Integer Overflow Detection in Hardware Designs at the Specification Level

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

Keywords:
Hardware Designs, Integer Overflows, Proof Assistants, Functional HDLs, Hardware Synthesis

Abstract:
In this work, we present a hardware design approach that allows the detection of integer overflows by describing finite integer types at the specification level. In contrast to the established design flow that uses infinite integer types at the specification level. This causes a semantic gap between these infinite types and the finite integer types used at the model level. The proposed design approach uses dependent types in combination with proof assistants. The combination allows the arguing about the behavior of finite integer types that is used to detect integer overflows at the specification level. To achieve this, we utilized the CompCert integer library that describes finite data types as dependent types.

1 Introduction

Nowadays, circuits are in almost every part of our live and their complexity continues to increase. With the increasing complexity the number of potential errors increase as well. For this reason, the development process of hardware designs should consider the complexity from the beginning.

Hardware designs are described at different levels to address the complexity of such designs. First, the design is formally specified, e.g. in SysML/OCL (Drechsler et al. 2012; OMG 2014; OMG 2019; Weilkiens 2007), which enables the specification of properties that argue about the desired design (Brucker and Wolff 2006; Hilken et al. 2014). Afterwards the specification is translated into a SystemC model which is the de facto standard for high-level synthesis (HLS) (Arnout 2000; Takach 2016). This translation is manual as OCL constraints cannot be translated into executable SystemC code automatically. For the final synthesizing step, the model is manually translated into in a low-level implementation, e.g. VHDL, as SystemC does not support the synthesis of arbitrary hardware models, because of its restricted synthesizeable subset (Accellera 2016; Stoppe et al. 2013).

We consider this approach as the established hardware design approach in the rest of the paper. Looking at the established design approach in terms of its integer type implementation a *semantic gap* is revealed between the infinite types of the SysML/OCL specification and the finite types of the SystemC model. As a result, properties of the SysML/OCL specification might not hold in the SystemC model. One result of this *semantic gap* is an integer overflow that occurs when an integer operation is executed in the SystemC model (Cousot et al. 2005; Cuoq et al. 2012; Dietz et al. 2015). An overflow leads to unintended behavior. The missing tool support for detecting integer overflows automatically results in the detection of integer overflows in the SystemC model explicitly by the engineer.

To address the basic problem of the *semantic gap* of the established design approach, we propose an alternative hardware design approach that enables the description of finite integer types at the specification level. This enables the description of types that are *semantic equivalent* to those of the model level. As a result, prop-

erties that argue about such types also hold on the model level. To achieve this, we utilized the CompCert integer library (Leroy et al. 2016) to describe hardware designs by the proof assistant Coq (Bertot and Castéran 2004; Chlipala 2013). This specification can subsequently be extracted into an executable functional model automatically (Bornebusch et al. 2020).

We present our work as follows: First, we explain the established hardware design approach in detail by referring to a running example and discuss the problem between infinite and finite integer types of this approach while Section 3 discusses the related work. Section 4 proposes our approach and explains how integer overflows are detected in this flow and how the extraction from a specification to a model is implemented. Section 5 summarizes and concludes this work.

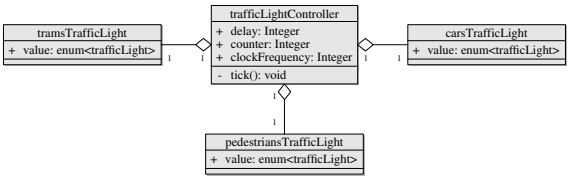

Figure 1: SysML class diagram.

```
1  context trafficLightController::tick()
2   pre: self.counter * self.delay <
3        self.counter * self.clockFrequency
4   post: self.counter = self.counter@pre +1
5
6  context trafficLightController::tick()
7   pre: self.counter * self.delay >=
8        self.counter * self.clockFrequency
9   post: self.counter = 1
10
11  inv: self.counter > 0
12  inv: self.delay > 0
13  inv: self.delay < self.clockFrequency
```

Listing 1: OCL constraints of the *tick* function.

2 Motivation

In this section, we briefly review the specification and the modeling of hardware designs in the established approach which uses SysML/OCL (OMG 2014; OMG 2019) at the specification level and SystemC (Arnout 2000; Takach 2016) at the model level. Based on that, the main problem of this approach is shown that describes why integer overflows occur during the translation from the specification to the model which motivates this work. A running example is introduced in this section to illustrate the established hardware design approach as well as the proposed approach.

