Language

The underlying formal foundation of ForSyDe enables application modelling with different programming languages. This paper introduces the ForSyDe modelling concepts using FORSYDE-SHALLOW, which is implemented in the functional programming language Haskell. A tutorial example in the form of a counter in the synchronous MoC is used to give a good understanding of the key concepts of ForSyDe.

3.2.1 Signals

ForSyDe defines signals as a *sequence* of *events*. In the synchronous MoC, the *tag* is given implicitly by the position in the signal, and the value of the event is directly given by the data value at that position. A signal s can be created by converting a list to a signal.

```
s = signal [1,2,3]
```

This defines a signal $s = \{e_0, e_1, e_2\}$ with the implicit tags t_0, t_1, t_2 and the values $v_0 = 1$, $v_1 = 2$, $v_3 = 3$. The tags are ordered within each signal, but the interpretation of the tags is defined by the specific MoC. This also means that it is possible that signals are not comparable wit respect to there tags, and there is no defined tag order relation between events of two different signals.

3.2.2 Processes

A process is a function of input signals to output signals. Processes in ForSyDe are created by *process constructors*. A process constructor is in general a higher-order function that takes arguments in the form of functions and values and creates a process, but there are also simpler process constructors, which only take values as arguments and create a process. A similar set of process constructors exist in each MoC. Process constructors can be classified as

- *combinational process constructors*, which create processes that have no internal state, and
- *sequential process constructors*, which create processes that have an internal state.

The basic combinational process constructor for the synchronous MoC is `combSY`, which takes a function `f` as an argument and applies it to all values in a signal. Thus, using the previously defined signal `s`, the function `combSY (+1)` creates a process, which increments all values in the signal `s`.

```
console> combSY (+1) s
{2,3,4}
```

In the same way, combinational processes with more than one input signal can be created. The process constructor `comb2SY` is used to create processes with two input signals and can be used to define an adder.

```
adder = comb2SY (+)
```

The basic sequential process constructor in the synchronous MoC is `delaySY`, which delays an input value one event cycle by returning an initial value at tag t_0.

```
console> delaySY 0 s
{0,1,2,3}
```

3.2.3 Process Composition

Processes can be composed by formulating a process netlist as a set of equations. Figure 4 shows the process network of a counter that shall be modelled in `ForSyDe-Shallow` using the synchronous MoC.

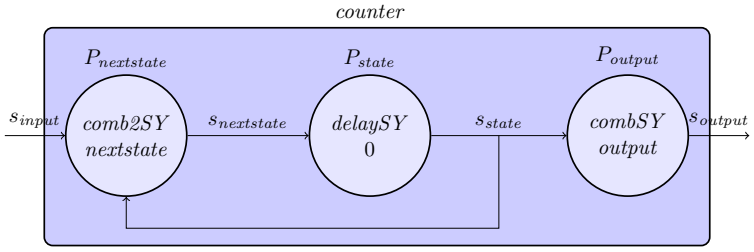

Figure 4 – Counter modelled in the synchronous MoC in FORSYDE-SHALLOW.

The counter has an input signal, which specifies if the counter should count upwards (`UP`) or should keep its present value (`HOLD`). Whenever the counter reaches the value 4, the counter shall continue with the value 0 at the next occurrence of `UP`. The counter also has an output, which should return the value `TICK` when it is in state `0`; otherwise, it should return the value `NO_TICK`.
Listing 1 gives the complete ForSyDe-Shallow code of the counter. It also illustrates the different steps as part of a top-down process, which the designer has to execute to create the model.

```
1  module Counter where
2
3  import ForSyDe.Shallow
4
5  data Direction = UP
6                 | HOLD deriving (Show)
7
8  data Clock = TICK
9             | NO_TICK deriving (Show)
10
11 -- Step 1: Specification of process network
12 counter s_input = s_output
13    where s_output = p_output s_state
14          s_state = p_state s_nextstate
15          s_nextstate = p_nextstate s_state s_input
16
17 -- Step 2: Selection of process constructors
18 p_nextstate = comb2SY nextstate
19 p_state = delaySY 0
20 p_output = combSY output
21
22 -- Step 3: Specification of leaf functions
23 nextstate state HOLD = state
24 nextstate 4     UP   = 0
25 nextstate state UP   = state + 1
26
27 output 0 = TICK
28 output _ = NO_TICK
```

Listing 1: ForSyDe model of the counter of Figure 4.

1. The designer sketches the process network, including selecting the model of computation for each process and specifying the communication between the processes through signals (line 11 to 15).

2. The designer selects suitable process constructors for all processes in the process network or expresses a high-level process by a composition of other processes. In Figure 4, line 17 to 20, the process constructors `comb2SY` and `combSY` are used to form combinational processes, while the process constructor `delaySY` is used to model a sequential process with internal state.

3. The designer formulates the arguments of the process constructors to form processes, i.e. the leaf functions `nextstate` (line 23 to 25), `output` (line 27 to 28) and other parameters, like the initial state of the process created by `delaySY` is set to 0 (line 19).

Haskell is a strongly typed language and although no data types are given, Haskell can infer the data type of the process `counter`.

```
console> :t counter
counter :: Signal Direction -> Signal Clock
```

This means that counter is a function that takes an input signal with event values of type `Direction` and produces a signal with event values of type `Clock`. The model can be simulated in FORSYDE-SHALLOW.

```
console>  counter (signal ([HOLD,UP,HOLD,UP,UP,UP,UP,HOLD,UP]))
{TICK,TICK,NO_TICK,NO_TICK,NO_TICK,NO_TICK,NO_TICK,TICK,TICK,NO_TICK}
```

More powerful process constructors can be created from the basic combinational and sequential process constructors. The process constructor `mooreSY` models a Moore finite state machine and takes three arguments, (1) a function to calculate the next state, (2) a function to calculate the output, and (3) a value to define the initial state. Using `mooreSY` the counter can be modelled in a more compact way, where the functions `nextstate` and `output` are the same functions as in Listing 1, and the value 0 denotes the initial state.

```
console> counter' =  mooreSY nextstate output 0
console> counter' (signal ([HOLD,UP,HOLD,UP,UP,UP,UP,HOLD,UP]))
{TICK,TICK,NO_TICK,NO_TICK,NO_TICK,NO_TICK,NO_TICK,TICK,TICK,NO_TICK}
```

3.3 Benefits of ForSyDe Modelling Framework

The process constructor to model a process is a key concept in ForSyDe. A process constructor separates computation, expressed by the function argument, from communication and synchronisation, expressed by the process constructor. The ForSyDe methodology prescribes that *all processes have to be modelled with process constructors* and *processes can only communicate via signals*. This rule enforces a structured and well-defined model with few but powerful building blocks in the form of process constructors. Due to its underlying formal foundation, ForSyDe models are highly analysable, which can be exploited in several ways; an example is design space exploration which is discussed in Section 6. In addition, the process constructor concept also enables a clear path to an implementation, which is elaborated in Section 5.

