Safety First
About the Detection of Arithmetic Overflows in Hardware Design Specifications*

Fritjof Bornebusch[1], Christoph Lüth[1,3], Robert Wille[1,2], and Rolf Drechsler[1,3]

[1] Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2] Integrated Circuit and System Design, Johannes Kepler University Linz, 4040 Linz, Austria
[3] Mathematics and Computer Science, University of Bremen, 28359 Bremen, Germany
{fritjof.bornebusch,christoph.lueth}@dfki.de, robert.wille@jku.at, drechsler@uni-bremen.de

Abstract This work proposes an alternative hardware design approach that allows the detection of arithmetic overflows at the specification level. The established hardware design approach describes infinite integer types at that level while the model describes finite types. This opens a semantic gap between both levels, which means that arithmetic overflows cannot be detected at the specification level. To address this problem the CompCert integer library is utilized that describes finite integer types as dependent types using the proof assistant Coq. Properties that argue about these finite types can be specified and verified at the specification level. This closes the semantic gap the established hardware design approach suffers from.

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

1 Introduction

Circuits are an integral part of our lives. Their area of application extends from airplanes, to medicine, to toothbrushes. These areas of application lead to an increasing number of complexity in circuits. As complexity increases, so does the number of potential errors. For this reason, the increasing complexity needs to be considered in the development phase of hardware designs from the beginning.

To address the increasing complexity, hardware designs are described at different levels. The established hardware design approach starts with a formal specification, e.g. in SysML/OCL [22, 21, 27]. This specification describes the functional behavior of the hardware design and allows the verification of properties that argue about that design [8, 9, 24]. After specifying and verifying the design it is translated to a SystemC model, which is the de facto standard for

* Research supported by BMBF grant SELFIE, grant no. 01IW16001.

high-level synthesis (HLS) [2, 25]. This translation step is manually as OCL constraints cannot be translated automatically into executable SystemC code. The final step is the translation of the model to an implementation in a low-level hardware description language (HDL), e.g. Verilog. As SystemC only supports a restricted synthesizeable subset, this translation step is also manually [1].

The established hardware design approach reveals a *semantic gap* between the specification and the model respectively the implementation. The specification describes infinite integer types, while the model and the implementation describe finite integer types. This *semantic gap* lead to properties that hold for the specification, but not for the model, e.g. the absence of arithmetic overflows. Finite integer types describe a wrap-around or overflow behavior, as they implement a quotient ring [15, 13, 14, 11]. As arithmetic integer operations for finite types are not semantically equivalent to arithmetic integer operations for infinite types, these operations might lead to unintended behavior, which again lead to serious problems in the final hardware design implementation. Through the lack of tool support for automatically detecting arithmetic overflows in the model, the engineer has to detect them manually.

To address the problem of the *semantic gap* of the established hardware design approach an alternative hardware design approach is proposed. This approach describes finite integer types at the specification level using dependent types [7, 17]. These types allow the definition of operations, which detect arithmetic integer overflows at the specification level. Properties that argue about these operations can be verified to ensure the reliable detection of these overflows. After the verification process a model in the functional hardware description language (HDL) CλaSH can be extracted automatically [6], which again can be synthesized to an implementation on the Register-Transfer-Level (RTL) [3]. The proposed alternative hardware design approach closes the *semantic gap* the established approach has, by describing finite integer types at the specification level.

To achieve this, we start with a specification for the proof assistant Coq [4, 10]. Analog to the established hardware design approach this specification allows the verification of properties. The finite integer types are described by the CompCert integer library [19]. This library implements finite types as *dependent types* [17, 7] and allows the implementation of both signed and unsigned finite types of arbitrary sizes.

Note that this work extends the work [5] already published by the authors. For this reason, some figures and listings are borrowed from that work as can be seen in the captions. The extensions in this work include that, in particular, there may be no overflow in an arithmetic operation implementing the proposed function type, because of specified bounds. It is shown how an operation is specified in this case using the proposed overflow detecting function type. A generic property that has to be proven to show the absence of the overflow is specified. It is also shown why the overflow detecting operation cannot be changed automatically to its corresponding basic arithmetic operation if there is a proof of the absence of the overflow. Furthermore, the closure of functions that implement the function type for the proposed overflow detection pattern is specified and proven. This enables the cascading of overflow detecting operations, analog to their corresponding basic arithmetic integer operations. An evaluation regarding the impact of the speed and space for a synthesized hardware design that implements the overflow detection pattern is provided. This evaluation compares a hardware design using the basic arithmetic integer operations with their corresponding overflow detecting operations and shows the applicability of the proposed overflow detection pattern.

We present our work as follows: First, we explain the established hardware design approach and describe the problem we address in this work. Section 3 discusses the related work and why it is not suitable to address the problem of the established hardware design approach properly. In Section 4 and Section 5 the proposed hardware design approach is described, how the considered problem is addressed and how the CλaSH model is generated. Section 6 describes the proposed generalizable integer overflow detection pattern. Section 7 evaluates the proposed approach by comparing basic arithmetic integer operations with their corresponding operations, which detect overflows, regarding the speed and consumed space in the final hardware implementations. The Section 8 discusses the result of the evaluation and the applicability of the approach proposed in this work, while Section 9 concludes this work.

2 Motivation

In this section, we briefly review the established hardware design approach which is the motivation of this work. The established approach relies on a SysML/OCL specification that is later translated to a SystemC model manually. We show why the combination of SysML/OCL and SystemC is a problem for the detection of arithmetic integer overflows.

A traffic light controller serves as a running example to illustrate the established hardware design approach as well as the approach proposed in this work. This controller is inspired by [23].

2.1 The Established Hardware Design Approach

The established hardware design approach starts with a SysML/OCL [22, 21, 27] specification, which can later be used for the verification of properties [8, 9, 24]. The structure of the design is described by SysML class diagrams, while the functional behavior is described by OCL constraints. These constraints are specified as preconditions and postconditions as well as invariants.