2.1 The Established Approach

Considering the established hardware design approach, the design is first described as a SysML/OCL specification. This specification describes the structure of a hardware design in SysML while the behavior is described by OCL constraints. In this work, we consider a traffic light controller (inspired by (Przigoda et al. 2016)) as a running example:

Example 1. *Figure 1 shows the SysML class diagram of the traffic light controller. The controller consists of three different traffic lights: for the* trams, cars *and* pedestrians, *as seen in Figure 1.*

This traffic light controller iterates over predefined states which determines whether the individual traffic lights are switched on or off. The transition from one state to the next depends on a given delay. *If the* delay *is expired, the transition to the next state is triggered, e.g. from green to yellow for the cars. To allow the consideration of traffic situations, such as rush-hour, the* delay *can be changed at run-time. After describing the structure of the controller in SysML, the desired behavior is described by OCL constraints, as the ones seen in Listing 1. The* tick *function represents the clock and evaluates whether the traffic light state is triggered or not, depending on the* delay[1] . *The variable* clockFrequency *is constant and describes the frequency of the hardware the controller runs on.*

The conditions in Line 1 of Listing 1 ensures that the counter *variable of the controller is increased by one until the precondition no longer holds, i.e. the upper bound is reached. In this case the counter is reset to 1 as seen in Line 9 of Listing 1.*

After the behavior is specified in SysML/OCL, a SystemC model is implemented as shown in the next example. The transformation from a SysML/OCL specification to a SystemC model is manual. The SysML structure can indeed be translated to SystemC classes automatically, but there is no automatic process that translates OCL constraints into executable SystemC code.

Example 2. *Listing 2 shows the implementation of the* tick *function, described above, which implements the OCL constraints, seen in Listing 1.*

[1]Note that this work considers integer overflows in hardware designs. Therefore, the individual state transitions are not considered, as they do not cause an integer overflow.

```
sc_uint<32> counter, delay, clockFrequency;

void tick() {
 if (counter*delay < counter*clockFrequency)
  counter++;
 else
  counter = 1;
}
```

Listing 2: SystemC model of the *tick* function.

Like in the specification of the tick *function, the counter is increased until it reaches its upper bound, according to the specification, and is reset to 1 again. If this upper bound is not yet reached the counter is increased by one.*

2.2 Considered Problem

From the constraints in Listing 1, we can derive properties which hold for the specified system, as shown in the next example.

Example 3. *The safety property (stated as an invariant) in Listing 3 can be* derived *from the specified behavior of the* tick *function. This means that if we implement this function such that the constraints from Listing 1 hold, then the implementation will satisfy the safety property.*

```
context trafficLightController
 inv: self.counter * self.delay <=
      self.counter * self.clockFrequency
```

Listing 3: Safety property derived from the SysML/OCL specification.

However, in the SystemC implementation the safety property does not hold! To examine why, we consider the proof in more detail. To show the safety property, we need to show that, for each constraint of the operation *tick* in Listing 1, if the precondition, invariants and safety property hold in the pre-state and the postconditions holds in the post-state, then the safety property holds in the post state.

In the following, we use the notation x' to denote the value of the variable x in the post-state, and we elide the self prefix. We then have the following assumption:

$$\begin{aligned} &\mathsf{counter} * \mathsf{delay} < \mathsf{counter} * \mathsf{clockFrequency} \\ \wedge\ &\mathsf{counter} * \mathsf{delay} \leq \mathsf{counter} * \mathsf{clockFrequency} \\ \wedge\ &\mathsf{counter}' = \mathsf{counter} + 1 \end{aligned} \tag{1}$$

We now need to show the safety property in the post state[2]:

$$\begin{aligned} &\mathsf{counter}' * \mathsf{delay}' \leq \mathsf{counter}' * \mathsf{clockFrequency}' \\ \Longleftrightarrow\ &(\mathsf{counter} + 1) * \mathsf{delay} \leq \\ &\quad (\mathsf{counter} + 1) * \mathsf{clockFrequency} \\ \Longleftrightarrow\ &\mathsf{counter} * \mathsf{delay} + \mathsf{delay} \leq \\ &\quad \mathsf{counter} * \mathsf{clockFrequency} + \mathsf{clockFrequency} \end{aligned} \tag{2}$$

For $\mathbb{N}$ and $\mathbb{Z}$ (the SysML Integer type represents $\mathbb{Z}$) this follows from the assumption and invariants (line 13 in Listing 1) because of monotonicity of addition, $a \leq c \wedge b \leq d \implies a + b \leq c + d$, but it does *not* hold for integers of limited size precisely because monotonicity does not hold there (e.g. in the quotient ring $\mathbb{N}/32$). In other words, multiplication in $\mathbb{Z}$ is not semantic equivalent to the one in $\mathbb{N}/32$.