4. Platform Model

The proposed methodology conceptually addresses the entire avionics platforms of today and foreseen in near future. The first IMA based avionics platform was implemented in one single physical cabinet; today the avionics platform is distributed over the entire aircraft. The processing network connectivity is normally hierarchical with redundant communication paths and interfaces to achieve fail-operational behaviour and resilience. The avionics platform, see definition in Section 2, consists of heterogeneous and multiple instances of modules providing resources for data processing, storage, communication, and input/output interface.

The module library, proposed for the IMA-based design flow in Section 2.1, contains models for all types of platform components, including the resource characteristics of computation components, the bandwidth and latency of communication components, and the I/O performance of interfaces. This includes all equations needed to calculate workload characteristics and resource utilisation based on application needs and other properties needed for design space exploration. The IMA platform is defined by selected modules and their connection; an example is shown as part of the case study in Section 7. The components in the platform modules need to provide a guaranteed quality of service to ensure that the final system can meet its design constraints. A suitable platform model is a prerequisite for an accurate and efficient design space exploration, which requires a suitable abstraction and the characterisation of the essential parameters of each platform module as well as the entire network. The current ForSyDe platform model, which is also used in the case study, uses only two component types: computation and communication components. There are so far no

specific storage or interface components. Instead, memory properties are part of the specification of computation and communication components. Thus, a target platform consists of modules which are created out of computation components connected by communication components. From the modelling point of view, there is no principle distinction between on-chip or off-chip communication components, enabling a flexible design space exploration using the same techniques for on-chip and distributed systems.
Since the methodology aims for correct-by-construction design for safety-critical systems, where several applications use a shared platform, the target platforms should support *predictability* and also *composability* [18]. Predictability ensures that an application receives a minimum guaranteed performance, while composability confirms that the behaviour of one application will not be affected by another application. While most industrial processor platforms aim for high average-case performance, several research projects address predictability and composability as fundamental properties [19, 20] in a similar way as in integrated modular avionics [8].

5. Synthesis and Code Generation from ForSyDe Models

This section gives the general principles on how ForSyDe application models can be synthesised into an implementation for a given target architecture, so that the semantics of the original ForSyDe model are preserved in the final implementation. The synthesis method described in this paper is possible due to the sound underlying theoretical foundation through well-defined models of computation and the clean structure and semantics of ForSyDe models, as discussed in Section 3. The following concepts build on and extend earlier ForSyDe-papers on hardware [21] and software [22] synthesis and code generation.

5.1 General Principles for Synthesis

An implementation of a ForSyDe model requires the realisition of the ForSyDe model components and their interconnection in the target language or architecture. Both the functional and the timely behaviour of the implementation need to satisfy the semantics of the ForSyDe model. A correct implementation of a ForSyDe model in a particular target technology, which can be a target language or platform consisting of hardware and software, requires a

1. a correct implementation of all process constructors in the target technology,
2. a correct translation of all function arguments and parameters, and
3. a correct implementation of the process network.

in the target technology.

5.2 Hardware Synthesis

To generate a hardware implementation, ForSyDe models have to be converted into synthesisable hardware models, expressed in hardware description languages like VHDL or Verilog. The ForSyDe tool FORSYDE-DEEP [17] can synthesise ForSyDe models belonging to the synchronous MoC to synthesisable hardware in VHDL.
The synchronous MoC of ForSyDe provides several process constructors. However, the synthesis of the process constructor `combSY` as the basic combinational process constructor and `delaySY` as the basic sequential process constructor illustrates the basic principles of the translation of ForSyDe processes to VHDL. More complex process constructors are created through the basic process constructors and an accompanying VHDL template. Figure 5 illustrates how processes based on the basic process constructors are implemented in hardware. A process `combSY f` is implemented as a combinational circuit, where the function `f` is translated to the corresponding VHDL function $\texttt{f}_{\texttt{HW}}$. The process `delaySY v` is implemented as register in hardware, where the initial state is given by the value `v`. It also includes the more complex process constructor *mooreSY* that is used to model Moore finite state machines, for which an efficient hardware implementation exists.
Figure 6 illustrates the synthesis of ForSyDe process networks to hardware. ForSyDe processes are converted into VHDL using the methods above and are instantiated as VHDL components. ForSyDe

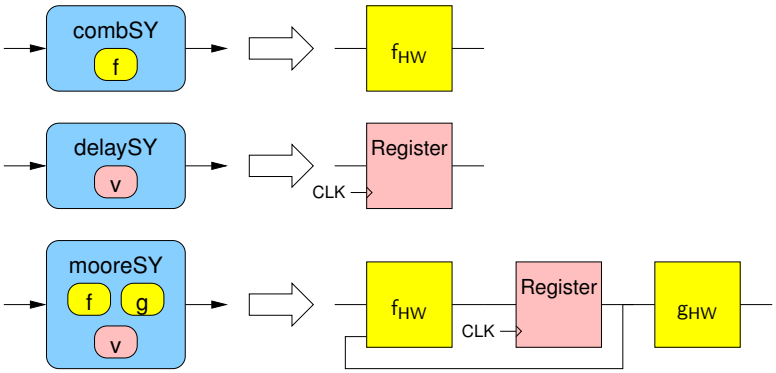

Figure 5 – Templates for hardware synthesis from ForSyDe processes.

