2022 IEEE XXIX International Conference on Electronics, Electrical Engineering and Computing (INTERCON) | 978-1-6654-8636-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INTERCON55795.2022.9870103

A High-Level Synthesis Approach Applicable to Autonomous Embedded Systems

Denis S. Loubach
Department of Computer Systems, Computer Science Division
Aeronautics Institute of Technology – ITA
São José dos Campos, São Paulo, Brazil
ORCID 0000-0003-1595-3448

***Abstract*—This paper proposes a high-level synthesis approach applicable to future autonomous systems, *i.e.* embedded systems, in the context of domain-specific architectures (DSA) along with domain-specific languages (DSL). That enables the system's design and specification to focus on the functions and requirements that such a system must provide and, at the same time, obtain an executable model. We show an illustrative example to demonstrate the potential and applicability of our approach down to automatic hardware description language generation and synthesis.**

***Index Terms*—High-level synthesis (HLS); Embedded domain-specific languages (EDSL); Domain-specific architecture (DSA); Cyber-physical systems (CPS); Embedded systems; Autonomous systems; Formal models of computation.**

I. Introduction

Nowadays, it is getting harder and harder to come up with new performance enhancements in the computer architecture area when compared to the period of about 60 years ago. It is mainly due to the end of Dennard's scaling, dark silicon, and the considerable slow down of Moore's law.

In this sense, the adoption of domain-specific architectures (DSA) along with domain-specific languages (DSL) brings an interesting alternative to face these present issues in the long run. Different from general-purpose architectures, the hardware can be specific and optimized even in runtime for a specific domain when taking into account DSA, as discussed in [1].

This modern strategy, *i.e.* DSA, can play a central role when conceiving and developing newly embedded and cyber-physical systems within challenging scenarios such as autonomous systems. Autonomous systems have to make decisions without human intervention [2]. In this sense, they are highly dependent on the software and hardware infrastructure.

A paradigm shift has to consider new strategies for the design of autonomous systems, for example, a minimum level of runtime adaptivity and high-level synthesis together with DSA to keep up with the dynamic aspects of such a domain.

Notice, however, that traditional embedded designs highly depend on a hardware platform during the specification and conception project phases. The possibility of having an application's high-level model that could be later mapped to a high-level hardware platform model (*e.g.,* a hardware template) presents itself as a highly desirable feature.

Considering this context, our paper proposes a high-level synthesis (HLS) approach applicable to future autonomous systems, *i.e.* embedded systems.

We briefly present a summary and analysis of formal models of computation, domain ontology, and reconfigurable processors as the basis for our proposed approach, according to [3]. Next, we show two levels of abstraction, specification and implementation, where we extend previous research [3] to add a *high-level synthesis approach*, *i.e.* a simple but powerful embedded domain-specific language (EDSL) named ForSyDe-Deep [4]. The high-level synthesis is a refinement in the design where a more abstract specification (*e.g.* application model) is translated to a less abstract one (*e.g.* implementation model). We achieve this by a set of well-defined mapping rules [3].

In turn, this enables the system's design and specification to focus on the functions and requirements that such a system must provide and, at the same time, obtain an *executable model*. With such a model, it is possible to "execute the specification", *i.e.* to simulate the system at a high level of abstraction before moving to the implementation and synthesis phases.

II. Related Works

Wakabayashi *et al.* [5] discuss the trade-offs involving HLS for field programmable gate array (FPGA), and compilers for central processing unit (CPU) and graphics processing unit (GPU). They conclude FPGA and HLS generate better results in terms of performance than fixed computation platforms. In the research [6], Lee *et al.* address an approach to map applications to coarse-grained architectures using HLS, and they point it as a complex task. Hu *et al.* [7] investigate possible errors in the HLS approach caused by the complexity and error-prone compiling process. Their study was in terms of global common subexpression elimination. The idea to optimize the area and power consumption of reconfigurable hardware was taken into account in [8]. The authors claimed a considerable figure reduction. Our approach here considers the use of formal models and ontology to generate a virtual implementation model that can be synthesized.

III. Background

Next, we introduce key concepts used in the way from high-level modeling down to synthesis in modern embedded

systems design.

