Towards Coherent Semantics: a Quantitatively Typed EDSL for Synchronous System Design

Rui Chen
School of EECS
KTH Royal Institute of Technology
Stockholm, Sweden
ruich@kth.se

Ingo Sander
School of EECS
KTH Royal Institute of Technology
Stockholm, Sweden
ingo@kth.se

***Abstract*—We present SynQ, an embedded DSL (EDSL) targeting synchronous system design with quantitative types. SynQ is designed to facilitate semantically coherent system design processes by language embedding and advanced type systems. The current case study indicates the potential for a seamless system design process.**

***Index Terms*—embedded system design, synchronous systems, embedded domain-specific language, functional language**

I. Introduction

The ever-increasing complexity of embedded systems raises the need for more rigorous and automatic system design methodologies. According to [1], the success of such a methodology relies on four interrelated principles: separation of concerns, component-based design, semantic coherency, and correctness by construction. Among these principles, *separation of concerns* suggests that a design process shall be separated into successive steps in which each step addresses concerns on a specific aspect. Meanwhile, *semantic coherency* means that successive design steps can be rigorously related by their semantics.

Addressing both the separation of concerns and semantic coherency in the same design flow is challenging because dedicated domain-specific languages (DSLs) are desired to effectively address the concerns involved in each step. Consequently, these DSLs are hard to unify in terms of their semantics. For instance, we may consider a C-based design process following the general three-stage (specification, proceduralisation/modelling and materialisation/implementation) system design model proposed in [2]. To address concerns at each stage, three different languages are required, including a specification language, e.g. the ANSI/ISO C Specification Language (ACSL) [3], a modelling language (C in our case), and an implementation language, e.g. the LLVM IR [4]. With these languages involved, at least two semantics are associated with C models: *axiomatic semantics* (Hoare Logic) given by ACSL and *operational semantics* implied by the IR. It is challenging to relate these semantics formally. Meanwhile, it is also hard, and possibly harmful, to replace these semantics with each other. For example, replacing axiomatic semantics with operational semantics will introduce extra implementation details to the modelling stage and significantly limit the design space.

In this research, we present **SynQ** (publicly available in [5]) an embedded DSL (EDSL) targeting **Syn**chronous system design with **Q**uantitative types. SynQ is based on a component-based design framework and, by design, facilitates semantic coherency by leveraging the quantitative type theory (QTT) [6] and language embedding according to [1]. By synchronous systems, we mean embedded control and signal processing systems obeying the *perfect synchrony hypothesis* [7].

Semantic coherency is assured by the design and implementation of SynQ as follows:

- the carefully selected meta-language (the language in which the DSL is embedded) Idris2 [8] in which stages in design processes can be seamlessly conducted by benefiting from the Curry-Howard correspondence and QTT; and
- a systematic embedding of the DSL by the tagless-final approach [9], which allows us to fully leverage the meta-language Idris2 and to employ a unified approach to model and specify DSLs that separate concerns in a design process.

In the meantime, SynQ is also a practical language which allows generating *executable software* and *synthesisable Verilog HDL code* from synchronous systems modelled in it.

II. Symantics of SynQ

In contrast to a conventional (deep) embedding that starts by specifying the object-language's *syntax* as a type in the meta-language, the tagless-final approach embeds a language by specifying the language's constructors as a set of polymorphic type signatures in the meta-language. This set of signatures is often referred to as **symantics** in the literature because it characterises both **sy**ntax by typing rules and se**mantics** by the limited implementation space of these signatures. In the implementation of SynQ, these signatures are organised as *interfaces*, similar to Haskell's type class. Aligned with component-based design methodologies [1], SynQ is characterised by *four interfaces* that correspond to *glue and*

This research was partially funded by the Sweden's Innovation Agency (Vinnova) via the Advanced and innovative digitalization project 2021-02484 EARLY BIRD – Seamless System Design from Concept Phase to Implementation.

978-3-9826741-0-0/DATE25/© 2025 EDAA

atomic components of stateless/combinational systems and stateful/sequential systems, respectively.