Example 4. *Consider again the OCL constraints from the SysML/OCL specification seen in Listing 1 and the resulting implementation of the SystemC model, seen in Listing 2. The implementation assumes that the multiplication operation applied in the model is defined the same as the multiplication applied in the specification. This assumption is reasonable, as they define apparently the same behavior. However, as described above this is not the case, as the specification defines infinite integer types while the model defines finite ones. This means the SystemC model violates the safety property in Listing 3, which holds for the specification. This bears a direct impact on the change of the transition time between the traffic lights implemented by the controller in the SystemC model. For instance, if the value is changed at run-time in a rush-hour situation, the resulting behavior of the* tick *function and by that of the entire state machine might be unintended which is a serious problem.*

The C++ standard describes two different behaviors of integer arithmetic (ISO/IEC 2017). Unsigned integer arithmetic might cause unintended behavior as seen in Listing 2, but does technically not overflow. The result is always performed modulo 2^n so it is never too big to be interpreted, i.e this type implements a wraparound behavior. Signed integer arithmetic, on the other hand, does not perform modulo 2^n so the result can be too big to be interpreted. Such an overflow causes undefined behavior as it is not specified by the standard how to proceed in this case, e.g. it may also wrap around, because of the 2's complement or trap on some platforms. Therefore, the signed integer arithmetic operations in C++ are partial and not total as in SysML. The term *integer overflow* often refers to both behaviors as they share the same basic problem (Cousot et al. 2005; Dietz et al. 2015). For this reason, we use

[2]Note that there is a tacit assumption that the values of variables do not change unless mentioned otherwise; here, we assume that $\mathsf{clockFrequency}' = \mathsf{clockFrequency}$ and that $\mathsf{delay}' = \mathsf{delay}$.

this term in the rest of the paper to address the problem discussed above.

The problem of the *semantic gap* between SysML's infinite types and SystemC's finite types motivates our work. In order to address this problem a semantic equivalent (finite) integer type is needed at the specification level as hardware designs rely on these types. Such a type allows the specification of a function that clearly distinguishes an integer overflow from the actual result of the integer operation by considering the lower and upper bounds of the finite integer type. In the next section, we evaluate the related work which, indeed, does not address the problem of the established hardware design approach properly. This leads to the design approach proposed in this work.

3 Related Work

In this section, we discuss the related work which shows that SysML/OCL at the specification level and SystemC at the model level are not suitable to detect integer overflows. To detect integer overflows in a SysML/OCL specification the implementation of semantic equivalent (*finite*) integer types to those of the SystemC model are required. However, this is not supported by the SysML standard (OMG 2019). Of course, constants could be introduced to artificially restrict the range of an *infinite* integer type by describing the lower and upper bounds. These bounds, however, are independent of the actual type, i.e. the one used in the SystemC model. The introduction of these bounds do not address the problem, discussed above, as if in the development phase of the model the actual type changes, e.g. from unsigned32 to unsigned31, such bounds invalidate the model which again trigger an integer overflow.

To detect integer overflows directly in the SystemC model, the overflow detection of programs in C++ has to be considered. The detection of such overflows in this language is quite challenging. The basic problem is that the low-level nature of C++ does not allow the detection of overflows reliably as bit manipulations are common in this language (Dietz et al. 2015). Furthermore, C++ has undefined behavior semantics for signed integer types which allow optimizations by the compiler (Dietz et al. 2015). C++ compiler are able to detect integer overflows if it is constant-expression evaluation, but there is no support for the automatic detection in general.

As a result, the automatic and reliable detection of arbitrary integer overflows is not supported, as it is not possible to distinguish a behavior intended by the engineer from unintended. As there is no support from the compiler, some static source-code analysis tools, such as

Astrée (Cousot et al. 2005), aims to prove the absence of run-time errors in C programs, like integer overflows, through *abstract interpretation* (Cousot 2012; Fähndrich and Logozzo 2010). Abstract interpretation is used to derive a computable abstract semantic interpretation from a behavior described in a programming language. This interpretation does not contain the actual values, but focuses on certain parts of the program execution. These parts determine the scope of the static analysis and what kind of errors are detected. Abstract interpretation reaches its limits when it comes to the analysis of loops, as they have an infinite number of paths in the abstract interpretation tree. As SystemC designs allow loops, the static analysis of integer overflows by abstract interpretation is not suitable to detect them in a hardware design, in general.