The idea of using *formal models* to develop embedded systems dates long ago [9]. For example, Lee and Sangiovanni-Vincentelli [10] proposed a meta-model to address models of computation (MoC) properties. They named it *tagged signal model* (TSM). In this sense, applications are assumed as a network of processes handling signals. A process *P* has defined behaviors and relates input S^I and output S^O signals. And it is considered as *functional* when it has a single value mapping, such as $F : S^I \rightarrow S^O$.

MoCs can be *timed*, *e.g.* synchronous (SY) [11], or *untimed e.g.* synchronous dataflow (SDF) [12], scenario-aware dataflow (SADF) [13].

At the same time a MoC provides semantics including execution, synchronization and concurrency to an application, *domain ontology* provides meaning and unambiguous understanding [14]. In [3], two domain ontologies are created and used to derive a classification and composition system where an *application model* can be mapped to a *virtual platform model*, addressing two distinct levels of abstraction: *specification* and *implementation*. The result of that design flow is a *virtual implementation model*.

Notice also the increasing number of modern embedded systems based on a system on chip (SoC) containing a fixed microcontroller processor and also an FPGA device within the context of cyber-physical systems (CPS) [15], [16]. However, it is fundamental to separate the specification level from the implementation level to leverage the way embedded systems are designed.

We believe that a design flow addressing formal modeling, ontologies, and this separation of concerns, *i.e.* specification, and implementation are the main aspects for achieving high-level synthesis.

IV. Towards High-Level Synthesis

In the present research work, we extend the systematic design methodology proposed in [3] by including an *automatic hardware description language (HDL) generation* based on a resulting virtual implementation model.

The original steps from [3] take into account i) first the *modeling of an application* supported by a proper model of computation; ii) obtaining the *classes of the application model elements*, based on the application domain ontology; iii) the *mapping rules utilization*, after fixing the given constraints for choosing among programmable, reconfigurable, or heterogeneous devices; iv) obtaining the feasible *model elements and functional blocks* related to the virtual platform model; and finally, v) getting the resulting *virtual implementation model* (VIM).

To be able to automatically generate the final HDL based on the resulting VIM, and using very high speed integrated circuit (VHSIC) hardware description language (VHDL) here, we adopt ForSyDe-Deep [17] besides ForSyDe-Shallow [18].

ForSyDe-Deep is a domain-specific embedded language intended to keep an eye on systems structure and has an embedded compiler to translate the high-level code into VHDL.

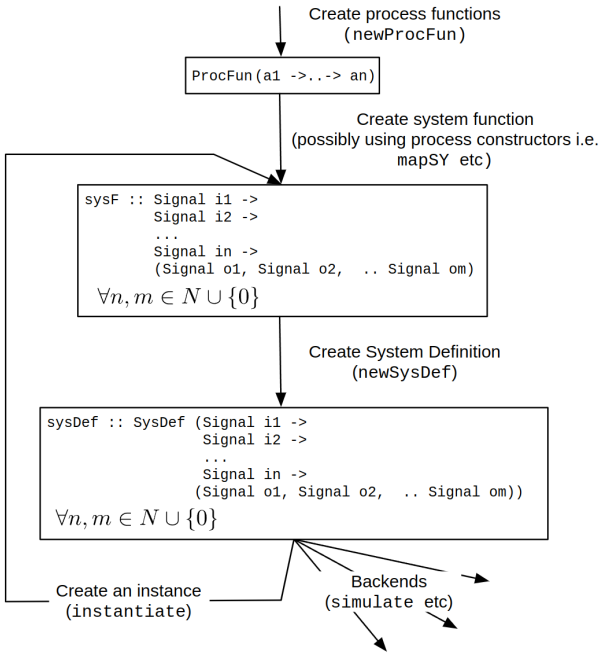

Figure 1. General ForSyDe-Deep code structure and flow [17].

Its deep-embedded application programming interface (API) only works with the synchronous model of computation so far. Both Deep and Shallow are implemented on top of Haskell, *i.e.*, a functional programming paradigm (FPP) language.