Below is a glue component `abst` for sequential systems defined in SynQ, which sends *a function in the meta-language* from a combinational component to *a sequential component in SynQ*.

```
abst: {auto aIsSig: Sig a} -> {auto bIsSig: Sig b}
   -> {auto sIsState: St s}
   -> (1 _: comb () a -> seq s () b) -> seq s a b
```

In this signature, `()`, `a`, `b` and `s` are type variables denoting the type of components input/output and state, respectively. These type variables are restricted by GADT-based predicates `Sig` and `St` so that components are implementable.

Because of QTT, very fine-grained system specifications are possible. For instance, the type signature of a unit delay component can be given as follows:

```
dly: {n:_} -> {comb:_} -> {seq:_} -> (Seq comb seq)
  => (1 reg: Reg comb seq)
  -> seq (!* BitVec n) (BitVec n) (BitVec n)
```

in which the 1 in `(1 reg: Reg comb seq)` guarantees that there will be *exactly one occurrence* of `reg` in its implementation. The usage of `reg` is, in essence, an extra-functional property for the usage of resources. Hence, the ability to specify such a property and use it to constrain the construction of the corresponding model indicates the *semantic coherency* facilitated by SynQ between the specification and modelling stage.

III. Interpertations of SynQ

An interpretation, or an interpreter, of SynQ is a *consistent implementation* of all interfaces characterising SynQ. Since all interpreters are implemented in the meta-language Idris2, semantic coherency is facilitated. Currently, three interpreters are implemented. These interpreters map components modelled in SynQ to Idris2 functions, typed netlists, and another DSL embedded in Idris2 by the tagless-final approach. They respectively indicate how system models can be interpreted to their functional behaviour (denotational semantics), implementations, and corresponding models at lower/higher abstraction levels. These interpreters evidently illustrate semantic coherency enabled by SynQ at the modelling and implementation stages.

IV. Case Study

A case study available in [5] has been made to illustrate further the capability of and the design process enabled by SynQ. This case study is adapted from the case study presented in [10] in which the RV32-I ISA is modelled, simulated, verified and implemented. The case study indicates that, to a great extent, these design steps can be hosted by SynQ and Idris2. Exceptions include co-simulation of the model's functional behaviour with external components/environments and implementation steps *after* generating the Verilog HDL code. Compared to the case study in [10] in which design steps are manually executed, the SynQ version is more automatic as more design steps are defined and implemented as functions in Idris2. Two steps that have not been automated yet are the verification step, in which proofs still need to be conducted manually, and the transformation step(s), due to the limited interpreters that have been implemented so far.

V. Future Work

SynQ is currently under active development in [5], which means a large number of implementations remain, including interpreters applying transformations/optimisations and proofs of the correctness of interpreters. On the other hand, the tagless-final approach and its functional behaviour interpreter can be related to investigations on *logic frameworks* [11]. Further research in this direction will give us the chance to formally and systematically investigate system design processes in type theory.

References

[1] J. Sifakis, "System design automation: Challenges and limitations," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2093–2103, 2015.

[2] J. Sifakis, "Toward a system design science," in *From Programs to Systems. The Systems perspective in Computing: ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings*, Springer, 2014, pp. 225–234.

[3] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "ACSL: ANSI/ISO C specification," *URL https://frama-c. com/html/acsl. html*, 2021.

[4] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, IEEE, 2004, pp. 75–86.

[5] "The repository of SynQ," [Online]. Available: https://github.com/ruich95/SynQ.git.

[6] C. McBride, "I got plenty o'nuttin'," *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pp. 207–233, 2016.

[7] 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.

[8] E. Brady, "Idris 2: Quantitative type theory in practice," in *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[9] J. Carette, O. Kiselyov, and C.-c. Shan, "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages," *Journal of Functional Programming*, vol. 19, no. 5, pp. 509–543, 2009.

[10] R. Chen and I. Sander, "A quantitative type approach to formal component-based system design," in *2024 Forum on Specification & Design Languages (FDL)*, IEEE, 2024, pp. 1–10.

[11] R. Harper, F. Honsell, and G. Plotkin, "A framework for defining logics," *Journal of the ACM (JACM)*, vol. 40, no. 1, pp. 143–184, 1993.