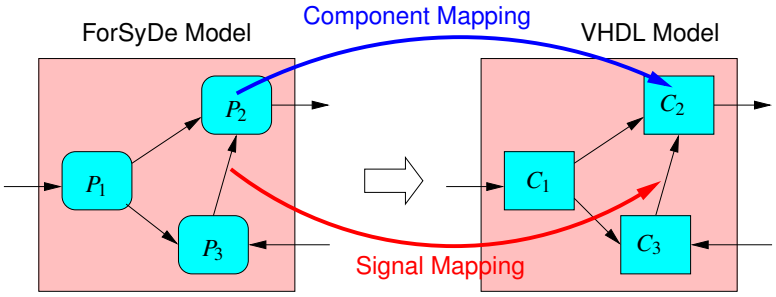

Figure 6 – ForSyDe process networks are synthesised to hardware by instantiating synthesised ForSyDe processes to VHDL components and ForSyDe signals to VHDL signals.

signals are converted to VHDL signals, and then the structure of the ForSyDe process network - expressed as a netlist in the form of a set of equations - is converted into the corresponding VHDL netlist. The principles have been demonstrated in the hardware synthesis tool FORSYDE-DEEP, which is publicly available on [17].

5.3 Software Synthesis

The synthesis to software follows the general principles and rules for the synthesis of ForSyDe models as described in Section 5.1. This section will focus on the synthesis from synchronous data flow (SDF) ForSyDe models to software in the form of C-code. The discussion will be conducted in a tutorial style and uses the ForSyDe SDF model of Listing 2. The functions in the model are simplified to be able to focus on the principles of the synthesis process, but the model includes all important ingredients of an SDF model, such as different production and consumption rates and a feedback loop with initial tokens.

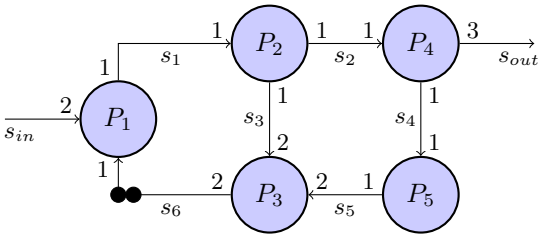

Figure 7 – SDF graph representation of the ForSyDe model in Listing 2

The ForSyDe model of Listing 2 can be directly translated into the SDF model in Figure 7, and further analysed using the methods for SDF [23]. The system netlist (lines 5-12) of the ForSyDe model are translated into the five processes P_1 to P_5, the signals s_1 to s_6, which form inputs, outputs and internal signals, and two initial tokens with the value 0 originating from the arguments `[0,0]` of the process constructor `delaySDF`. An SDF process, also called *actor*, consumes and produces during each

```
1  module SDF_System_Model where
2
3  import ForSyDe.Shallow
4
5  -- System Netlist
6  system s_in = s_out where
7    s_1 = p_1 s_in s_6_delayed
8    (s_2, s_3) = p_2 s_1
9    s_6 = p_3 s_3 s_5
10   (s_out, s_4) = p_4 s_2
11   s_5 = p_5 s_4
12   s_6_delayed = delaySDF [0,0] s_6
13
14 -- Process Specification
15 p_1 = actor21SDF (2,1) 1 f_1
16   where f_1 [x1,x2] [y] = [x1+x2+y]
17 p_2 = actor12SDF 1 (1,1) f_2
18   where f_2 [x] = ([x],[x+1])
19 p_3 = actor21SDF (2,2) 2 f_3
20   where f_3 [x1,x2] [y1,y2]
21       = [x1+x2,y1+y2]
22 p_4 = actor12SDF 1 (3,1) f_4
23   where f_4 [x] = ([x,x+1,x+2],[x])
24 p_5 = actor11SDF 1 1 f_5
25   where f_5 [x] = [x+1]
```

A simulation of the model in Haskell gives the following output.

```
1  *SDF_System_Model> let s = signal [1..10]
2  *SDF_System_Model> system s
3  {3,4,5,7,8,9,23,24,25,27,28,29,71,72,73}
```

Listing 2: Executable ForSyDe SDF model serving as running example for software synthesis.

execution, also called *firing*, a fixed number of tokens on each incoming and outgoing signal. The numbers of consumed and produced tokens are given as argument to the process constructors, e.g. the term `p_1 = actor21SDF (2,1) 1 f_1` in line 15 defines that process P_1 is an SDF actor with two input and one output signals. P_1 will during execution consume two tokens on the first and one token on the second input signal and execute the function `f_1` given on line 16.
Following the principles and rules for synthesis from ForSyDe models, a synthesis of an SDF-model to a bare-metal single-processor implementation in C requires that

1. the process network is translated into a static schedule with sufficient buffer space using the methods for SDF discussed in [23],
2. each process constructor is converted into a corresponding C-code template, and
3. each function is translated into the corresponding C-code.

Using the methods of [23] a feasible schedule $P_1P_2P_4P_5P_1P_2P_4P_5P_3$, and the required minimum buffer size s_i for each FIFO, $s_1 = 1, s_2 = 1, s_3 = 2, s_4 = 1, s_5 = 2, s_6 = 2$, can be derived. A data structure for each FIFO with read and write access functions has to be implemented. For a single-processor bare-metal implementation, a suitable data structure is a circular buffer with the following access functions.

1. `channel createFIFO(token* buffer, size_t size)` creates a FIFO-buffer in memory,

2. `void readToken(channel ch, token* x)` reads a token `x` of type `token` from channel `ch`, and
3. `void writeToken(channel ch, token x)` writes a token `x` of type `token` to channel `ch`.

At the start of the C-program the FIFOs are created and the two initial tokens are put into channel s_6.

```
/* Create FIFO-Buffers for signals */
token* buffer_s_in  = malloc(2 * sizeof(token));
channel s_in = createFIFO(buffer_s_in, 2);
token* buffer_s_1  = malloc(1 * sizeof(token));
channel s_1 = createFIFO(buffer_s_1, 1);
...
token* buffer_s_out  = malloc(3 * sizeof(token));
channel s_out = createFIFO(buffer_s_out, 3);

/* Put initial tokens in channel s_6 */
writeToken(s_6, 0);
writeToken(s_6, 0);
```

As next step, the netlist needs to be created by using the schedule derived earlier. The schedule is implemented as an infinite while-loop, where each actor is implemented using a powerful function template `actorXYSDF(..)`, where `X` corresponds to the number of inputs of the actor and `Y` to the number of outputs.

```
while(1) {
   for(i = 0; i < 2; i++) {
     /* Read input tokens */
     ...
     /* P_1 */
     actor21SDF(2, 1, 1, &s_in, &s_6, &s_1, f_1);
     /* P_2 */
     actor12SDF(1, 1, 1, &s_1, &s_2, &s_3, f_2);
     /* P_4 */
     actor12SDF(1, 3, 1, &s_2, &s_out, &s_4, f_4);
     /* Write output tokens */
     ...
     /* P_5 */
     actor11SDF(1, 1, &s_4, &s_5, f_5);
   }
   /* P_3 */
   actor21SDF(2, 2, 2, &s_3, &s_5, &s_6, f_3);
}
```

