Towards Automatic Hardware Synthesis from Formal Specification to Implementation

Fritjof Bornebusch[1], Christoph Lüth[1,2], Robert Wille[1,3], Rolf Drechsler[1,2]
[1]Cyber Physical Systems, DFKI GmbH, Bremen, Germany,
[2]Group of Computer Architecture, University of Bremen, Germany,
[3]Johannes Kepler University Linz, Austria
{fritjof.bornebusch,christoph.lueth}@dfki.de, robert.wille@jku.at, drechsler@uni-bremen.de

Abstract—In this work, we sketch an automated design flow for hardware synthesis based on a formal specification. Verification results are propagated from the FSL level through the proposed flow to generate an ESL model as well as an RTL implementation automatically. In contrast, the established design flow relies on manual implementations at the ESL and RTL level. The proposed design flow combines proof assistants with functional hardware description languages. This combination decreases the implementation effort significantly and the generation of testbenches is no longer needed. We illustrate our design flow by specifying and synthesizing a set of benchmarks that contain sequential and combinational hardware designs. We compare them with implementations required by the established hardware design flow.

1 INTRODUCTION

Nowadays, circuits are in almost every part of our lives and getting more complex over time. With the increasing complexity, the number of potential failures increases as well. To reduce potential failures in circuit designs, the increasing complexity must be considered from the beginning. To address this, iterative improvements have been proposed, e.g. the consideration of hardware designs at a higher abstraction level. The established design flow specifies a hardware design at the *Formal Specification Level* (FSL) [12] in SysML/OCL [22, 23] which can later be verified [11, 24]. After the hardware design was specified, a SystemC [17, 27] model at the *Electronic System Level* (ESL) [21] is implemented. As the model is implemented manually, testbenches at this level ensure that the model corresponds to the specification [28, 29]. Finally, an implementation at the *Register Transfer Level* (RTL) is implemented for synthesizing the desired hardware design. Testbench generation ensures that the implementation corresponds to the model [8].

Looking at this design flow in terms of its implementation effort, it can be seen that, thus far, almost all implementation steps between the individual abstraction levels have to be conducted manually. More precisely, while a structure for a SystemC model can indeed be generated automatically from a SysML class diagram, its behavior (e.g. described by OCL constraints) have to be implemented manually. The final RTL implementation has to be implemented manually as well, because of the limitations of SystemCs synthesizeable subset [13, 18, 26]. This makes the implementation effort of the established design flow time-consuming. Furthermore, due to the manual translation steps, the ESL model as well as the RTL implementation rely on the quality of the generated testbenches (and any verification results obtained in higher abstraction levels usually cannot be transferred to lower abstraction levels). Consequently, similar design tasks are frequently repeated at different abstraction levels. That is, the basic problem of the established design flow is the lack of an automatic translation process between the individual levels. To address this problem, proof assistant based design flows emerged [4, 6, 10, 14]. A formal specification of a hardware design is specified and verified by the proof assistant and translated into an RTL implementation automatically afterwards. Because of the automatic propagation of verification results, this requires no testbenches to be generated at the RTL level. However, following this flow, the hardware design is specified in a restricted *Domain-Specific Language* (DSL). Proof assistants provide a specification language to describe programs which is used to embed such a DSL. As the DSL is focused on a certain hardware design, these approaches cannot be treated as an alternative to the established design flow as it allows the specification and synthesis of arbitrary hardware designs.

In this work, we propose a new design flow which combines the proof assistant based design flow with functional hardware description languages (HDL) [2, 5]. Proof assistants like Coq [4] provide the extraction of formal specifications into executable code [20] which we extended to extract a CλaSH model [2] which again generates an RTL implementation. Both translation steps are automatically, i.e. no testbenches need to be generated at the ESL and RTL level. The combination of proof assistants and functional HDLs provides a design flow that does not require a restricted DSL and eliminates the problems of the established design flow, as discussed above. We present our new design flow as follows: First, we motivate and discuss the established design flow in detail. In Section 3, we evaluate existing approaches with respect to the lack of automation of the established design flow and describe the proposed design flow in detail. Section 4 describes the implementation while Section 5 evaluates the new design flow and shows the obtained results. Finally, Section 6 summarizes and concludes this work.