Example 1. Figure 1 shows the SysML class diagram for the traffic light controller that serves as a running example in this work. The controller connects three different traffic lights: for the *trams*, *cars* and *pedestrians*. The basis of this controller are two finite state machines (FSMs), implemented by the *switch* and the *tick* function. The OCL constraints for these state machines can be seen in Listing 1.1.

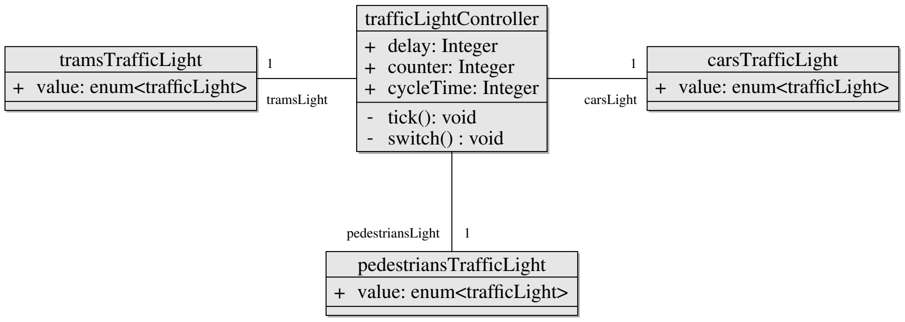

Figure 1. SysML class diagram of the traffic light controller [5]. This controller serves as a running example in this work.

```
context trafficLightController::tick()
 pre pre_incr_counter : self.counter < (self.delay -1) * self.cycleTime
 post incr_counter : self.counter = self.counter@pre +self.cycleTime and
                     self.delay = self.delay@pre and
                     self.cycleTime = self.cycleTime@pre

context trafficLightController::tick()
 pre pre_reset_counter : self.counter >= (self.delay -1) * self.cycleTime
 post reset_counter: self.counter = 0 and
                     self.delay = self.delay@pre and
                     self.cycleTime = self.cycleTime@pre

context trafficLightController::switch()
 pre pre_switch : self.counter >= (self.delay -1) * self.cycleTime and
                  self.tramsLight.value = Red and
                  self.pedestriansLight.value = Red and
                  self.carsLight.value = Green
 post post_switch : self.tramsLight.value = Red and
                    self.pedestriansLight.value = Red and
                    self.carsLight.value = Yellow

 inv: self.counter > -1
 inv: self.delay > 0
 inv: self.cycleTime > 0
```

Listing 1.1. OCL constraints for the *tick* function and the *switch* function introduced in Figure 1. Additionally, the range for the variables, *counter*, *delay* and *cylcleTime* is restricted by invariants.

The *tick* function represents the clock in the SysML/OCL specification. As seen in Listing 1.1, it increases a *counter* and resets it back to 0 if an upper bound is reached (pre_reset_counter). This *counter* is used to count the amount of nanoseconds until the *switch* function is called. The controller considers traffic situations, such as rush hour. For this reason, the *delay* can be configured at runtime, which allows the configuration of a dynamic transition time. The transition time is the time the *counter* takes to reach its upper bound. Until that bound is not reached, the *counter* is increased by the *cycleTime* as the OCL constraints *pre_incr_counter* and *incr_counter* states. If the upper bound is reached, the *counter* is reset to 0 as state by *reset_counter*. In this case, the FSM implemented by the *switch* function moves into a new state where the traffic light for the cars is no longer green, but yellow as stated by the constraints *pre_switch* and *post_switch*. The *cycleTime* is constant and indicates the cycle time of the hardware in nanoseconds (nsec). For example, if the transition time is 30 seconds the *delay* has to be set to 1.500.000.000 with a *cycleTime* of 20 nsec.

The *switch* function implements the state transitions for the traffic lights. This state machine determines whether a traffic light is switched on or off in order to avoid situations such as the lights for cars and pedestrians are both green at the same time. The different states for the lights, are encoded as *Green*, *Yellow* and *RedYellow* and *Red*. An exemplary state transition is stated by the *pre_switch* and the *post_switch* constraints as seen in Listing 1.1. Note that the *delay* might not always be necessary for the state transition, e.g. the pedestrians might have a constant transition time while the transition time for the cars rely on the *delay* (rush hour). Since this work considers arithmetic integer overflows, the state machine is not described in detail, as no arithmetic operations are involved in the state transitions.

After specifying and verifying the behavior of the traffic light controller in SysML/OCL, a model in SystemC is described. This step is manually as indeed the SysML structure can be translated automatically in the form of C++ classes[4]. However, the behavior specified by OCL constraints cannot automatically be translated to executable SystemC code.

Example 2. Listing 1.2 shows the implementation of the *tick* function in the SystemC model.

```
 1 sc_uint<32> counter, delay, cycleTime;
 2 States states;
 3
 4 void tick() {
 5  if ( counter < (delay -1) * cycleTime )
 6   counter = counter + cycleTime;
 7  else
 8   counter = 0;
 9   switch_();
10 }
```

Listing 1.2. Implementation of the *tick* function, introduced in Listing 1.1, of the SystemC model.

As specified by the OCL constraints in Listing 1.1, the SystemC model increases the counter by the *cycleTime* until the upper bound is reached, as seen in Line 5 of Listing 1.2. Otherwise, the counter is reset to 0.

2.2 Considered Problem

To illustrate the problem that motivates this work, we take a look at the safety property that can be derived from the specification, seen in Listing 1.1. This safety property holds for the specification, but not for the model and in this section we show why not.

Example 3. Listing 1.3 shows the safety property that is derived from the SysML/OCL specification. This property is specified as an OCL invariant.

[4] Note that SystemC is a collection of C++ class libraries designed to describe hardware designs.

```
context trafficLightController
 inv: self.counter < self.delay * self.cycleTime
```

Listing 1.3. Safety property derived from the OCL constraints introduced in Listing 1.1.