For each process constructor `actorXYSDF` a corresponding C-function template is available. An actor process constructor in ForSyDe is a higher-order function that takes several arguments. One of these arguments is the function that is executed during an actor firing. The `actorXYSDF` template is implemented as higher-order function using a function pointer, which enables a direct conversion of the ForSyDe actor process into an implementation in C. The actor process constructor `actor12SDF` is used as example to explain how these process constructors are implemented in C.

```
void actor12SDF(int consum, int prod1, int prod2,
                channel* ch_in, channel* ch_out1, channel* ch_out2,
                void (*f) (token*, token*, token*))
{
  token input[consum], output1[prod1], output2[prod2];
```

```
    int i;

    for(i = 0; i < consum; i++) {
            readToken(*ch_in, &input[i]);
    }
    f(input, output1, output2);
    for(i = 0; i< prod1; i++) {
            writeToken(*ch_out1, output1[i]);
    }
    for(i = 0; i< prod2; i++) {
            writeToken(*ch_out2, output2[i]);
    }
}
```

The actor process constructor C-functions `actorXYSDF` have the following arguments: (1) the number of consumed tokens on each input arc, (2) the number of produced tokens on each input arc, (3) the input channels, (4) the output channels, and (5) the function that is executed during each firing. The actor functions first consume the input tokens from each input channel, then apply their function on the consumed inputs, and finally produce the tokens on each output channel.
Finally, each function argument needs to be translated into the corresponding C-code. This requires that the number of produced input and output tokens of the actor is matched between by the function. To illustrate this, the original ForSyDe code for actor P_4 and the C-code for the implementation is given.

```
// Original ForSyDe model:
// p_4 = actor12SDF 1 (3 ,1) f_4
//   where f_4 [ x ] = ([ x , x +1 , x +2] ,[ x ])
void f_4(token* in, token* out1, token* out2) {
  out1[0] = in[0];
  out1[1] = in[0] + 1;
  out1[2] = in[0] + 2;
  out2[0] = in[0];
```

Following these steps, the resulting C-code implements the same functionality as the original ForSyDe SDF model in Haskell. Although there is no software synthesis tool for the automation of the synthesis procedure available yet, the development of an automated tool is considered mainly an engineering effort and planned for future work. The software synthesis method can also be extended to support other software platforms based on the following ideas. If the target platform is a single-processor with real-time operating system (RTOS), then the actors will be implemented as tasks and the FIFO buffers are implemented as message queues. As long as the FIFO buffers have sufficient buffer size, the RTOS-scheduler will be able to schedule the system dynamically. The method can also be extended to a multi-processor implementation, where the scheduling for each processor has to be derived, and where the underlying software architecture for the multiprocessor platform has to provide a correct implementation of the FIFO access functions `writeToken`, `readToken`, and `writeToken`.

6. Design Space Exploration

The design space exploration (DSE) phase aims to find an efficient solution for an IMA System on the shared target platform, so that all applications (aircraft functions) meet their design constraints and the global design constraints are satisfied. So far, there is a lack of automated DSE tool support. The huge design space in IMA or comparable industrial systems poses a big challenge, and also an efficient DSE tool will only be able to search a part of it. In addition, for safety-critical systems, it is critical that the DSE tool only provides solutions that satisfy all the design constraints in the final implementation. Pimentel gives a very good introduction into the general aspects and challenges of design space exploration [24]. The correct-by-construction design process in this paper is based on carefully selected application models, with limited but few modelling elements and predictable

and composable platform components, which gives a sound base for efficient and design space exploration, where correctness guarantees can be given.

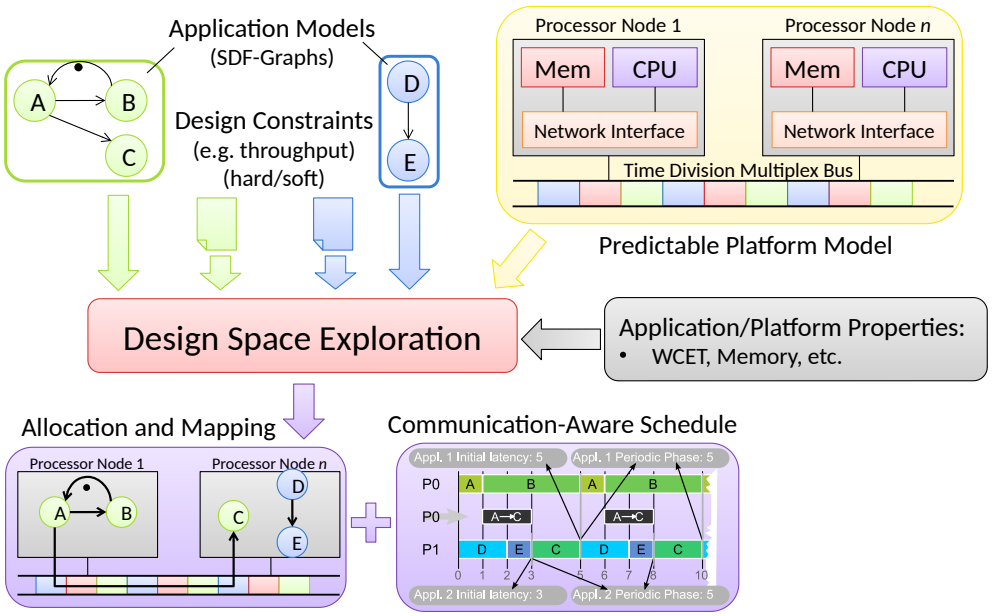

Figure 8 – Overview of the design space exploration tool.

Figure 8 gives an overview of the DSE tool. The DSE tool uses the following inputs:

1. a set of analysable application models (Section 3) in the form of workload models, where the function is abstracted by workload parameters, for instance, in the form of an SDF-graph.
2. a set of non-functional design constraints for each application, e.g. timing requirements,
3. a set of design constraints that the system has to fulfil, e.g. power, energy, memory requirements or design rules to satisfy a safety standard,
4. a model of the target platform model (Section 4),
5. and performance parameters for mappings of application model elements to platform components, like worst-case execution time, worst-case communication time or memory consumption.