Therefore, another step enters the previously mentioned design flow: the *automatic code generation* able to be synthesized by a third-party tool, *e.g.* Intel-FPGA Quartus. ForSyDe-Deep can even generate a Quartus project. We are using Quartus Prime Standard version 16.0.0.

There are basically two options to integrate this new step into our design flow. One can still keep the original application model written in ForSyDe-Shallow and then port it to Deep. A second option is to go straight to ForSyDe-Deep when modeling an application based on the synchronous MoC. Here, we kept the Shallow version and ported it to Deep in our approach.

The ForSyDe-Deep code follows a well-defined structure, as described next and illustrated in Fig. 1.

1) process function definition, with `ProcFun`;
2) system function definition, using for example constructors such as `mapSY`, `zipWithSY`;
3) system definition in terms of inputs and outputs, with `newSysDef`;
4) adding the simulation part, with `simulate`; and
5) generating the VHDL code, with `writeVHDLOps vhdlOps`.

That structure has to be observed to allow for automatic HDL code generation at the end of our proposed design flow.

V. Illustrative Example

To demonstrate the *automatic code generation step* included in the design flow and the *synthesis* part, we adapted and

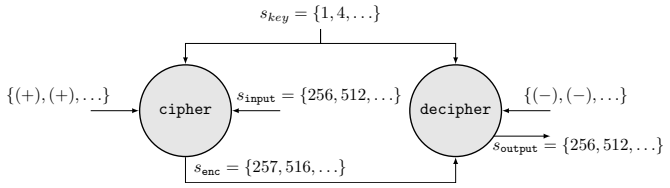

Figure 2. Encoder and decoder hierarchical processes network, extended from [3].

extended the application model described in [3].

That is an encoder and decoder process network, as illustrated in Fig. 2. It comprises a `cipher` and `decipher` processes. The `cipher` gets a plain text and a key, given by the signals `s_key` and `s_input`, and produces the encoded result in the signal `s_enc` considering a given ciphering function. On the other hand, the `decipher` gets the key and the encoded signal, finally producing the decoded information in the signal `s_output`.

Here, we use fixed functions for `cipher` and `decipher`. For the sake of clarity, we show simple functions for both. However, they can be as complex as required by a given application.

Listing 1 shows the ForSyDe-Deep code where `cipher` and `decipher` functions are defined, as simple add and sub. They follow the i) process function definition; ii) system function definition; and iii) system definition in terms of inputs and outputs.

Listing 1. Functions definition in Haskell/ForSyDe-Deep code snippet.

```
{-# LANGUAGE TemplateHaskell #-}
-- this stands for the application model
module EncoderDecoderDeep where

import ForSyDe.Deep
import Data.Int (Int32)

-- add function
-- 1st step: process function (PF) definition
add_pf :: ProcFun (Int32 -> Int32 -> Int32)
add_pf = $(newProcFun
          [d|
             add_pf :: Int32 -> Int32 -> Int32
             add_pf x y = x + y
          |])

-- 2nd step: system function (SF) definition
add_sf :: Signal Int32 -> Signal Int32 -> Signal Int32
add_sf = zipWithSY "add_sf" add_pf

-- 3rd step: system definition (SD) in terms of inputs/
    outputs
add_sd :: SysDef (Signal Int32 -> Signal Int32 -> Signal
    Int32)
add_sd = newSysDef add_sf "add_sd" ["input1","input2"] ["
    output"]

-- sub function
-- 1st step: process function (PF) definition
sub_pf :: ProcFun (Int32 -> Int32 -> Int32)
sub_pf = $(newProcFun
          [d|
             sub_pf :: Int32 -> Int32 -> Int32
             sub_pf x y = x - y
          |])

-- 2nd step: system function (SF) definition
sub_sf :: Signal Int32 -> Signal Int32 -> Signal Int32
sub_sf = zipWithSY "sub_sf" sub_pf

-- 3rd step: system definition (SD) in terms of inputs/
    outputs
sub_sd :: SysDef (Signal Int32 -> Signal Int32 -> Signal
    Int32)
sub_sd = newSysDef sub_sf "sub_sd" ["input1","input2"] ["
    output"]
```

Next, Listing 2 shows the resulting virtual implementation model definition as a new system function definition, following the structure as shown in Fig. 2.