This invariant determines that the *counter* is less than the multiplication of the *delay* and the *cycleTime*. As the SysML data type *Integer* is infinite, the property holds for the specification.

To prove that the safety property holds we show that, if the precondition, invariants and safety property hold in the pre state and the postcondition holds in the post state, then the safety property holds in post state as well.

This proof consists of a case analysis of the OCL constraints for the *tick* function, seen in Listing 1.1. The notation x' is used to denote the value of the variable x in the post state. The *self* prefix seen in the OCL constraints is also omitted.

Example 4. In order to show that the safety property, seen in Listing 1.3, holds in the above specification, we take a look at some assumptions that can be derived from the specification, seen in Listing 1.1. We assume that the preconditions and the safety property hold in the pre states and that the postconditions hold in the post states.

Using these assumptions, we want to prove that if we are in a pre state in which both the precondition and the safety property hold, and we move to the post state in which the postcondition holds, then the safety property also holds.

We prove this property by case analysis. The first case is the precondition *pre_reset_counter* and the postcondition *reset_counter*. The second case is the precondition *pre_incr_counter* and the postcondition *incr_counter*. In the first case the *counter* is reset to 0 in the postcondition. The invariants state that the *delay* and the *cycleTime* are both greater than 0, so the safety property holds in the post state. To prove the safety property for the second case, we take a look at the precondition *pre_incr_counter*. Since the monotonicity of the addition holds in $\mathbb{Z}$, we add *cycleTime* to both sides of the precondition. This gives us the postcondition *incr_counter* on the left side. If we dissolve the right side, we see that the safety property holds in the post state.

$$
\begin{aligned}
&\mathsf{counter} + \mathsf{cycleTime}' < ((\mathsf{delay}' - 1) * \mathsf{cycleTime}') + \mathsf{cycleTime}' \\
= &\mathsf{counter} + \mathsf{cycleTime}' < (\mathsf{delay}' * \mathsf{cycleTime}' - \mathsf{cycleTime}') + \mathsf{cycleTime}' \\
= &\mathsf{counter} + \mathsf{cycleTime}' < \mathsf{delay}' * \mathsf{cycleTime}'
\end{aligned}
$$

Now that we have proven that the safety property holds in the post states of the SysML/OCL specification, why does it not hold in the SystemC model? If we consider the case analysis of the proof for the SystemC model, we see that for the first case the proof holds. However, for the second case the monotonicity of the addition does not hold. The SystemC model describes the quotient ring $\mathbb{Z}_{>-1}/32\mathbb{Z}_{>-1}$. This ring describes an integer type of limited size and that is precisely the reason why the safety property does not hold in the SystemC model, as the monotonicity of the addition does not hold for quotient rings.

In other words, the multiplication operation in the SysML/OCL specification is not semantic equivalent to the one in the SystemC model, as in SystemC all integer types describe a quotient ring: $\mathbb{Z}/m\mathbb{Z}, m \in \mathbb{N}$ (signed integer) or $\mathbb{Z}_{>-1}/m\mathbb{Z}_{>-1}, m \in \mathbb{N}$ (unsigned integers). The *semantic gap* between SysMLs infinite integer type and SystemCs finite integer types motivates this work and results in the proposal of an alternative hardware design approach that allows the description of finite integer types at the specification level.

Example 5. Let us consider again the translation step of the OCL constraints seen in Listing 1.1 for the SystemC model seen in Listing 1.2. The model assumes that the implementation of the unsigned integer multiplication operation is the same as in the specification. This assumption is understandable at first glance, since the same behavior is apparently described. However, as we have seen above this is not the case, as the integer type in the specification is infinite, while the one in the model is finite. As a result, the SystemC model violates the safety property, shown in Listing 1.3.

This violation bears a direct impact on the change of the configurable delay at runtime and thus on the transition time of the state machine, which considers traffic situations such as rush hour. For instance, a changed delay might lead to unintended behavior as the multiplication operation on the quotient ring *sc_uint<32>* implements a wrap-around behavior. In this case, instead of increasing the transition time it is decreased which is a serious problem.

A look in the C++ standard[5] reveals two different behaviors of integer arithmetic regarding overflows.

Unsigned integer arithmetic defines total functions and does not overflow. A result that cannot be interpreted by the resulting data type is reduced by $2^n, n \in \mathbb{N}$, where n is the number of bits in the value representation, e.g. *sc_uint<32>*. Through the modulo operation, arithmetic operations on these data types implements a wrap-around behavior. So in the case of unsigned arithmetic the operation might lead to unintended behavior.

Signed integer arithmetic does overflow and defines either total functions or partial functions, depending on the underlying hardware platform. The functions are total, if the platform represents the values in the 2's complement. In this case, the same wrap-around behavior is implemented as for the unsigned integer arithmetic. If the platform uses traps[6] to indicate an overflow the arithmetic function becomes partial, as in this case the function does not define a return value for a pair of input values. As the behavior of signed integer arithmetic is platform dependent, it is undefined in general.

[5] The current standard for the C++ programming language is specified in ISO/IEC 14882:2017.

[6] A trap is a software interrupt that is triggered due to an instruction execution, e.g. division-by-zero, by the processor.

The term *arithmetic integer overflow* often refers to both unsigned integer and signed integer arithmetic [15, 13]. For this reason, we use that term in the rest of this work to address both behaviors.

The basic problem of the *semantic gap* between SysMLs infinite integer types and SystemCs finite types motivates our work. To address this problem a semantic equivalent finite type is needed at the specification level as hardware descriptions are finite by design and, therefore, rely on these types. Having such types at the specification level enables the clear distinction between the correct result of an arithmetic integer operation and the occurred overflow. We call this distinction the *detection of overflows.* As overflows are inevitable on finite integer types this work proposes an overflow detection pattern by a total function that makes the distinction between the result of an arithmetic integer operation and the overflow explicitly.