Frama-C is another source-code analyzing tool which relies on *C Intermediate Language* (CIL) (Necula et al. 2002) and supports annotations written in *ANSI/ISO C Specification Language* (ACSL) (Cuoq et al. 2012). It allows the application of different static analysis techniques which includes the deductive verification of annotated C programs by external automatic provers, e.g. Z3 (Cuoq et al. 2012). Considering the detection of integer overflows, Frama-C provides the *Runtime Error Annotation Generation* (RTE) plugin which includes the generation of annotations by syntactic constant folding in the form of assertions for integer overflows. The main purpose of RTE is seed these annotations into other plugins, e.g. for the generation of weakest-preconditions, with proof obligations. However, Astrée and Frama-C cover integer overflows in C programs while SystemC models are not supported. For this reason, they are not suitable to address the problem of the established hardware design flow properly.

As described above, the detection of integer overflows using SysML/OCL at the specification and SystemC at the model level is not suitable. At the specification level the integer type is *infinite* and at the model level the engineer needs to detect overflows pro-active and explicitly since there is little to no tool support. For this reason, the safety property described in Listing 1 cannot be implemented properly without the explicit

and pro-active consideration of integer overflows occurred in the model, by the engineer.

The considered problem, described in Section 2.2, in connection with the related work discussed in this section lead to the following question: *Can an alternative hardware design approach be developed that allows the verifiable detection of integer overflows at the specification level?*

4 Proposed Solution

This section reviews alternative methods which describe finite integer types at the specification level since this is the main problem of the established hardware design approach as described in Section 2.2. After reviewing these methods, we propose a hardware design approach that enables the formal specification as well as the verifiable detection of integer overflows in hardware designs.

4.1 Proof Assistants

An alternative approach to specify and subsequently verify an arbitrary behavior at the specification level are *proof assistants*, so-called interactive theorem provers (Bertot and Castéran 2004). Proof assistants formally specify the functional behavior of programs in a higher-order logic (specification language). This behavior is defined by total functions which allows the verification of properties. A property ϕ is proven if and only if ϕ is derivable in the logic of the proof assistant. As higher-order logic is too expressive for automated theorem proving, the proof assistant is guided interactively through the proof process by the engineer. Apart from the specification and verification of functional behavior, some proof assistants, like Coq, allows the extraction of this behavior into executable code. This way of program development is called certified programming (Chlipala 2013). It is achieved by embedding a functional language into the specification language of the proof assistant. This functional language enables the extraction of a specification into a functional programming language, e.g. Haskell or Ocaml, by syntactical substitution.

4.2 Dependent Types

To describe the limited size bit vectors of hardware designs for the in- and outputs of a circuit functionally, *dependent types* are used. Describing hardware designs using *dependent types* is not new and started back in the 90s (Brady et al. 2007; Hanna and Daeche 1992). A *dependent type* allows a type definition that relies on an additional value. For instance, the type A^n defines a vector of length n with components of type A. We say that A depends on n what makes A^n a *dependent type* and enables the definition of finite integer types, e.g. $Unsigned^{32}$. As proof assistants, like Coq, allow the description of *dependent types* by the user, this gives us the opportunity to describe *finite* integer types at the specification level. We utilized CompCert's integer library to describe finite *signed* and *unsigned* integer types in Coq (Leroy et al. 2016). This library describes these types as *dependent types* which allows the definition of arbitrary limited size types.

4.3 The Proposed Approach

To address the problem of the *semantic gap* between the specification and the model level, as described in Section 2.2, we propose a hardware design approach that utilizes the proof assistant Coq in connection with the usage of dependent types to detect integer overflows at the specification level.

In contrast to a specification described in SysML/OCL, the approach proposed in this work describes a hardware design specification in Coq's functional specification language.

4.3.1 Detecting Integer Overflows

The *unsigned integer multiplication* overflow in the SystemC model, seen in Listing 2 occurs when implementing the multiplication, because of the semantic gap between the *infinite* types of SysML and *finite* types of SystemC.