2 MOTIVATION

In this section, we briefly review the established design flow, thus far, when aiming to realize a given formal specification as a final RTL implementation. Based on that, the main problems are discussed which provide the motivation of this work. A running example is introduced in this section which is used to illustrate the established design flow as well as the design flow proposed later in this paper.

2.1 The Established Hardware Design Flow

Formal specifications, such as SysML/OCL [22, 23], provide an abstract description of an arbitrary hardware design to be realized. These descriptions allow for formal verification [11, 24] and provide a proper starting point for the design flow as sketched in Figure 1.

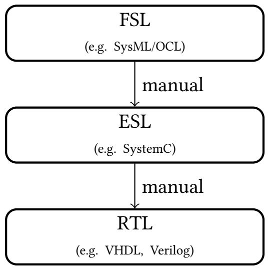

Figure 1: The established hardware design flow.

A formal specification is provided in modeling languages such as the *Systems Modeling Language* (SysML) which describe the structure of the desired hardware design, while constraints in the *Object Constraint Language* (OCL) describe the behavior. Afterwards, a corresponding SystemC [17] model is to be realized. Code skeletons can straightforwardly and automatically be derived from a SysML class diagram. The functional behavior of the resulting SystemC model (as, e.g., described through OCL constraints) has to be re-implemented manually since no executable SystemC code can be derived automatically from these constraints. Since manual implementations are error-prone, the generation of testbenches ensure that the model corresponds to the specification [28, 29]. Thus, the model relies significantly on the quality of these testbenches. The resulting manual implementation of the SystemC model as well as the generation of good quality testbenches is obviously a time-consuming process. To synthesize a hardware design the final RTL implementation, e.g. in VHDL [1], has to be implemented. This step is also manual as SystemC does not support the synthesis of an arbitrary hardware design, but of a restricted subset [13, 18, 26]. As the SystemC model, the RTL implementation also relies on the quality of the generated testbenches to ensure the implementation corresponds to the model [8]. The resulting manual implementation and the generation of good quality testbenches makes this step time-consuming as well.

2.1.1 Specification of a Hardware Design in SysML/OCL. In this work, we consider a traffic light controller as running example which controls the lights for *trams*, *cars*, and *pedestrians*. Figure 2 shows a SysML specification of the system and which is specified as follows:

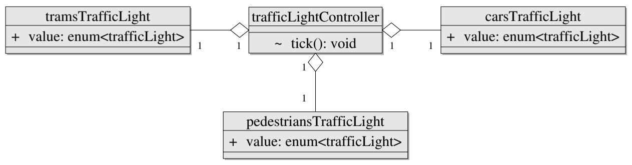

Figure 2: SysML specification of the traffic light example.

- *tick*: This function describes a finite state machine (FSM) which iterates over the individual states and sets the lights based on the current state. The different traffic lights are encoded as states of the FSM.
- *value*: This attribute of each traffic light contains the information how the individual lights are switched off and on.

```
context trafficLightController
  inv pedestriansTrafficLightGreen:
    self.pedestriansTrafficLight.value = trafficLight::Green implies
    (self.tramsTrafficLight.value <> trafficLight::Green and
    self.carsTrafficLight.value <> trafficLight::Green)
```

Listing 1: Safety property for the *trafficLightController* as OCL constraint.

In addition to the SysML description, OCL constraints as, e.g., the one shown in Listing 1 are implemented to specify a desired behavior. For example, the invariant in Listing 1 specifies that if the *pedestrians* traffic light is green, it is not green for the *cars* and the *trams*.