In the next section, we evaluate the related work and discuss why it is not suitable to address the problem described above properly. This discussion leads eventually to the alternative hardware design approach.

3 Related Work

In this section we evaluate and discuss the related work to show why a specification in SysML/OCL and a model in SystemC is not suitable to detect integer overflows properly.

To detect integer overflows in the SysML specification the possibility to define finite integer types of arbitrary sizes need to be implemented, but this is not the case in the current standard [22]. Of course, invariants can be used to restrict SysMLs *Integer* type by describing a lower and upper bound. But, these bounds are independent of the integer type used in the SystemC model. For instance, after the automatic generation of the SystemC class structure from SysML: what should the equivalent type to SysMLs *Integer* type be in SystemC? Either a standard type, like *Integer* is always represented as *sc_uint<32>*, but in this case the bounds can never change, or the extracted type of the model is dependent from the bounds chosen in the specification. Such a restriction can be described by OCL invariants, but it is not possible to extract these invariants in executable SystemC code automatically. If these bounds are translated manually to the SystemC model, they might change during the development phase of the model. For example, it was discovered that a different type, e.g. *sc_uint<31>*, is needed which again invalidates the bounds from the specification. The basic problem is that a SysML/OCL specification describes infinite integer types while the SystemC model describes finite ones.

To detect integer overflows directly in the SystemC model, the automatic overflow detection of C++ programs need to be considered. The detection of overflows by a C++ compiler is quite challenging, because of the low level nature of C++. The standard allows bit manipulations, which are very common [15]. This makes it very challenging to detect overflows by the compiler reliably, as it is not always clear whether such a manipulation is intended by the engineer or not. Furthermore, the standard defines undefined behavior semantics that allow optimizations by the compiler [15]. For this reason, C++ compiler can only detect arithmetic overflows in constant-expression evaluation, but not in general. As a result, C++ compilers are not suitable to detect arithmetic integer overflows automatically.

Since there is no support by the compilers static code analysis tools, such as Astrée [13] or Frama-C [14], should to be considered.

Astrée relies on abstract interpretation [12, 16] and aims to prove the absence of runtime errors, such as integer overflows, in C programs. Abstract interpretation is used to derive a computational abstract semantic interpretation from a behavior expressed in a programming language. The resulting interpretation does not contain the actual values, but focuses on dedicated parts of the program. The scope of the static analysis is determined by these parts and define what kinds of errors are detected. The limit of abstract interpretation is the analysis of loops, as loops define an infinite number of paths in the interpretation tree. SystemC models are C++ programs, which is not the input language of Astrée. Astrée could, of course, be extended to support C++ programs, but SystemC describes hardware designs. Such designs rely on parallel execution and run in infinite loops. As mentioned above, loops create an infinite number of paths in the interpretation tree. For this reason, Astrée is not suitable to detect integer overflows in hardware designs.

Frama-C is another static code analysis tool which relies on *C Intermediate Language* (CIL) [20] and supports annotations written in *ANSI/ISO C Specification Language* (ACSL) [14]. Frama-C enables the application of different static analysis techniques, such as deductive verification of annotated C programs by automatic provers, e.g. Z3 [14]. The detection of integer overflows is supported by the *Runtime Error Annotation Generation* (RTE) plugin which includes the generation of annotations by syntactic constant folding in the form of assertions. RTE seeds these annotations into other plugins, e.g. for generating weakest-preconditions with proof obligations. Similar to Astrée the input language for Frama-C is a C program, which could , of course, be extended to support C++ programs. But the static analysis of the infinite loops hardware designs rely on is quite challenging. For this reason, Frama-C is not suitable to detect integer overflows in SystemC models.

As discussed in this section a SysML/OCL specification and a SystemC model are not suitable to detect integer overflows. The specification describes infinite types and lacks the definition of finite integer types of arbitrary sizes. The model describes finite integer types and does not get support by compilers or static analysis tools for detecting integer overflows. As a result, the engineer need to detect overflows pro-active and explicitly at the model level.

The problem discussed above in combination with the related work leads to the following question: *Can arithmetic integer overflows in hardware designs be detected at the specification level?*

4 Proposal of the Alternative Design Approach

In this section, we propose an alternative design approach that addresses the problem of the *semantic gap* of the established hardware design approach, described in Section 2.

The alternative approach uses the proof assistant Coq [4, 10] to specify and verify the functional behavior of hardware designs. Coq describes functional behavior in a specification language, called Gallina, which is based on the *Calculus of Inductive Constructions* (CiC). This calculus combines a higher-order logic with a richly-typed functional programming language. As higher-order logic is too expressive for automatic reasoning, a separate tactic language is used that provides proof methods, but let the user define his own ones as well. Therefore, proof assistants are also known as interactive theorem provers.

As discussed in Section 2.2 the problem of the established approach is the *semantic gap* between the infinite integer types in SysML and the finite integer types in SystemC. To address this problem we use *dependent types* [17, 7] to implement finite integer types in Coq. These types are used to functionally describe the limited size bit vectors for the inputs and outputs of hardware designs. The idea to describe hardware designs using dependent types is not new and started back in the 1990s. These types allow a type definition that relies on an additional value. For instance, the type A^n defines a vector of the length $n, n \in \mathbb{N}$ with elements of the arbitrary type A. We say that A depends on n that is where the name *dependent type* comes from. Proof Assistants, like Coq, allow the definition of dependent types by the user which gives us the opportunity to describe hardware designs with finite integer types at the specification level. In order to describe such types, we utilized the CompCert integer library to describe both signed and unsigned integer types of arbitrary sizes [19].

In contrast to the established approach, we use the proof assistant Coq at the specification level to specify and verify hardware design. Furthermore, we describe finite integer types using *dependent types*, which enables the detection of integer overflows at the specification level. We describe below how the detection is specified and verified and how a final hardware implementation is generated automatically from a specification written in Gallina.