The ForSyDe modelling framework ensures that application models comply with well-defined and analysable computation models, from which an application workload model can be extracted. The DSE tool uses the performance analysis models for these models of computation internally. For instance, the analysis of SDF-graphs to calculate a schedule and the required buffer sizes in case such a schedule exists. Performance guarantees can then be given due to the requirements on the composability and predictability of the target platform. The DSE tool searches the design space to identify feasible implementations.
The feasibility of the concepts has been demonstrated in the DESYDE tool [25]. Figure 8 illustrates a typical DSE problem, which DESYDE can solve. Two application models with individual design constraints shall be implemented on a multiprocessor platform, where processor nodes communicate via a time division multiplex (TDM) bus. Parameters in the form of worst-case execution time or memory size are known for each SDF-actor for the available processor nodes. In addition, the size of the data sent via signals between the actors is given, and parameters like worst-case communication time for a given data size via the TDM bus are known. A typical task for the DSE tool is to find an optimised implementation that satisfies all design constraints, where the optimisation could be to minimise the number of used processors. Figure 8 shows a possible result of the design space exploration, where the following results are returned:

1. an *allocation* of platform components required for the design,

2. a *mapping* of application computation and communication components to platform resources,
3. a set of *schedules* for computation and communication,
4. *performance metrics* to show the efficiency of the solution for design objectives like throughput, latency, memory consumption, utilisation and minimal buffer sizes.

In this example, two processor nodes are allocated to satisfy the design requirements. The green application's actors, A and B, are mapped to processor node 1, and actor C of the green application and the blue application's actors D and E are mapped to processor node 2. Since actor A and C are on different processor nodes, the signal between actor A and actor C is mapped to the TDM bus. All other signals map to the shared memory communication on their assigned processor nodes. On processor node 1, a repetitive static schedule, AB, controls the execution of the actors A and B, while on processor node C, the repetitive schedule DEC is executed. The communication between the green actor A on processor node 1 and the green actor C on processor node 2 is scheduled in time slots 1 and 2 of a TDM schedule, where a round consists of five time slots. The DSE tool reports performance metrics, like latency and throughput, calculated via the periodic phase.
The DESYDE DSE tool supports the SDF model of computation and uses constraint programming to formulate the DSE problem and a constraint solver to solve the DSE problem. Current work focuses on the new design space exploration tool IDESYDE, which, based on the concept of design space identification [26] aims at providing more flexibility, scalability and efficiency. It shall also support additional models of computation, like task graphs, more complex platforms and additional solvers.

7. Case Study

To evaluate the correct-by-construction design approach a case study in an industrial setting has been conducted. The overall objective of this case study is to evaluate the correct-by-construction design approach, which aims to shift the focus from integration and the analysis of the detailed final design to an earlier conceptual design phase, where functional and non-function requirements are formulated at a high level of abstraction. The case study consists of the following five activities:

1. development of a multifunctional demonstrator to evaluate correctness and scalability of the design space exploration (DSE) and the correct-by-construction design methodology for a highly distributed system over a network,
2. formulation of non-functional requirements, e.g. safety, energy, latency, or throughput,
3. conducting a design case following the early concept flow as specified in Section 2.1,
4. evaluating the suitability of the design space exploration tool to find a feasible solution, and
5. evaluating the scalability of the design space exploration tool and the correct-by-construction design methodology.

The multifunctional demonstrator has been defined to illustrate, challenge, and evaluate the suggested design flow. Special focus lies on safety properties that shall be maintained during the design flow in the form of functional, performance, and energy constraints. The selected aircraft functionality for this is a combination of two functions, one air flight information function together with one AESA (Active Electronically Scanned Array) radar function. The safety aspects are highlighted in the interaction between the first function needed for safe control of the aircraft and the second function acting like a resource-demanding payload, both needing processing capabilities supported by the avionic platform. The case study uses the following top-level requirements for the design case:

- functionality given as sequence of specified tasks for the flight information function,
- functionality specified by a mathematical algorithm for the AESA radar processing function,
- input data size and periodicity for the flight information function,

- input data size and periodicity for the AESA radar processing function,
- energy budget for each of the two functions,
- overall safety requirement stating that the flight information function is not allowed to fail due to any single fault in the design method or in the platform with mapped functionality,
- a safety requirement that a function that is not allowed to fail shall not be exposed to any interference from other functions that may result in not being fulfilled,
- real-time latency requirements for the AESA radar processing function, and
- real-time latency requirements for the flight information function.

During model refinement and decomposition into components of the two top-level functionalities, e.g. creation of the IMA Models, engineering knowledge is applied, and design decisions are made. These are captured and defined as rules to be formally expressed and used as constraints during the following DSE design process as well as platform enhancement. Rules to be demonstrated are:

- periodic scheduling shall be used for functions that are not allowed to fail,
- resources in platform components used for functions that are not allowed to fail must be utilised below 50%,
- energy shall be handled in a way that maximum peak power is below two times mean value, and
- each signal processing module is connected to at least one communication module.

The following platform characteristics shall be used for the demonstration:

- heterogeneous selectable processing modules of the types singlecore, multicore, and system-on-chip (SoC),
- communication modules based on Ethernet with different performance characteristics,
- internal communication within multicore or SoC are specified, and
- remote data concentrator (RDC) and Remote Control Electronics (RCE) modules for input and output communicate using Ethernet.

During the evaluation of the proposed methodology, each aircraft function is first expressed as a high-level model, then stepwise refined and de-composed into a detailed model used as input for the DSE tool. In parallel, the platform modules are selected and instantiated to define the IMA platform to be used by the DSE process. Characteristics are given for the functionality, e.g. processing and communication needs, as well as for the platform modules, e.g. processing and communication capability.

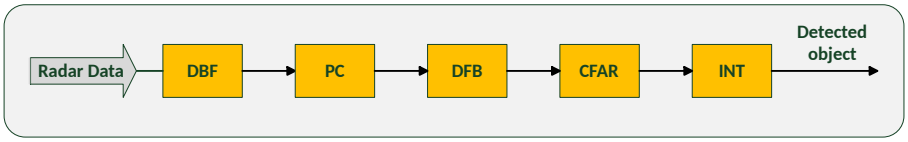

Figure 9 – Overview of AESA radar function processing steps.

The AESA radar is a processing and communication demanding function. Sampled data from 64 parallel antenna elements is streaming through the signal processing chain to identify the detected target as output, see Figure 9. The complete function has been modelled and simulated in ForSyDe and verified with radar data [27, 28].