2.1.2 Implementation of the SystemC Model. In contrast, the precise implementations of functions such as *tick* (shown in Listing 2) have to be provided manually. As described above, testbenches ensure the correct behavior. After passing these testbenches, the final RTL implementation, e.g. in VHDL [1], is to be implemented which is also covered by testbenches.

```
void tick(TrafficLight *states) {
  Car cars = states->cars; Pedestrians pedestrians = states->pedestrians;
  Tram trams = states->trams;
  if (cars == Red && pedestrians == Red && trams == Red) {
    states->cars = Red;  states->pedestrians = Red; states->trams = Green;
  } else if (cars == Red && pedestrians == Red && trams == Green) {
    states->cars = RedYellow; states->pedestrians = Red; states->trams = Red;
  } else  if (cars == RedYellow && pedestrians == Red && tram == Red) {
    states->cars = Green; states->pedestrians = Red; states->trams = Red;
  } else  if (cars == Green && pedestrians == Red && tram == Red) {
    states->cars = Yellow; states->pedestrians = Red; states->trams = Red;
  } else  if (car == Yellow && pedestrians == Red && tram == Red) {
    states->cars = Red; states->pedestrians = Green; states->trams = Red;
  } else {
    states->cars = Red; states->pedestrians = Red; states->trams = Red;
  }
}
```

Listing 2: Traffic light state machine implementation of the SystemC model.

2.2 Problems of the Established Design Flow

The basic problem of the established design flow are the manual and time-consuming translation steps between the individual levels. As a result, the SystemC model and the RTL implementation rely on the quality of the generated testbenches as no verification results can be propagated through the design flow automatically. Using SysML/OCL at the FSL level and SystemC at the ESL level, these problems cannot be eliminated since OCL constraints cannot automatically be translated into executable SystemC code and an arbitrary SystemC model is not synthesizeable. A design flow that eliminates the problems of the established one must firstly allow the specification and verification of arbitrary hardware designs and secondly propagate verification results from the FSL level to the RTL level automatically.

3 GENERAL IDEA

In this work, we propose a new design flow which addresses the problems of the design flow discussed above. To this end, we are utilizing alternative design flows which already have been evaluated in the context of hardware verification and synthesis. These design flows already provide partial solutions to the problems discussed above. After reviewing those flows, we introduce a new design flow that combines the best of these and, by this, eliminates the problems of the established design flow.

3.1 Proof Assistant Based Hardware Design Flow

To address the lack of automation between the FSL and ESL level of the established design flow, we take a look at the verification of programs by proof assistants (so-called interactive theorem provers) [4, 6, 16]. Proof assistants are utilized for specification of programs and for verification of properties about the program's behavior. They specify a program as a collection of recursive function definitions and data types. The specification language is a higher-order logic and a property ϕ is proven if and only if ϕ is derivable in this logic. This logic is indeed too expressive for automated theorem proving and the proof assistant has to be guided manually through the proof [16]. For the automatic generation of executable code (certified programming), proof assistants, such as Coq [4], support the automatic extraction from a formal specification into a functional programming language, such as Haskell or Ocaml. A functional language is embedded into the specification language of the proof assistant which enables the extraction of proofs and functions into executable code by a straightforward syntactical substitution [20].

There is already research in the area of combining proof assistants with hardware design verification and synthesis [10, 14]. Figure 3 sketches their general design flow. A *Hardware Domain-Specific-Language* (Hardware DSL) is embedded in the specification language of the proof assistant. The hardware design is specified as a collection of functions and data types in the DSL. Combinational circuits are described as recursive function definitions, while sequential ones are a composition of such functions in combination with a finite state machine (e.g. a Mealy machine). This machine describes the clock as a transition from one state to the next. The Hardware DSL only describes the functional behavior of the design, without the consideration of dedicated hardware properties, e.g. parallelism. This enables the automatic analysis of a specified behavior and the generation of RTL implementations [3, 15].

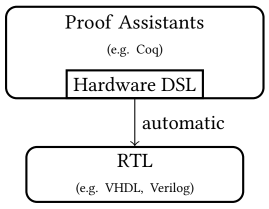

Figure 3: Hardware design flow using proof assistants.