4.1 Detection of Integer Overflows

As described in Section 2.2, we need an explicit distinction between the correct result of an arithmetic integer operation, e.g. multiplication, and the occurred overflow. Therefore, we use a dedicated type which either contains the result of an operation or indicates an occurred overflow. This data type is called *option*, as seen in Listing 1.4, and has two constructors: *None* and *Some* which takes an arbitrary type (A) as parameter.

```
Inductive option (A : Type) : Type :=
  | Some : A -> option A
  | None : option A.
```

Listing 1.4. Definition of the *option* type in Gallina provided by the Coq standard library (Coq.Init.Datatypes) [5].

The constructor *Some* contains the result, while the constructor *None* indicates the overflow. Consider again the running example introduced in Section 2.1. This example uses a multiplication operation of the type:

$$n \in \mathbb{N} \Rightarrow Unsigned^n \rightarrow Unsigned^n \rightarrow Unsigned^n$$

, where an overflow cannot be distinguished from the actual result. We use the term *basic arithmetic operation* for arithmetic integer operations that have the above type. Using the *option* type, we create an alternative multiplication operation, called *safe_mult* seen in Listing 1.5, which has the type:

$$n \in \mathbb{N} \Rightarrow Unsigned^n \rightarrow Unsigned^n \rightarrow option\ Unsigned^n$$

The *safe_mult* operation returns *None* in the case of an integer overflow and *Some(A)* otherwise. Now, the question is: *How are both cases explicitly distinguished?* We take a look at the case where an overflow occurs to answer this question. Note that both a and b are of the type $Unsigned^n$ and $x \mapsto y$ donates: x is transformed to y. The function *max* returns the maximum representable value of a given data type.

$$a * b > max(a) \mapsto b \neq 0 \wedge a > max(a)/b$$

The condition on the left side (x) indicates the intuitive check of an overflow. If the result of a multiplication is larger than the maximum value of the integer type of the operand (max(a)) than, obviously, an overflow occurred in the multiplication. But, if we implement this using finite integer types this condition always evaluates to true, as by definition there is no larger value of a type than its maximum. For this reason, we need to transform the left side to the right side (y). The condition on the right side evaluates only to true in the case of an integer overflow in the multiplication. If the condition evaluates to false both operands can be multiplied safely. In order to avoid a *division-by-zero* error, we first ensure that b is not equal to zero.

By using the alternative *option* type definition, described above, and the transformed condition, we are able to implement a multiplication operation that detects an occurred overflow, as seen in Listing 1.5.

```

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

Listing 1.5. Definition of the *safe_mult* function in Gallina that detects a multiplication overflow for 32-bit unsigned values [5].

Like in the SystemC example, illustrated in Listing 1.2, our multiplication is defined for 32-bit unsigned values (*Unsigned32.int*). This type definition was implemented using the CompCert integer library. As seen in Listing 1.4, our function implementation returns *Some(a*b)* in the case no overflow occurs and *None* otherwise.

After the definition of the *safe_mult* function in Coq, a proof is needed that verifies that the definition satisfies its specification. This specification is formulated as theorems in Coq. Two theorems are formulated in order to proof the *safe_mult* definition: the detection of the occurred overflow and the returning of the result of the unsigned 32-bit multiplication operation if no overflow occurs. To verify this, we show that the multiplication defined for 32-bit unsigned values maps the multiplication for integer values ($\mathbb{Z}$) which is a subset of it, but detects the occurred overflow. Both theorems are shown Listing 1.6.

```
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 1.6. Theorems specified Coq to verify that the *safe_mult* function detects the overflow correct and returns the result of the multiplication otherwise [5].

The theorem *detect_overflow* states: if the two values *a* and *b* of type *Z* are less than or equal to the maximum value of the unsigned 32-bit integer type (Unsigned32.max_unsigned) and their multiplication is greater than this value, *None* is returned. The function *Unsigned32.repr* comes from the CompCert Integer library and converts a value of type *Z* into a value of type *Unsigned32.int*. The theorem *no_overflow* states: if two values *a* and *b* of type *Z* are less than or equal to the maximum value and their multiplication is also less than or equal to the maximum value, *Some(A)* is returned. The arbitrary type *A* is in this case the type *Unsigned32.int*.

After specifying and verifying a safe multiplication integer operation, the *tick* function, described in Listing 1.1 has to be specified in Gallina. This specified function has also been changed to use the *safe_mult* function, specified above, as seen in Listing 1.7. Like for the SystemC model, seen in Listing 1.2, we specified an unsigned 32-bit integer value (Unsigned32.int), which was described using the CompCert integer library [19]. The specification of the *tick* function can be seen in Listing 1.7.

```
Definition switch (s : State) : State.

Definition tick (input : Unsigned32.int*Unsigned32.int*Unsigned32.int*States)
 : option Unsigned32.int*State :=
 match input with
    | (counter, delay, cycleTime, state) =>
       match safe_mult (delay -1%unsigned32) cycleTime with
        | Some res => if counter <? res
                         then (Some(counter + cycleTime), state)
                         else (Some(1%unsigned32), switch state)
        | _ => (None, state)
    end
 end.
```

Listing 1.7. Specification of the *tick* function in Gallina, which used the *safe_mult* function introduced in Listing 1.5.

Note that the *switch* function, seen in Figure 1, has a different type in the Coq specification, shown below. Pure functional languages, such as Gallina, do not allow internal states, in contrast to a SysML specification. For this reason, the type of the *switch* function had to be changed. As this work considers the detection of arithmetic overflows and there are no arithmetic overflows involved in the state transitions of that function, we omit the function implementation.