As one can observe in that listing, the `lambaExample` is an *executable model*, *i.e.* it can be simulated as described by the end of the code.

Listing 2. System function definition based in Haskell/ForSyDe-Deep code.

```
-- application model as a new system function
-- system function (SF), based on previous processes
    functions
lambdaExample_sf
  :: Signal Int32 -> Signal Int32 -> (Signal Int32, Signal
      Int32)
lambdaExample_sf s_key s_input = (s_enc,s_output)
  where
    s_enc    = (instantiate "add_sd" add_sd) s_input s_key
    s_output = (instantiate "sub_sd" sub_sd) s_enc s_key

-- system definition (SD)
lambdaExample_sd
  :: SysDef
       (Signal Int32 -> Signal Int32 -> (Signal Int32,
           Signal Int32))
lambdaExample_sd =
  newSysDef lambdaExample_sf "lambdaExample_sd" ["s_key","
      s_input"] ["s_output"]

-- simulation setup
lambdaExample_simulation
  :: [Int32] -> [Int32] -> ([Int32], [Int32])
lambdaExample_simulation = simulate lambdaExample_sd
-- run that to simulate
-- > lambdaExample_simulation [1, 4, 6, 1, 1] [256, 512,
    1024, 2048, -512]
-- ([257,516,1030,2049,-511],[256,512,1024,2048,-512])
```

In the last step, after simulating the model and checking it works as required and expect, one can generate the VHDL to be synthesized in a third-party tool, in our case, Quartus. This is done by calling `compileVHDL`, as in Listing 3.

Listing 3. VHDL code generation in Haskell/ForSyDe-Deep code.

```
-- final step: hardware generation
compileVHDL :: IO ()
compileVHDL = writeVHDLOps vhdlOps lambdaExample_sd
 where vhdlOps = defaultVHDLOps{execQuartus=Just quartusOps}
       quartusOps = QuartusOps{action=FullCompilation,
                               fMax=Just 50, -- in MHz
                               fpgaFamiliyDevice=
                               Just ("Cyclone_V",
                                Just "5CSEMA4U23C6"),
                               -- Possibility for Pin
                                   Assignments
                               pinAssigs=[]
                              }
```

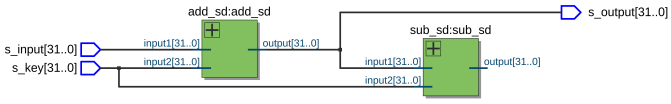

Figure 3. RTL view generated by Intel-FPGA Quartus.

That function automatically generates the code and calls Quartus to do the synthesis task. The register transfer level (RTL) view of the generated HDL is shown in Fig. 3. It represents the specifications defined in the previous listings, with some implicit/abstracted naming such as `s_enc`.

The complete listing of the original code in ForSyDe-Shallow is available in [19]. The complete listing of the code in ForSyDe-Deep (Listings 1 to 3) is available in [20].

The synthesis was made based on the Cyclone V SoC 5CSEMA4U23C6 chip, containing an FPGA.

VI. Discussion

One current limitation of our approach is that it does not comprise HDL generation for runtime reconfigurable models [16], [21]. So far, ForSyDe-Deep only foresees fixed functions in a process. Nevertheless, some strategies are possible when identifying runtime behavior in the specification model. In that case, it can be synthesized as a *partial reconfiguration* [22] in a reconfigurable hardware implementation (FPGA) or even be a *function pointer* in software (fixed microcontroller) implementations.

One of the key points of our proposed approach is to narrow down the gap considering high-level modeling, *i.e.*, abstract specification, and the implementation parts when abstraction is more and more decreased. With these, it is possible to speedup the development process. Also, we plan to integrate design space exploration (DSE) into our approach to better address performance parameters, *e.g.*, operating frequency and resource utilization.

VII. Conclusion

We introduced a *high-level synthesis approach* applicable to future autonomous systems, *i.e.* modern embedded systems, in the context of domain-specific architectures (DSA) along with domain-specific languages (DSL). That was an extension of previous research work.

We showed an example to demonstrate the potential and applicability of our approach down to automatic hardware description language generation and synthesis, where we used ForSyDe-Deep.