At first view, this design flow addresses the problem of the established design flow, described in Section 2.2. The Hardware DSLs are indeed focused on dedicated hardware designs, e.g. multi-core CPUs [10] or low-level circuit designs [14]. The specification language already provided by the proof assistant is restricted and the DSL results in a limited expressivity of hardware designs: The Kami project [10] is not designed for describing combinational circuits while the PI-Ware project [14] is focused on low-level circuits, e.g. multiplexer or parallel prefix sum. To eliminate the problem of the restricted expressivity of hardware designs the DSL could, of course, be extended which is obviously time-consuming. The ability of proof assistants to extract executable code, however, leads to the following question: *Can arbitrary hardware designs be specified by using the specification language of proof assistants and exploit their existing extraction mechanism?*

3.2 Functional Hardware Description Languages

To address the lack of automation between the ESL and RTL level of the established design flow, we consider functional hardware description languages (functional HDLs). The idea of describing hardware designs using functional languages started back in the 1980s [25]. During the last 20 years functional hardware description languages became more popular and more implementations emerge [2, 5]. Analogous to the proof assistant based design flow, described above, combinational circuits are described as recursive functions and data types while sequential ones are a composition of such functions combined with a finite state machine. The unique representation of such hardware design models and the structured communication between their components, ensured by the type system, enables the automatic analysis and synthesis into an RTL implementation. Figure 4 sketches the synthesizing of hardware designs by functional HDLs.

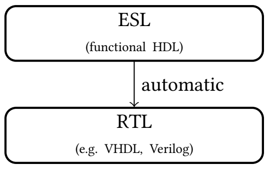

Figure 4: Hardware design flow using functional hardware description languages.

Simply replacing SystemC [17] by functional HDLs does not solve the problem of the established design flow, discussed in Section 2, since it is not possible to automatically generate an executable functional model from an arbitrary SysML/OCL [22, 23] specification. As functional HDLs enables the modeling of arbitrary hardware designs, the following question emerges: *Can proof assistants be combined with functional HDLs for automatic hardware synthesis?*

3.3 Proposed Hardware Design Flow

To eliminate the manual and time-consuming translation steps of the established hardware design flow, we merge the proof assistant based design flow with functional hardware description languages, as reviewed above. At the FSL level we use the proof assistant Coq [4, 9]. This proof assistant provides a program specification language (Gallina) which is based on an expressive formal language called the *Calculus of Inductive Constructions* (CiC) [4]. CiC combines higher-order logic with a richly-typed inductive design. A separate *tactic* language provides the implementation of user defined proof methods. As described in Section 3.1, Coq provides the extraction from a formal specification into executable code. To generate an ESL model we extended the extraction mechanism of Coq by the functional HDL CλaSH [2]. In comparison with embedded functional HDLs, like Lava [5], CλaSH borrows its syntax

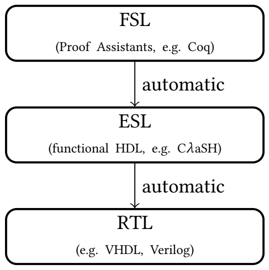

Figure 5: Proposed hardware design flow which combines proof assistants with functional hardware description languages.

```
 1  Definition tick (s : state) : state
 2  Definition transitionFunction (data:(state*Unsigned32.int)) (dummy:bool) :=
 3    match data with
 4      | (state, counter) => let state' := tick state in
 5        if counter = (clockFrequency * delay)
 6          then ((state', unsigned32_zero), state')
 7          else ((state, increment counter), state)
 8    end.
 9  Definition topEntity := mealy transitionFunction
10    (State carsRed pedestriansRed tramsRed, unsigned32_zero).
```

Listing 3: Coq specification of the *trafficLightController*.

```
Theorem pedestriansTrafficLightGreen:
  forall s : state, forall cars : trafficLightCar,
  forall pedestrians : trafficLightWalker,
  forall trams : trafficLightTram,
  (State cars pedestrians trams = tick s /\ pedestrians = pedestriansGreen) ->
  (cars <> carsGreen /\ trams <> tramsGreen).
```

Listing 4: Specification of the safety property in Coq.

and semantics from Haskell. This gives us access to all of Haskell's choice elements, like *case*-expressions and pattern matching [2]. Combinational as well as synchronous sequential circuits, either as a *Mealy* or a *Moore* machine, can be modeled in CλaSH. CλaSH itself supports the generation of different RTL implementations, e.g. VHDL, Verilog or SystemVerilog, from a model. Figure 5 sketches our proposed design flow.