Since the unsigned multiplication operation has semantically changed the question is: *How to handle the case where an overflow occurred?* The handling highly depends on the environment the traffic light controller runs in, e.g. return to a safe state or ignore the new configured delay. As this would be out of scope for this work, the *tick* function just returns an instance of the tuple *option Unsigned32.int*State*. The first value of the tuple contains an instance of the type *option Unsigned32.int*, while the second value of the tuple is the new state. This state can either be the same as the old one or be changed by the *switch* function. The overflow is not handled by this function directly, but is propagated to the calling function instead. The state remains unchanged in this case.

After defining the *tick* function in Gallina, the verification of the property is needed that the definition still satisfies the safety property, shown in Listing 1.3. This property had to be translated to Coq first. This transformation results in the definition of two theorems, which is shown in Listing 1.8. Theorem *safety_property_no_overflow* describes the case no overflow occurs and Theorem *safety_property_overflow* describes the case an overflow occurs. The verification of those theorems proves that the *tick* function either changes the *counter* or propagates the detected overflow.

```
Theorem safety_property_no_overflow:
  forall counter counter' delay cycleTime res : Unsigned32.int,
  forall s s' : State,
  Some(res) = safe_mult delay cycleTime <->
  tick (counter, delay, cycleTime, s) = (Some (counter'), s') /\
  counter' = (delay-1) * cycleTime /\ counter' < res.

Theorem safety_property_overflow:
  forall counter delay cycleTime : Unsigned32.int,
  forall s : State,
  None = safe_mult (delay -1) cycleTime <->
  tick (counter, delay, cycleTime, s) = (None, s).
```

Listing 1.8. Theorem in Coq that represents the OCL safety property adapted to finite integer types.

The first theorem states: if no overflow occurred in the multiplication of *delay* and *cycleTime*, the tick function returns the new counter (*counter'*) and the new state (*s'*). Note that the new state might be the old state as it depends on a condition whether the state is changed or not, as seen in Listing 1.7. The new counter (*counter'*) is the result of the multiplication of *delay -1* and *cylceTime* and is less than *res*, which is essentially the safety property, shown in Listing 1.3.

The second theorem states: if the result of the safe multiplication of *delay -1* and *cycleTime* is *None* than the *tick* function returns *None* as well. The state remains unchanged in this case, as described above.

In this section, we illustrated how to specify a safe multiplication operation using *dependent types* in order to detect an overflow for the 32-bit unsigned values. This was the problem, we described in Section 2.2. The specification of the *tick* function, shown in Listing 1.1, was transformed into a Coq specification manually and it was verified that the safety property, shown in Listing 1.3 satisfies our specification, as seen in Listing 1.8. This shows that we have successfully addressed the problem of missing finite integer types at the specification level, as described in Section 2.

5 Extraction of the CλaSH Model

In this section, we describe how the specification in Gallina, described above, is translated to a CλaSH model and finally to an RTL implementation that can be synthesized on an FPGA.

To illustrate the extraction process from a specification to a model in the functional hardware description language CλaSH [3, 18], we take a look at Coq's extended extraction process, proposed in this work [6]. The process allows the extraction of a specification in Gallina into an executable CλaSH model. The extraction is done by syntactical replacement, since Gallina is a functional specification language and follows the same semantic rules as functional programming languages, e.g. Haskell or OCaml. The extraction process has two different modes. The first mode is that it extracts everything that is related to the function that should be extracted, such as other called function or data types. The second mode is the replacement of functions and data types by their semantic equivalent representations in the target language. This mode is used to intrinsic functions or to replace constant functions that have a different syntax. For instance, the constant function *Unsigned32.max_unsigned* used in Listing 1.6 is replaced by $(2^{32}) - 1$ in the CλaSH model, as seen in Listing 1.9. The specification and verification of a behavior by a proof assistant and the extraction of this behavior afterwards to executable code is called *certified programming* [10].

```
switch :: State -> State

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

tick :: ((,) ((,) ((,) (Unsigned 32) (Unsigned 32)) (Unsigned 32)) State) ->
        (,) (CLaSH.Prelude.Maybe (Unsigned 32)) State
tick input =
  case input of {
   (,) p states ->
    case p of {
     (,) p0 cycleTime ->
      case p0 of {
       (,) counter delay ->
        case safe_mult ((CLaSH.Prelude.-) delay 1) cycleTime of {
         CLaSH.Prelude.Just res ->
          case (CLaSH.Prelude.<) counter res of {
           CLaSH.Prelude.True -> (,) (CLaSH.Prelude.Just
            ((CLaSH.Prelude.+) counter 1)) states;
           CLaSH.Prelude.False -> (,)(CLaSH.Prelude.Just 1) (switch states)};
         CLaSH.Prelude.Nothing -> (,)CLaSH.Prelude.Nothing states}}}}
```

Listing 1.9. Extracted CλaSH model of the *safe_mult* and *tick* function introduced in Section 1.5.

CλaSH borrows its syntax and semantics from the functional programming language Haskell. Combinational circuits are described as recursive functions and synchronous sequential ones as a combination of these functions with a finite state machine, either as a Mealy machine or a Moore machine [3]. After the CλaSH model was extracted the final RTL (Register-Transfer-Level) implementation, e.g. in VHDL or Verilog, it can be synthesized automatically. The unique representation of hardware model and the structured communication between the components, ensured by the type system of CλaSH, allows the automatic analysis of models and the final synthesis into a low-level RTL implementation, e.g. VHDL or Verilog.

6 Overflow Detection Pattern

In this section, we propose a detection pattern that can be used to detect integer overflows in different arithmetic operations. The pattern defines a total function that distinguishes the result of the arithmetic operation from the overflow by the *option* type described in Section 4.1. The proposed detection pattern is shown in Listing 1.10.