The basic problem behind the semantic gap is that the arithmetic operations (like multiplication) behave differently when we move to a finite type. That is, if the result of $a * b$ is larger than the maximum size, the value of $a * b$ in $\mathbb{N}$ and $\mathbb{N}/32$, or $\mathbb{Z}$ and $\mathbb{Z}/32$, no longer agree. We propose to make this distinction explicit by making the multiplication operation partial at the type level. This is modeled by a datatype *option* with two constructors, *None* and *Some(a)*, where *None* stands for undefined.

Listing 4 shows the definition of the explicitly partial multiplication function. It will return *None* whenever an overflow occurs, and *Some(a*b)* if not. Based on the bounds of the

```
type a option = None | Some of a
safe_mult : n ∈ ℕ
          ⇒ Unsigned^n
          → Unsigned^n
          → option(Unsigned^n)
safe_mult x y =
  if y ≠ 0 ∧ x > max_unsigned(x) / y)
    then None
    else Some(x*y)
```

Listing 4: Function definition for unsigned multiplication overflow detection.

```
Definition safe_mult (a b:Unsigned32.int)
: option Unsigned32.int :=
 if b =? 0%unsigned32
  then Some (a*b)
  else if a <=? (Unsigned32.max_unsigned / b)
        then Some (a*b)
        else None.

Definition tick
 (counter clockFrequency delay:Unsigned32.int)
: option Unsigned32.int :=
 match (safe_mult counter delay,
        safe_mult counter clockFrequency) with
  | (Some a, Some b) =>
   if a <? b else Some (counter +1%unsigned32)
              then Some (1%unsigned32)
  | _ => None
  end.
```

Listing 5: Function definitions in Coq.

finite integer type, a condition checks whether an overflow occurs. This is the case if $(x * y) > unsigned\_max(x)$, but we cannot check this directly (because of the overflow); hence, we check whether x is greater than the result of the maximum value of the data type ($unsigned\_max(x)$) divided by y. The maximum value depends on the dedicated unsigned integer type, i.e. $2^n - 1$, where $n \in \mathbb{N}$, is the number of bits used to represent the values of the type. To avoid a *division-by-zero* error, it is ensured that y is not equal to 0 (in that case, no overflow can occur). The *safe_mult* function essentially wraps the multiplication function for *Unsigned32* and provides an overflow-save replacement.

Example 5. *As described in Section 4.3, we utilized the CompCert integer library as it provides the description of* unsigned *and* signed *integer types of arbitrary sizes. In order to detect an integer overflow, a clear distinction is needed between the occurrence of the overflow and the result of the applied operation. Coq describes a specification by total functions which we used to define such a distinction. The* safe_mult *function, seen in Listing 5, implements the behavior of the function described in Listing 4.*

In order to clearly distinguish an overflow from the result of the multiplication, the safe_mult *function returns a value of Coq's built-in container type* option. *This container type has the same semantic behavior as the* option *type, defined in Listing 4. In contrast to the SysML/OCL specification, introduced in Section 2, our approach allows the definition of an overflow save integer multiplication function at the specification level. As a result, the multiplication operation in the* tick *function is replaced by the* safe_mult *function.*

4.3.2 Proving Overflow Detection

To ensure that the above specification detects the unsigned multiplication overflow reliable, properties that describe how this overflow is detected are required. Considering the semantic gap between $\mathbb{Z}$ and $\mathbb{N}/32$, discussed in Section 2.2, two properties have to be satisfied to either detect an overflow or to return the result of the multiplication. These properties are defined as theorem in Coq, as shown in Listing 6.

```
Theorem detect_overflow:
forall a b : Z,
a <= Unsigned32.max_unsigned /\
b <= Unsigned32.max_unsigned /\
a * b > Unsigned32.max_unsigned <->
safe_mult (Unsigned32.repr a)
          (Unsigned32.repr b) = None.

Theorem no_overflow:
forall a b : Z,
a <= Unsigned32.max_unsigned /\
b <= Unsigned32.max_unsigned /\
a * b <= Unsigned32.max_unsigned <->
safe_mult (Unsigned32.repr a)
          (Unsigned32.repr b) =
Some ( (Unsigned32.repr a) *
       (Unsigned32.repr b) ).
```

Listing 6: Theorems in Coq to verify the behavior of the *safe_mult* function.