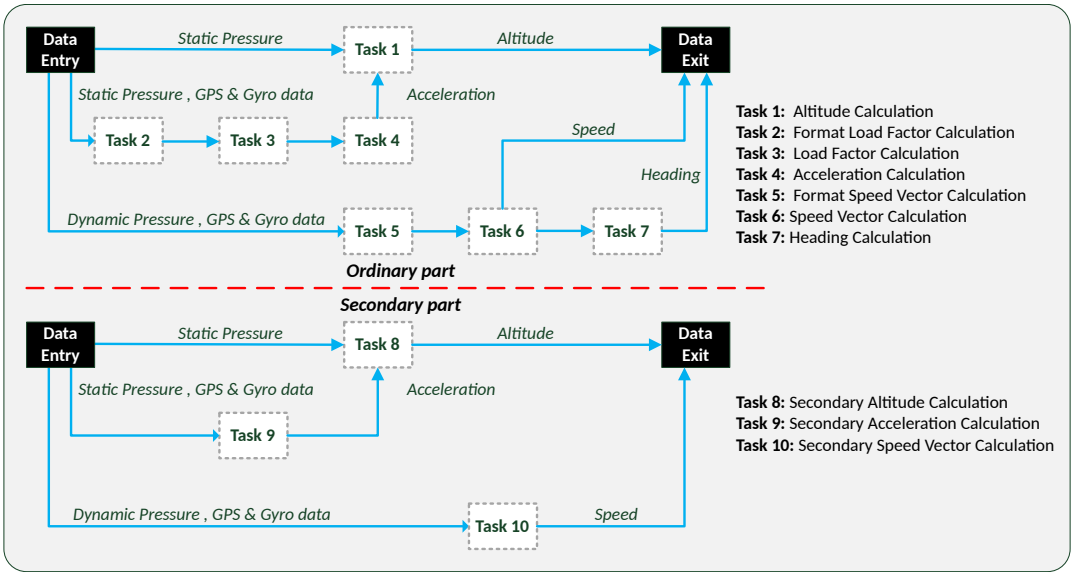

Figure 10 – IMA model of flight information function.

The flight information function in Figure 10 consists of two independent parts due to the functional safety requirement, one ordinary part and another secondary redundant part. The processes execute periodically, denoted as tasks, according to one of the rules and communicate through the indicated connections. Real-time latency requirements for the flight information function are defined from 'Data Entry' to 'Data Exit'.

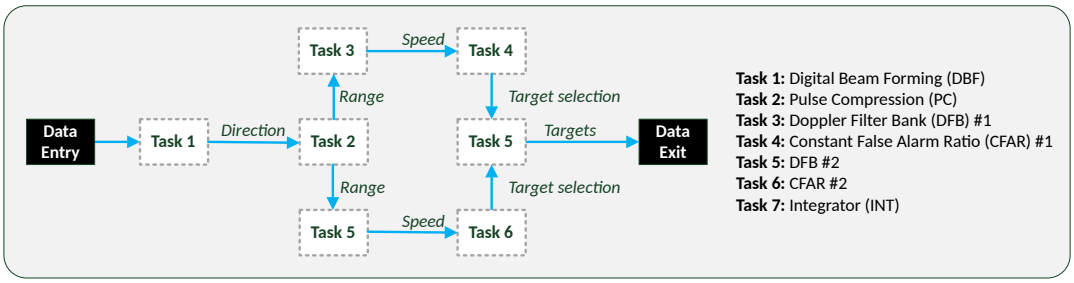

Figure 11 – IMA model of AESA radar function.

A decomposed version of the AESA radar function, see Figure 11, has been modelled with two parallel chains to allow the DSE tool to utilise the parallel behaviour of the function further and to satisfy the demanding data input requirement. Real-time latency requirements for the AESA radar function are defined from 'Data Entry' to 'Data Exit'. Here the latency requirement is more relaxed compared to the flight information function. Instead, the data size and throughput requirements at 'Data Entry' feeding 'Task 1' is demanding.
A module library has been defined and populated for the case study with basic modules needed for processing, communication, and memory. It is used to build an IMA platform with the resources needed for the two functions to execute simultaneously. Examples of characteristics for each module in the library are:

- the maximum performance per core in operations per second,
- the maximum communication volume per port in bytes per second,
- an equation to predict the execution time based on the number of operations needed,
- a core connectivity scheme, i.e. network architecture and possible data channels, and
- the energy consumption per operation.

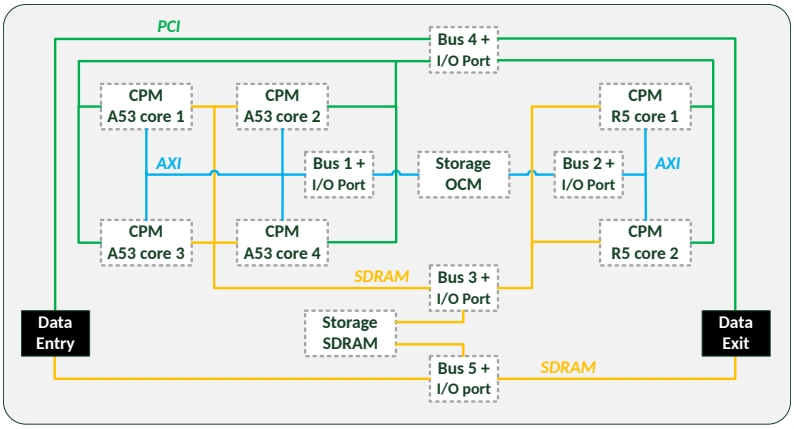

Figure 12 – IMA Platform used by DSE in the case study.

The IMA platform in Figure 12 is based on system-on-chip (SoC) technology with heterogeneous processing cores as well as communication channels. This emphasises the necessity to clearly express the computation needs for each task in the IMA models and to provide the equations related to the modules, so that the DSE tool can calculate the actual execution time depending on the mapping of a function to a specific core type. This is necessary to achieve complete separations between the top and bottom abstraction levels of the hour glass in Figure 1 — the worst case execution time cannot be a property of the IMA model task. Also, here the latency is defined from 'Data Entry' to 'Data Exit', where the same symbols are used as in the IMA model. The difference is that at this abstraction level, it should be interpreted as electrical signals at the interface.
The case study has evaluated the correct-by-construction design methodology including design space exploration concerning the initial goal to shift focus from integration and analysis of detailed designs to early conceptual design. Three maturity aspects have been evaluated: the methodology and related concepts, capabilities to capture the design problem, and the DSE tool. The focus has been on the following aspects:

- **Dependability - functional requirements:** The DSE tools ability to propose an explainable solution together with resource utilisation within modules and architecture for functional correctness.
- **Dependability, non-functional requirements:** The DSE tools ability to maintain and show non-functional correctness from conceptual level down to proposed platform mapping.
- **Constraints:** Evaluate rules in form of design constraints as guidance for the DSE tool to maintain non-functional requirements.
- **Scalability:** Possibility for the DSE to find solutions when size and number of functions as well as conceptual requirements are increasing.