The pattern requires two definitions. First, a data type that defines two constructors: *None* and *Some A*, where *A* is an arbitrary finite integer type. Second, a function of the type: $A \rightarrow A \rightarrow$ *option A*. This function takes two arguments of the integer type *A* and returns a value of the previous defined *option* type. Where *None* indicates the overflow and *Some a* indicates the result of the operation that was executed. This case analysis is made by a condition (overflowDetected), e.g. by the one defined in Section 4.1 for the multiplication of unsigned 32-bit values.

```
data option A = None | Some A

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

Listing 1.10. Proposed overflow detection pattern [5].

The specified function *f* is used to replace the basic arithmetic operation, e.g. unsigned multiplication, that is not able to distinguish an overflow from the correct result. In order to verify that function *f* distinguishes the overflow from the correct result, two theorems have to be proven. A proof of the first theorem, as can be seen in Theorem 1, verifies that for all inputs which cause an overflow for the performed arithmetic operation *None* is returned.

Theorem 1 (Detect overflow in integer arithmetic operation). $\forall x, y \in$ *A, where A is an arbitrary finite integer type.*

$$<\textit{overflowDetected}>\ x\ y \Longleftrightarrow f\ x\ y = \textit{None}$$

A proof of the second theorem, as can be seen in Theorem 2, verifies that for all inputs that do not cause an overflow for the performed arithmetic operation the result of this operation is returned.

Theorem 2 (No overflow in arithmetic integer operation). *$\forall x, y \in A$, where A is an arbitrary finite integer type.*

$$\textit{not}\ (\textit{<overflowDetected>}\ x\ y) \Longleftrightarrow f\ x\ y = \textit{Some}\ (x\ \textit{<operation>}\ y)$$

Now, that we have defined the overflow detection pattern, one question remains: *Is this pattern always necessary?*

In the following, we answer this question and explain in which cases it is necessary and in which one it is not. If we look at the general behavior of arithmetic integer operations, an overflow might always occur. The result of an arithmetic operation can potentially be larger than its finite integer type is able to represent. This might lead to an unintended wrap-around behavior, as explained in Section 2.2. So in general, the overflow detection pattern, described above, should be applied.

However, there are cases where this pattern can be avoided. First, it has to be verified that the arithmetic operation that is applied on both operands never causes an overflow. The theorem that has to be proven can be seen in Theorem 3.

Theorem 3. *$\forall x, y \in A'$, where $A' \subset A$ and A is an arbitrary finite integer type.*

$$f\ x\ y = \textit{Some}(x\ \textit{<operation>}\ y)$$

If and only if this theorem holds, then there is no need to replace the basic arithmetic operation with the one defined by *f*. The general steps for the proposed pattern are the following: first, define the function f for the desired integer type and operation, second, prove the above theorem, to verify that the chosen subset of values is never too large to cause an overflow. As described above, this is not the case in general, but might be in particular.

Now, that we have the proof, that the arithmetic operation defined by function *f* never returns an overflow, the question is: *Can this proof be used to replace function* f *in a specification by its corresponding basic arithmetic operation automatically?*

To answer this question, we take a look again at the extraction feature of Coq. As mentioned above, Coq provides the specification language Gallina and a tactic language for property proving. The extraction process only extracts the functional behavior of a specification written in Gallina to an executable target language. Theorems and Lemmas, which state propositions, are ignored during this process, as they do not have a semantic equivalent representation in the target language.

Thus, it is not possible to automatically replace the defined function f by its corresponding basic integer operation using a proof without changing Coq's entire extraction process. Furthermore, as both functions are semantically not equivalent such a replacement would effect the entire specification recursively.

As discussed above, the automation process is quite challenging as the type of function f would change from $A \rightarrow A \rightarrow \textit{option}\ A$ to $A \rightarrow A \rightarrow A$ what recursively effects the entire specification. A more suitable way is to propagate the value *Some A* of function *f* through the specification. This avoids the recursive changing of all functions depending on *f* manually as the type of function *f* remains the same.

In summary, if and only if Theorem 3 holds, we have a proof that the specification of function *f* can be changed to just return *Some(x <operation> y)* as no overflow occurs.

6.1 Closure of Functions

As we propose an overflow detection pattern in this work that has the function type $A \rightarrow A \rightarrow$ *option* A, functions that implement this pattern are no longer closed. A set is called closed under an operation if an operation performed on members of a set always produce a member of that set. For this reason, it is not possible to cascade these functions, e.g. *safe_mult (safe_mult 3 4) 5*. In order to address this problem we implement the *option* monad in Coq. Monads come from the mathematical field of category theory and model computations [26]. It is used as a design pattern in functional languages and represents a specific form of computation. Analog to the implementations of monads in other functional languages, e.g. Ocaml, two functions were implemented, seen in Listing 1.11.

Since the cascading of these functions might not always be wanted, we propose this monad instead of changing the proposed pattern, seen in Listing 1.10. This allows a greater flexibility between both use cases.

```
Definition ret {A : Type} (x : A) : option A := Some x.

Definition bind {A : Type} (f : A -> A -> option A) (x y: option A)
  : option A :=
  match (x, y) with
    | (Some x', Some y') => f x' y'
    | (_,_) => None
  end.
```

Listing 1.11. Definition of the option monad operations.

The *option* monad contains two functions: *ret* and *bind*. The *ret* function takes an argument of type *a* and transforms it into a value of the type *option A*. The *bind* function takes a function of the proposed pattern type (f) and two arguments of the type *option A* (x and y). If both arguments contain a value of the type *A*, the function *f* is called with these values. Otherwise, *None* is returned.

The *option* monad applies to all functions that require two arguments and return the *option* type. Since it is not restricted to one dedicated type, it can be used for all functions that implement the proposed overflow detection pattern, seen in Listing 1.10. To verify the correct behavior of the *bind* function, two theorems were proved.

```
Theorem fIfSome:
  forall (A : Type),
  forall f : (A -> A -> option A),
  forall x y : option A,
  forall x' y' : A,
  x = Some (x') /\ y = Some (y') -> bind f x y = f x' y'.
```

Listing 1.12. Theorem that verifies that the function *f* is only called by the *bind* function if both arguments are of type *A*.

The first theorem, seen in Listing 1.12, verifies that if both arguments x and y contain values of type A (x' and y') then the *bind* function calls the function f with these two values, as seen in Listing 1.12. This theorem verifies that only in the case were both arguments for f contain values this function is called.

The second theorem, seen in Listing 1.13, verifies that if either the first argument of the function f (x) or the second (y) is *None* the *bind* function returns *None*, as seen in Listing 1.13. This theorem verifies that the function f is not called with invalid values (*None*).