By combining proof assistants with functional hardware description languages, we propose a new design flow that does not require manual translation steps as the established one, described in Section 2.1. The automatic translation steps reduce the implementation effort significantly what accelerates the entire hardware design process. Only a specification at the FSL level is needed while the ESL model and the RTL implementation are generated automatically. The proposed design flow does also not rely on testbench generation at the ESL and RTL level as verification results from the FSL level are propagated through the entire flow automatically.

4 IMPLEMENTATION OF THE PROPOSED HARDWARE DESIGN FLOW

To show how our proposed design flow is implemented, we consider the *trafficLightController* example, described in Section 2. This section shows the specification in Coq as well as the generated code in CλaSH to give an overview of how hardware designs are specified and verified in the proposed design flow. The same transitions of the state machine, as shown in Listing 2, are used. For this reason, we only show its function definition rather than its implementation in the following sections.

4.1 Specification and Verification of Hardware Designs in Coq

Low-level hardware description languages, such as VHDL or Verilog, use fixed size bit vectors for describing the in- and outputs of a circuit. To implement such bit vectors functional, dependent types are used [7]. The CompCert [19] compiler provides such data types as a library for Coq which we utilized to describe *signed* and *unsigned* types. This implementation of dependent types allows us to specify the communication between the individual components of a hardware design in the same way as CλaSH does. Line 2 of Listing 3 shows a dependent type for an *unsigned* 32 bit vector. Sequential circuits in functional HDLs are modeled as state machines. The function definition for CλaSHs Mealy machine in Coq looks like this: *mealy (f: S ⟶ I ⟶ (S×O)) (s: S) (l: list(I)) : list(O)*. An input list `list(I)` is mapped on an output list `list(O)` by a transition function *f*. This function takes two parameters: a state of the type `S` and an input of the type `I` which calculates the output `(S×O)`. This tuple contains the new state and the output for the Mealy machine. The types `S`, `I` and `O` are inferred at compile time, as seen in Line 10 of Listing 3 which makes this implementation of the state machine generic. The state machines initial state is a tuple that contains the initial state for the traffic lights *State carsRed pedestriansRed tramsRed* and an initial counter value *unsigned32_zero*. The *clockFrequency* and the *delay* are constant functions. The *transitionFunction* definition implements the transition function for the Mealy machine. Our specification of the *trafficLightController* is not related to an input list, as the *data* tuple contains all information. Our definition of the transition function (*f*), however, requires an input, but the input is ignored in this case (*dummy*). The *tick* function, as seen in Line 1 of Listing 3, transforms a given state (*s*) into a new state. The state transitions are analog to the ones described in Listing 2. The inductive type *state* implements the different traffic lights for the *cars*, *pedestrians* and *trams*.

After specifying the hardware design we need to verify that this specification corresponds to the safety property described in Listing 1. Listing 4 shows this safety property in Coqs specification language. It says: for all states we call *tick* for and the resulting traffic light for the pedestrians is green, then it is not green for the cars and for the trams. To specify arbitrary hardware designs in Coq, all we need is a specification of the state machine (*Mealy* or *Moore*) in combination with dependent types to describe the desired design and to extract it into a valid CλaSH model automatically.