The case study study covered all five activities listed in the beginning of this section. It showed that the ForSyDe modelling framework is capable to model and simulate a complete AESA radar signal processing function [27, 28]. A simplified version of the design space exploration problem consisting of a multifunctional mapping of a flight information and a radar function has been formulated and solved using the DSE tool IDESYDE [17, 26]. The major part of the four aspects listed above were successfully addressed; the scalability and explainability need to be further investigated. The safety requirements have been expressed and successfully analysed by the DSE tool.
Regarding the overall correct-by-construction design methodology, the case study showed very promising results. The proposed methodology and its associated concepts have shown to be in line with the industrial needs and challenges of the near future. However, it is too early to draw general conclusions, since the case study was relatively small and parts of the ForSyDe tools and libraries are still at a low technical readiness level. Section 8 gives a more detailed view about the state of the ForSyDe tools and libraries as part of the correct-by-construction design flow.

8. Current Status of Correct-By-Construction Design Flow

The correct-by-construction design flow, illustrated in Figure 2, and its associated concepts have been formulated and rely on a sound formal base. Different parts of the methodology have different maturity levels, and the lack of a common data exchange format prevents so far a good connection between the different activities and tools. In the following, the maturity level and tool support for the ForSyDe design flow are evaluated before a summary of the state of the complete flow is given.

Application Modelling The underlying ForSyDe modelling concepts have a sound base in the underlying theory of models of computation and can be considered stable and mature. ForSyDe provides modelling libraries for several models of computation, among others several variants of dataflow MoCs, synchronous MoC, and continuous-time MoC. This enables the modelling and simulation of heterogeneous systems together with their environment. The capabilities of the ForSyDe modelling framework have been demonstrated by the modelling of the complete processing chain of an AESA antenna and the simulation and validation against test input data [27]. Starting with FORSYDE-SHALLOW, which has been discussed in this paper, there are now additional ForSyDe modelling frameworks available, which all rely on the same fundamental concepts. FORSYDE-SYSTEMC [29] uses SystemC as modelling language, while FORSYDE-ATOM [28] extends the concepts of orthogonalisation with Haskell as modelling language. The different ForSyDe libraries are publicly available under a permissive license on GitHub [17].

Platform Modelling ForSyDe has so far only limited platform modelling support, which is used for the design space exploration framework, where platforms are composed of processor nodes and communication components. Since individual parameters are associated with these components, it is possible to model on-chip and distributed platforms and use them as part of the design space exploration. Recent work [26] takes a step in the direction of a more flexible platform model by introducing intermediate layers, enabling to model static or dynamic schedulers as the first step to virtual components.

Synthesis and Code Generation The principles for synthesis and code generation from ForSyDe models can be considered stable. However, there is currently only tool support for hardware synthesis in the form of the FORSYDE-DEEP tool [30, 17], which generates VHDL-code from ForSyDe models belonging to the synchronous model of computation. FORSYDE-DEEP has also been used in the radar signal processing system case study to generate the hardware for the pulse compression component on an FPGA [27]. For software synthesis, the general principles and synthesis steps have been formulated and conducted manually for the SDF model of computation, but there is currently no tool automating the synthesis.

Design Space Exploration Due to its focus on safety-critical systems, the methodology has a sound foundation through the formal nature of the application model and the focus on predictable and composable platform components. The design space exploration in ForSyDe exploits this by employing research results related to performance analysis methods for analysable models of computation. The DESYDE DSE tool [25] has demonstrated its potential through case studies using a set of applications targeting a shared multiprocessor platform. The results showed that DESYDE could find the optimal solution for smaller problems, but since the DSE problem suffers from state explosion, only a part of the solution space can be explored for larger problems. The new tool IDESYDE uses the novel concept of design space identification [26]. It aims at providing more flexibility, scalability and efficiency, where one of the key questions is how the designer's experience and knowledge can be integrated into the search strategy of the DSE tool. IDESYDE has been used and evaluated in the case study (Section 7).

Tool Integration The ForSyDe framework provides methods and tools for different phases of the correct-by-construction design process, but the tools are so far only very loosely connected, since there is no common and established data exchange format. FORSYDE IO [17] proposes a new data exchange format with associated support libraries that the ForSyDe tools should use to enable a seamless tool flow from the system model down to the final implementation.

FORSYDE IO is under development, its data exchange format is used in the new DSE tool IDESYDE for input and output data.

Currently, there is a conceptually complete, but not automated, chain from application model to final implementation, which does not allow a final statement on the full potential of the proposed correct-by-construction design methodology yet. The concepts for the overall methodology and the individual parts have been defined, and there are software libraries and tool support for a large part of the different activities of the methodology. Also, there have been very promising case studies on different parts of the methodology. Nevertheless, the connection between the activities and tools is so far not supported by a common data exchange format, making it at this stage difficult to show and automate the entire flow from the system model to the final implementation.

9. Conclusion and Future Work

The paper presented the concepts for a novel correct-by-construction design methodology, combining an IMA-based design methodology with the ForSyDe design flow. The methodology emphasises separations of concerns and orthogonalisation in several forms and at different abstraction levels. The IMA-based methodology cleanly separates the function models from the platform modules. At the same time, the ForSyDe framework provides a clean separation of computation and communication as part of the modelling methodology based on well-defined computation models. The design flow enables to design systems at a high level of abstraction and provides at the same time a clear path to implementation with tool support for modelling, design space exploration and synthesis. The design space exploration tool plays a key role in the IMA-based design methodology. It connects the design of the functional models with the design of the platform, by an automated exploration of the design space to identify a correct and efficient mapping of functions models to platform modules that satisfy the design constraints including safety constraints. The proposed correct-by-construction design methodology has been evaluated by an avionics case study in an industrial context. A radar processing system has been modelled and simulated, the design space exploration tool has been used to find an implementation of two avionics functions on a shared avionics platform, and the suitability of the methodology and the concepts have been assessed. The case study showed very encouraging results and validated the suitability of the underlying concepts of the methodology. However, to assess if the suggested methodology is fully acceptable and compliant with avionic system development requirements, further investigations and additional tool development are needed to show the scalability of the methodology and their associated tools for larger industrial examples. Future work aims to push the design entry for the functional design to an even higher level of abstraction within the Vinnova NFFP7 TRANSFORM project. Here, the idea is to start with an initial system model and top-level requirements and to simultaneously transform the initial system model by well-defined design transformations into a model of the same abstraction level as the application model in this paper, to establish the connection with the presented correct-by-construction design methodology.