The *detect_overflow* theorem says: for all a and b of the type $\mathbb{Z}$ which are less than or equal to the maximal *unsigned32* value and the multiplication of both values is greater than this maximal value if and only if (<->) our defined *safe_mult* function returns *None* for the same values that are converted to equivalent elements of the quotient ring *Unsigned32*. This property ensures that only in the case of an overflow *None* is returned. The second property that has to be satisfied is that the result of the multiplication operation has to be returned if no overflow occurs. The theorem *no_overflow* specifies this property and says: for all a and b of the type $\mathbb{Z}$ which are less than or equal to the maximal *unsigned32* value and the multiplication of both values is less than or equal to this maximal value if and only if (<->) our defined *safe_mult* function returns *Some()*. This property ensures that only in the case where no overflow occurs the result of the multiplication is returned.

```
Theorem safety_property_no_overflow:
forall counter delay clockFrequency a b
: Unsigned32.int,
delay < clockFrequency /\
Some (a) = safe_mult counter delay /\
Some (b) = safe_mult counter clockFrequency
<-> a <= b /\
tick counter delay clockFrequency <> None.

Theorem safety_property_overflow:
forall counter delay clockFrequency
: Unsigned32.int,
delay < clockFrequency /\
None = safe_mult counter delay /\
None = safe_mult counter clockFrequency <->
tick counter delay clockFrequency = None.
```

Listing 7: Theorem in Coq that represents the OCL safety property adapted to finite integer types.

```
safe_mult :: (Unsigned 32) -> (Unsigned 32)
          -> CLaSH.Prelude.Maybe(Unsigned 32)
safe_mult a b =
 case (CLaSH.Prelude.==) b 0 of {
  CLaSH.Prelude.True ->
   CLaSH.Prelude.Just((CLaSH.Prelude.*) a b);
  CLaSH.Prelude.False ->
   case (CLaSH.Prelude.<=) a
    ((CLaSH.Prelude.div) ((2^32) -1) b) of {
    CLaSH.Prelude.True -> CLaSH.Prelude.Just
     ((CLaSH.Prelude.*) a b);
    CLaSH.Prelude.False ->
     CLaSH.Prelude.Nothing}}
```

Listing 8: Extracted CλaSH model.

To verify the derived safety property (stated as an OCL invariant), described in Section 2, this invariant has to be transformed first, as the proposed specification uses finite types and the integer overflow has to be considered. The resulting theorems are shown in Listing 7.

For illustration purposes, we only explain the theorem *safety_property_no_overflow* in detail, as the *safety_property_overflow* theorem works analog. The theorem says: for all *counter*, *delay* and *clockFrequency*, where the *delay* is smaller than the *clockFrequency* and the multiplication does not overflow (*Some(a)* and *Some(b)* are returned) if and only if a is less than or equal to b and the specified *tick* function returns a constructor that is not *None*, i.e. either *Some(counter +1%unsigned32)* or *Some(1%unsigned32)*, since the *option* type has two constructors, as described above. As we have seen above, the problem discussed in Section 2.2 was addressed my providing a total function that wraps the multiplication operation. This function clearly distinguishes between the result of the multiplication and the overflow by a condition, because of Coq's built-in *option* type.

4.3.3 CλaSH Model Extraction

Having the hardware design specified and verified in Coq, we applied the design flow proposed in this work (Bornebusch et al. 2020), in order to extract an executable model. This flow extracts a CλaSH (Baaij et al. 2010) model from a specification automatically, by syntactical substitution. The extracted model for the *safe_mult* function is seen in Listing 8.

Analog to the way hardware designs are described in Coq, CλaSH describes circuits as recursive functions and data types. The unique representation of CλaSH models and the structured communication between their components, ensured by the static and strong type system, enables an automatic analysis and the final synthesis into low-level implementations, e.g. VHDL.

4.3.4 Integer Overflow Detection Pattern

In order to generalize the mechanism applied above to detect the integer multiplication overflow, we propose a pattern that allows the detection of any integer operation overflows. Listing 9 shows the proposed pattern. The angle brackets notate placeholders for the actual implementation of the condition that detects the integer overflow (<*overflowDetected*>) and the desired operation that is applied (<*operation*>).

```
f : a -> a -> option(a)
f x y = if <overflowDetected>
        then None
        else Some(x <operation> y)
```

Listing 9: Proposed overflow detection pattern.