4.2 Automatic Generation of CλaSH Models

To generate a CλaSH [2] model at the ESL level, we extended Coq's extraction mechanism to generate such a model from an arbitrary specification [20]. This mechanism supports two ways to extract a specification. First, it extracts everything that is related to the function that should be extracted, i.e. other called functions or used data types. Second, it replaces functions or data types by those of the target language. In our case Coqs *Unsigned32.int* type was replaced by CλaSHs *Unsigned 32* type, as seen in Listing 5 which shows the automatic generated CλaSH model of the above specification. These replacements have to be configured ones by hand. CλaSH

```
tick :: State0 -> State0
transitionFunction :: ((,) State0 (Unsigned 32)) -> CLaSH.Prelude.Bool -> (,)
                        ((,) State0 (Unsigned 32)) State0
transitionFunction data0 _ =
  case data0 of {
   (,) state counter -> let {state' = tick state} in
    case (CLaSH.Prelude.==) counter ((CLaSH.Prelude.*) (50*(10^6)) 15) of {
     CLaSH.Prelude.True -> (,) ((,) state' 0) state';
     CLaSH.Prelude.False -> (,) ((,) state (increment counter)) state}}
topEntity :: CLaSH.Prelude.Bool -> (,) ((,) State0 (Unsigned 32)) State0
topEntity = mealy transitionFunction ((,) (State carsRed pedestriansRed tramsRed
       ) 0)
```

Listing 5: Automatic generated CλaSH model of Coq's *trafficLightController* specification.

in particular relies on a special notation for dependent types. For this reason, we cannot extract the types from the CompCert library directly. The same applies to the Mealy machine as we need CλaSHs native implementation to execute and synthesize the model. The goal of such replacements is to use the semantic equivalent types of the target language and only extract the functional behavior.

From the functional model, we are able to automatically synthesize an implementation. CλaSH supports different low-level HDLs, such as VHDL, Verilog and SystemVerilog and its type system ensures that every model is synthesizeable.

5 EVALUATION

In order to evaluate the design flow proposed in this paper, we specified a set of benchmarks which describes combinational as well as sequential circuits. To show the reduction of the implementation effort of the proposed design flow compared with the established one, we applied both flows to our set of benchmarks. The final RTL implementations of both flows are synthesized on an FPGA to compare the consumed space and the maximum clock frequency.

5.1 Benchmarks

The following set of benchmarks have been implemented for evaluation purposes:

- *MAC:* The combinational *multiply-and-accumulate* (MAC) circuit.
- *Chaser Light:* This circuit implements a sequential chaser light that iterates over the LEDs of the FPGA we used for evaluation purposes.
- *Traffic Light:* This circuit implements an extended version of the sequential traffic light controller described in Section 4.
- *Airbag Controller:* This circuit represents a sequential airbag controller that links sensors with their corresponding airbags and triggers them independently.
- *Ticket Machine:* This circuit is a sequential ticket machine, e.g. for a tram. It offers different kinds of tickets, e.g. for children, groups or saving prices.

5.2 Implementation Effort

In this section, we compare the implementation effort of the established design flow, described in Section 2.1 with the proposed one, described in Section 3.3. To do this, we implemented every benchmark in the languages both design flows rely on. The implementation effort of the individual implementations ranges from $0.5h$ (MAC) to $8h$ (Ticket Machine)

5.2.1 *Established Hardware Design Flow.*

The benchmarks have been implemented in the following languages for the established design flow: SysML/OCL [22, 23], SystemC [17] and VHDL [1] which results in three manual implementations. This manual implementation effort can be applied to larger hardware designs, since it cannot be eliminated as described in Section 2.1. The verification of the SysML/OCL specification as well as the generation of testbenches for the SystemC model and the VHDL implementation further increase the implementation effort.

5.2.2 *Proposed Hardware Design Flow.*

The benchmarks for the proposed design flow have to be implemented only in Coq [4], since the CλaSH [2] model as well as the final VHDL [1] implementation have been generated automatically. The implementation effort of the Coq specification has been the same as for the individual implementations required by the established design flow. Even though the verification of the specification in Coq would increase the implementation effort, the reduction of this effort remains as the hardware design has to be specified only ones instead of three times as discussed above.

The evaluation of the implementation effort of the established design flow and the proposed one shows that the one proposed in this work decreases this effort significantly which accelerates the hardware design process. While the established one requires three manual implementations, on the FSL, the ESL and the RTL level, the proposed one only requires one at the FSL level. From this FSL specification the model at the ESL level and the final implementation at the RTL level are generated automatically.