10. Acknowledgements

This research was partially funded by the Sweden's Innovation Agency (Vinnova), by the NFFP7 project 2019-02743: TRANSFORM - Design transformation for correct-by-construction design methodology, NFFP7 project 2017-04892: Correct by construction design methodology (CORRECT), and the ITEA3 project 2018-02228: PANORAMA - Boosting Design Efficiency for Heterogeneous Systems.

11. Contact Author Email Address

Contact author: Ingo Sander, ingo@kth.se

12. Copyright Statement

References

[1] C. B. Watkins. Integrated modular avionics: Managing the allocation of shared intersystem resources. In *Digital Avionics Systems Conference*, pages 1–12, October 2006.

[2] SAE Aerospace. Aerospace recommended practice – Guidelines for development of civil aircraft and systems (ARP4754A). December 2010.

[3] SAE Aerospace. Aerospace recommended practice – Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment (ARP4761). December 1996.

[4] Aeronautical Radio, INC. Avionics application software standard interface part 1 – required services (ARINC653P1-3). November 2010.

[5] Integrated modular avionics (IMA) development guidance and certification considerations (RTCA DO-297). November 2005.

[6] Christopher B. Watkins and Randy Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, 2007.

[7] Gregg Bartley and Barbara Lingberg. Certification concerns of integrated modular avionics (IMA) systems. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, October 2008.

[8] Thomas Gaska, Chris Watkin, and Yu Chen. Integrated modular avionics - past, present, and future. *IEEE Aerospace and Electronic Systems Magazine*, 30(9):12–23, 2015.

[9] SCARLETT: SCAlable & ReconfigurabLe Electronics plaTforms and Tools. `https://cordis.europa.eu/project/id/211439`.

[10] ASHLEY: Avionics Systems Hosted on a distributed modular electronics Large scale dEmonstrator for multiple tYpe of aircraft. `https://cordis.europa.eu/project/id/605442`.

[11] IEEE International roadmap for devices and systems (IRDS). `https://irds.ieee.org/`.

[12] Marco Di Natale and Alberto Luigi Sangiovanni-Vincentelli. Moving from federated to integrated architectures in automotive : The role of standards, methods and tools. *Proceedings of the IEEE*, 98(4):603–620, 2010.

[13] Joseph Sifakis. System design automation: Challenges and limitations. *Proceedings of the IEEE*, 103(11):2093–2103, 2015.

[14] Alberto Sangiovanni-Vincentelli. Quo vadis, SLD? Reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[15] Ingo Sander, Axel Jantsch, and Seyed-Hosein Attarzadeh-Niaki. ForSyDe: System design using a functional language and models of computation. In Soonhoi Ha and Jürgen Teich, editors, *Handbook of Hardware/Software Codesign*, pages 99–140. Springer Netherlands, 2017.

[16] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[17] ForSyDe: A methodology for Formal System Design. `https://forsyde.github.io/`.

[18] Kees Goossens, Martijn Koedam, Andrew Nelson, Shubhendu Sinha, Sven Goossens, Yonghui Li, Gabriela Breaban, Reinier van Kampenhout, Rasool Tavakoli, Juan Valencia, Hadi Ahmadi Balef, Benny Akesson, Sander Stuijk, Marc Geilen, Dip Goswami, and Majid Nabi. *Handbook of Hardware/Software Codesign*, chapter NOC-Based Multi-Processor Architecture for Mixed Time-Criticality Applications. Springer, 2017.

[19] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems*, 2009.

[20] Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. A multicore processor for time-critical applications. *IEEE Design and Test*, 35(2):38–47, 2018.

[21] Ingo Sander and Axel Jantsch. System synthesis based on a formal computational model and skeletons. In *Proceedings IEEE Workshop on VLSI'99*, pages 32–39, Orlando, Florida, USA, April 1999. IEEE Computer Society.

[22] Zhonghai Lu, Ingo Sander, and Axel Jantsch. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the 15th International Symposium on System Synthesis*, pages 86–91, Kyoto, Japan, October 2002.

[23] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[24] A. D. Pimentel. Exploring exploration: A tutorial introduction to embedded systems design space exploration. *IEEE Design Test*, 34(1):77–90, February 2017.

[25] Kathrin Rosvall and Ingo Sander. Flexible and trade-off-aware constraint-based design space exploration for streaming applications on heterogeneous platforms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23(2), January 2018.

[26] Rodolfo Jordão, Ingo Sander, and Matthias Becker. Formulation of design space exploration problems by composable design space identification. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, February 2021. Accepted for publication.

[27] George Ungureanu, Timmy Sundström, Anders Åhlander, Ingo Sander, and Ingemar Söderquist. Formal design, co-simulation and validation of a radar signal processing system. In *Forum on Specification and Design Languages (FDL 2019)*, Southamption, United Kingdom, September 2019.

[28] George Ungureanu, José E. G. de Medeiros, Timmy Sundström, Ingemar Söderquist, Anders Åhlander, and Ingo Sander. ForSyDe-Atom: Taming complexity in cyber physical system design with layers. *ACM Trans. Des. Autom. Electron. Syst.*, 2021.

[29] S.H. Attarzadeh Niaki, M.K. Jakobsen, T. Sulonen, and I. Sander. Formal heterogeneous system modeling with SystemC. In *Forum on Specification and Design Languages (FDL 2012)*, pages 160–167, Vienna, Austria, 2012.

[30] Alfonso Acosta. Hardware synthesis in ForSyDe. Master's thesis, School for Information and Communication Technology, Royal Institute of Technology (KTH), Stockholm, Sweden, 2007. KTH/ICT/ECS-2007-81.