Future works should consider the runtime reconfiguration HDL generation to unlock the full potential of such an approach. Moreover, generated VHDL using synchronous elements should be addressed in demonstrations.

Acknowledgment

This research work is supported by regular research grant #2019/27327-6, São Paulo Research Foundation (FAPESP).

References

[1] D. S. Loubach, J. C. Marques, and A. M. da Cunha, "Considerations on Domain-Specific Architectures Applicability in Future Avionics Systems," in *FT2019. Proceedings of the 10th Aerospace Technology Congress*. Stockholm: Linköping ECP, 2019, pp. 156–161.

[2] A. Samadi, M. Ammar, and O. A. Mohamed, "Fault Tree Analysis And Risk Mitigation Strategies For Autonomous Systems Via Statistical Model Checking," in *2021 IEEE International Conference on Autonomous Systems (ICAS)*, 2021, pp. 1–5.

[3] D. S. Loubach, R. Bonna, G. Ungureanu, I. Sander, and I. Söderquist, "Classification and Mapping of Model Elements for Designing Runtime Reconfigurable Systems," *IEEE Access*, vol. 9, pp. 156 337–156 360, 2021.

[4] A. Acosta and I. Sander. (2018) ForSyDe-Deep: A Deep-Embedded Synthesizer for ForSyDe Models. https://forsyde.github.io/forsyde-deep.

[5] K. Wakabayashi, T. Takenaka, and H. Inoue, "Mapping Complex Algorithm Into FPGA with High Level Synthesis Reconfigurable Chips with High Level Synthesis Compared With CPU, GPGPU," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 282–284.

[6] G. Lee, S. Lee, and K. Choi, "Automatic Mapping of Application to Coarse-Grained Reconfigurable Architecture Based on High-Level Synthesis Techniques," in *2008 International SoC Design Conference*, vol. 01, 2008, pp. I–395–I–398.

[7] J. Hu, Y. Hu, L. Yu, W. Wang, H. Yang, Y. Kang, and J. Cheng, "Formal Verification of GCSE in the Scheduling of High-level Synthesis: Work-in-Progress," in *2020 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2020, pp. 1–2.

[8] B. L. Gal, C. Andriamisaina, and E. Casseau, "Bit-Width Aware High-Level Synthesis for Digital Signal Processing Systems," in *2006 IEEE International SOC Conference*, 2006, pp. 175–178.

[9] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–390, 1997.

[10] E. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.

[11] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[12] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[13] R. Bonna, D. S. Loubach, G. Ungureanu, and I. Sander, "Modeling and Simulation of Dynamic Applications Using Scenario-Aware Dataflow," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 5, 2019.

[14] D. S. Loubach, "Fundamental Concepts to Build a Runtime Reconfigurable Virtual Platform Model," in *2021 10th Mediterranean Conference on Embedded Computing (MECO)*, 2021, pp. 1–4.

[15] ——, "An Overview of Cyber-Physical Systems' Hardware Architecture Concerning Machine Learning," in *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, 2021, pp. 1–6.

[16] ——, "A Runtime Reconfiguration Design Targeting Avionics Systems," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, 2016, pp. 1–8.

[17] A. Acosta. (2009) System Design with ForSyDe-Deep. https://forsyde.github.io/forsyde-deep/forsyde-deep-tutorial.

[18] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe Formal System Design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 23, no. 1, pp. 17–32, Jan. 2004.

[19] D. S. Loubach. (2022) Encoder/Decoder Example. https://github.com/dloubach/encoder-decoder-example.

[20] ——. (2022) Encoder/Decoder Synthesis Example. https://github.com/dloubach/encoder-decoder-synth-example.

[21] ——, "An Analysis on Power Consumption and Performance in Runtime Hardware Reconfiguration," *International Journal of Embedded Systems*, vol. 14, no. 3, p. 277–288, 2021.

[22] J. W. Abdala Castro and A. Morales-Villanueva, "Exploring Dynamic Partial Reconfiguration in a Tightly-coupled Coprocessor Attached to a RISC-V Soft-processor on a FPGA," in *2021 IEEE XXVIII International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*, 2021, pp. 1–4.