This pattern uses the algebraic data type *option*, as described in Section 4.3.1. Any function that calls f has to make a case distinction: if the result is *Some(a)*, it can proceed with a as before, but if the result is *None*, it has to propagate the indefiniteness (or handle the overflow appropriately). This in turn makes the function calling f partial, and forces the propagation of the occurred overflow through the specification at the type level which enables the verification of properties, as described in Section 4.3.2.

5 Conclusion

In this work we propose a hardware design approach that allows the detection of integer overflows at the specification level. The established

design approach uses SysML/OCL at the specification level which implements *infinite* integer types. These types do not share the semantic behavior of the types that are implemented in a SystemC model as those are *finite* which results in a semantic gap between these two levels. This semantic gap motivates our approach, and we address this problem by specifying hardware designs using the proof assistant Coq and utilizing the CompCert integer library that describes *finite* integer types through dependent types. Such a specification enables the verifiable detection of integer overflows which is presented in this work. Furthermore, we proposed a generalizable pattern to detect overflows which extends our approach to detect overflows in any integer operation.

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

Accellera. (2016). Accellera Systems Initiative Inc SystemC Synthesizable Subset. (Version 1.5.7).

Arnout, G. (2000). Systemc standard. In *Asia and south pacific design automation conference (asp-dac)* (pp. 573–578).

Baaij, C., Kooijman, M., Kuper, J., Boeijink, A., & Gerards, M. (2010). CΛash: Structural descriptions of synchronous hardware using haskell. *Euromicro conference on digital system design (dsd)*, 714–721.

Bertot, Y., & Castéran, P. (2004). *Interactive theorem proving and program development - coq'art: The calculus of inductive constructions.* Springer.

Bornebusch, F., Lüth, C., Wille, R., & Drechsler, R. (2020). Towards automatic hardware synthesis from formal specification to implementation. In *Asia and south pacific design automation conference (asp-dac).*

Brady, E., McKinna, J., & Hammond, K. (2007). Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types. In *Trends in functional programming (tfp)* (pp. 159–176).

Brucker, A. D., & Wolff, B. (2006). *The HOL-OCL book* (tech. rep. No. 525). ETH Zurich.

Chlipala, A. (2013). *Certified programming with dependent types - A pragmatic introduction to the coq proof assistant.* MIT Press.

Cousot, P. (2012). Formal verification by abstract interpretation. In *NASA formal methods - international symposium, NFM* (pp. 3–7).

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., & Rival, X. (2005). The astreé analyzer. In *European symposium on programming* (pp. 21–30).

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., & Yakobowski, B. (2012). Frama-C - A software analysis perspective. In *International conference on software engineering and formal methods* (pp. 233–247).

Dietz, W., Li, P., Regehr, J., & Adve, V. S. (2015). Understanding integer overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.*, *25*(1).

Drechsler, R., Soeken, M., & Wille, R. (2012). Formal specification level: Towards verification-driven design based on natural language processing. In *Forum on specification and design languages (fdl)* (pp. 53–58).

Fähndrich, M., & Logozzo, F. (2010). Static contract checking with abstract interpretation. In *International conference on formal verification of object-oriented software* (pp. 10–30).

Hanna, F. K., & Daeche, N. (1992). Dependent types and formal synthesis.

Hilken, F., Niemann, P., Gogolla, M., & Wille, R. (2014). Filmstripping and unrolling: A comparison of verification approaches for UML and OCL behavioral models. In *International conference on tests & proofs (tap)* (pp. 99–116).

ISO/IEC. (2017). ISO International Standard ISO/IEC 14882:2017(E) Programming Language C++. (Edition 5).

Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., & Ferdinand, C. (2016). Compcert – a formally verified optimizing compiler. In *Embedded real time software and systems (erts).*

Necula, G. C., McPeak, S., Rahul, S. P., & Weimer, W. (2002). CIL: intermediate language and tools for analysis and transformation of C programs. In *European joint conferences on theorey and & practice of software* (pp. 213–228).

OMG. (2014). Object Management Group Object Constraint Language (OCL). (Version 2.4).

OMG. (2019). Open Management Group System Modeling Language (SysML). (Version 1.6).

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

Stoppe, J., Wille, R., & Drechsler, R. (2013). Data extraction from SystemC designs using debug symbols and the SystemC API. In *Ieee computer society annual symposium on vlsi (isvlsi)* (pp. 26–31).

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

Weilkiens, T. (2007). *Systems engineering with sysml / UML - modeling, analysis, design.* Morgan Kaufmann.