5.3 Quality of the Results

Doing design tasks automatically usually leads to an automatization overhead, i.e. the resulting designs are often not as efficient/compact as when the design has been determined manually. In order to evaluate that, the final VHDL [1] implementations of our benchmark set for the established as well as the proposed design flow have been synthesized on an FPGA (terasIC DE10-Lite), to measure the consumed space (LUTs/Register) and the maximum clock frequency (F_{max}). For synthesizing these implementations we used the Intel® Quartus Prime Software Suite. The results are provided in Table 1. The table shows that the proposed design flow is practical as all VHDL implementations have been synthesized on the FPGA. These implementations might consume more space, which is due to the fact that engineers can optimize more efficient by hand than tools. Looking at the maximum clock frequency of the *Traffic Light* and the *Ticket Machine* it shows that for some implementations the difference between those generated by the proposed design flow and by those of the established one is marginal. This shows that the design flow, proposed in this paper, can be employed in practice.

Our results show that the design flow proposed in this work eliminates the problems of the established one, described in Section 2.2. Specified and verified hardware designs generate an RTL implementation by propagating verification results automatically. Testbenches at the ESL and RTL level are no longer needed which makes the proposed design flow unsusceptible for implementation errors at these levels as no manual intervention is needed.

Table 1: Evaluation by comparing the consumed space and the maximum clock frequency.

	established flow		**proposed flow**	
Circuit	LUTs / Register	F_{max}	LUTs / Register	F_{max}
MAC	61 / 0	-	61 / 0	-
Chaser Light	78 / 60	252.02 MHz	121 / 69	191.46 MHz
Traffic Light	163 / 45	231.75 MHz	759 / 36	230.79 MHz
Airbag Controller	128 / 110	240.96 MHz	299 / 125	185.98 MHz
Ticket Machine	747 / 196	74.48 MHz	1141 / 146	66.36 MHz

Note that the *MAC* circuit is combinational. For this reason, it does neither consume registers nor has a maximum clock frequency.

6 CONCLUSION

In this work, we proposed a design flow to automatically generate a hardware design implementation at the RTL level from a formal and verified specification. The established flow which uses SysML/OCL [22, 23] and SystemC [17] cannot provide such an automatic translation process as it is not possible to generate an arbitrary executable SystemC model from a SysML/OCL specification and synthesize it afterwards into an RTL implementation automatically. This motivates our approach, and we address this problem by combining the proof assistant Coq [4] with the functional hardware description languages CλaSH [2]. We extended the automatic code extraction mechanism of Coq to automatically generate an executable CλaSH model from a non-executable but provable specification. CλaSH describes combinational as well as synchronous sequential circuits (either as a *Mealy* or a *Moore* machine). The model is synthesized into an RTL implementation automatically. Thus, the proposed design flow reduces the implementation effort compared with the established one. Since we can propagate the verification results from the FSL level down to the RTL level, we are not required to generate testbenches. This accelerates the hardware design process significantly. A set of different hardware designs were used as benchmarks to evaluate the proposed design flow and have been synthesized on an FPGA. This shows the applicability of our flow and introduced it as an alternative to the established one which opens the door for further research in this area.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SELFIE under grant no. 01IW16001 as well as the LIT Secure and Correct System Lab funded by the State of Upper Austria.

REFERENCES

[1] Ashenden, P. J. *The designer's guide to VHDL, 2nd Edition.* The Morgan Kaufmann series in systems on silicon. Kaufmann, 2002.

[2] Baaij, C., Kooijman, M., Kuper, J., Boeijink, A., and Gerards, M. Cλash: Structural descriptions of synchronous hardware using haskell. In *Euromicro Conference on Digital System Design (DSD)* (2010), pp. 714–721.

[3] Baaij, C., and Kuper, J. Using rewriting to synthesize functional languages to digital circuits. In *Trends in Functional Programming (TFP)* (2013), pp. 17–33.

[4] Bertot, Y., and Castéran, P. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[5] Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. Lava: Hardware design in haskell. In *The ACM SIGPLAN International Conference on Functional Programming(ICFP)* (1998), pp. 174–184.

[6] Bove, A., Dybjer, P., and Norell, U. A brief overview of agda - A functional language with dependent types. In *Theorem Proving in Higher Order Logics (TOPHOLS)* (2009), pp. 73–78.

[7] Brady, E., McKinna, J., and Hammond, K. Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types. In *Trends in Functional Programming (TFP)* (2007), pp. 159–176.

[8] Chen, M., Mishra, P., and Kalita, D. Automatic RTL test generation from systemc TLM specifications. *ACM Trans. on Embedded Computing Systems 11*, 2 (2012), 38:1–38:25.

[9] Chlipala, A. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2013.

[10] Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., and Arvind. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages (PACMPL) 1*, ICFP (2017), 24:1–24:30.

[11] Debbabi, M., Hassaïne, F., Jarraya, Y., Soeanu, A., and Alawneh, L. *Verification and Validation in Systems Engineering - Assessing UML / SysML Design Models.* Springer, 2010.

[12] Drechsler, R., Soeken, M., and Wille, R. Formal Specification Level. In *Forum on Specification and Design Languages (FDL)* (2012), pp. 37–52.

[13] Falk, J., Haubelt, C., and Teich, J. Efficient representation and simulation of model-based designs. In *Forum on Specification and Design Languages (FDL)* (2006), pp. 129–135.

[14] Flor, J. P. P., Swierstra, W., and Sijsling, Y. Pi-ware: Hardware description and verification in agda. In *International Conference on Types for Proofs and Programs (TYPES)* (2015), pp. 9:1–9:27.

[15] Gammie, P. Synchronous digital circuits as functional programs. *ACM, Comp. Surveys 46*, 2 (2013), 21:1–21:27.

[16] Geuvers, H. Proof assistants : history, ideas and future. *Sadhana 34*, 1 (2009), 3–25.

[17] Grotker, T. *System Design with SystemC.* 2002.

[18] Inc, A. S. I. Systemc synthesizable subset version 1.4.7. https://www.accellera.org/images/downloads/standards/systemc/SystemC_Synthesis_Subset_1_4_7.pdf.

[19] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. Compcert – a formally verified optimizing compiler. In *Embedded Real Time Software and Systems (ERTS)* (2016), SEE.

[20] Letouzey, P. Extraction in coq: An overview. In *Computability in Europe (CIE)* (2008), pp. 359–369.

[21] Martin, G., Bailey, B., and Piziali, A. *ESL Design and Verification: A Prescription for Electronic System Level Methodology.* Morgan Kaufmann Publishers Inc., 2007.

[22] Object Management Group. Object Constraint Language. Tech. Rep. formal/2014-02-03, OMG, 2012.

[23] Object Management Group. OMG Systems Modeling Language (OMG SysML). Tech. Rep. formal/2015-06-04, OMG, 2015.

[24] Przigoda, N., Wille, R., and Drechsler, R. Analyzing inconsistencies in UML/OCL models. *Journal of Circuits, Systems, and Computers 25*, 3 (2016).

[25] Sheeran, M. Designing regular array architectures using higher order functions. In *Functional Programming Languages and Computer Architecture (FPCA)* (1985), pp. 220–237.

[26] Stoppe, J., Wille, R., and Drechsler, R. Data extraction from systemc designs using debug symbols and the systemc API. In *isvlsi* (2013), pp. 26–31.

[27] Takach, A. High-level synthesis: Status, trends, and future directions. *IEEE Design & Test 33*, 3 (2016), 116–124.

[28] Weissnegger, R., Pistauer, M., Kreiner, C., Schuss, M., Römer, K., and Steger, C. Automatic testbench generation for simulation-based verification of safety-critical systems in UML. In *International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS)* (2016), pp. 70–75.

[29] Wille, R., Grosse, D., Haedicke, F., and Drechsler, R. Smt-based stimuli generation in the systemc verification library. In *Forum on Specification and Design Languages (FDL)* (2009), pp. 1–6.