```
Theorem noneIfNone:
  forall (A : Type),
  forall f : (A -> A -> option A),
  forall x y : option A,
  x = None \/ y = None -> bind f x y = None.
```

Listing 1.13. Theorem that verifies that in the case of invalid arguments for function *f None* is returned by the *bind* function.

The *option* monad closes the operations that implement the proposed overflow detection pattern, which allows the cascading of these functions. e.g. *bind safe_mult (bind safe_mult (ret 3) (ret 4)) (ret 5)*. The cascading of operations enables the formulation of more complex operations based on the application of the basic arithmetic operations. The *bind* function propagates an occurred overflow through the cascaded operations. At the end of the calculation it can be evaluated whether the result is correct or if there was an overflow in one of the operations.

7 Evaluation

In this section, we evaluate the hardware design approach proposed in this work. The foundation of this evaluation is a comparison of basic arithmetic integer operations with their corresponding overflow detecting operations regarding their impact of the speed and consumed space. To determine these values the operations were specified for both signed and unsigned integer operations and used by the traffic light controller, seen in Section 4.1. The resulting specification was synthesized on an FPGA using the synthesize process introduced in this work [6].

7.1 Integer Overflow Detection Implementations

This section introduces the different implementations of the overflow detecting arithmetic integer operations used for the evaluation. All implementations follow the pattern introduced in Section 6.

```
Definition safe_add_unsigned (a b : Unsigned32.int) : option Unsigned32.int
    :=
  if a >? (Unsigned32.max_unsigned - b)
    then None
    else Some (a+b).
```

Listing 1.14. Definition of the *safe_add_unsigned* function in Gallina that detects an overflow in the addition operation for unsigned 32-bit values.

Listing 1.14 shows the implementation that detects an overflow in the addition operation of two unsigned 32-bit values. The condition that checks whether an overflow occurs or not, follows the transformation pattern, introduced in Section 4.1.

Listing 1.15 shows the implementation that detects an overflow for signed 32-bit values. To detect the overflow there are multiple conditions needed, to cover all possible overflow cases. Since signed integer values are negative or positive the *Signed32.min_signed* function determines the minimum representable value of the *Signed32* type and the *Signed32.max_signed* function the maximum representable value.

```
Definition safe_mult_signed (a b : Signed32.int) : option Signed32.int :=
  if (a >? 0%signed32) &&
     (b >? 0%signed32) &&
     (a >? (Signed32.max_signed / b))
     then None
     else if (a >? 0%signed32) &&
             (b <? 0) &&
             (a <? (signed32.min_signed / b))
      then None
      else if (a <? 0%signed32) &&
              (b >? 0%signed32) &&
              (a <? (Signed32.min_signed / b))
     then None
     else if (a <? 0%signed32) &&
             (b <? 0%signed32) &&
             (a >? Signed32.max_signed / b)
      then None
      else Some(a*b).
```

Listing 1.15. Definition of the *safe_mult_signed* function in Coq that detects an overflow in the multiplication operation for signed 32 bit values.

Listing 1.16 shows the implementation that detects an overflow in the addition of two signed 32-bit values. As seen in the previous overflow detection implementations the *Signed32.max_signed* function determines the maximum value and the *Signed32.min_signed* function the minimum value.

```
Definition safe_add_signed (a b : Signed32.int ) : option Signed32.int :=
  if (a >? 0%signed32) &&
     (b >? 0%signed32) &&
     (a >? (Signed32.max_signed - b))
     then None
     else if (a <? 0%signed32) &&
             (b <? 0%signed32) &&
             (a <? (Signed32.min_signed -b))
      then None
      else Some (a+b).
```

Listing 1.16. Definition of the *safe_add_signed* function in Gallina that detects an overflow on the multiplication operation for signed 32 bit values.

7.2 Comparison of Arithmetic Integer Operations

In this section, we compare the different arithmetic integer overflow detection implementations proposed in this work regarding their consumed space in LUTs and registers and maximum clock frequency. The foundation for this comparison is the implementation of the traffic light controller, shown in Section 4.1, which was specified in Gallina and synthesized on an FPGA. The comparison is between the specified controller with the basic arithmetic operations and their corresponding overflow detecting operations, as seen in Table 1.

Table 1. Evaluation by comparing the consumed space in LUTs and registers and the maximum clock frequency (F_{max}) for signed 32 and unsigned 32 integer operations used by the traffic light controller, described in Section 4.1. The *basic operation* column contains the values for the basic arithmetic operations, while the *overflow detection* column contains the values for the arithmetic operations introduced in Section 4.1 and Section 7.1.

	basic operation		**overflow detection**	
arithmetic operation	LUTs / Registers	F_{max}	LUTs / Registers	F_{max}
unsigned multiplication	92 / 36	72.20 MHz	670 / 36	65.51 MHz
unsigned addition	81 / 36	111.76 MHz	112 / 36	109.57 MHz
signed multiplication	112 / 36	68.19 MHz	122 / 36	68.84 MHz
signed addition	81 / 36	119.82 MHz	148 / 36	109.24 MHz

Consumed space and maximum clock frequency synthesized for the Cyclone V family using the Quartus Prime tool chain version 18.1.0.

The values in Table 1 cannot necessarily be seen as fixed values, but as a relation between the synthesized traffic light controller specification that uses the basic arithmetic integer operations and the one using the detecting overflow operations. The concrete values highly depend on the FPGA a design is synthesized for. FPGAs are often optimized for a certain purpose, e.g. speed or larger space. As seen in the table above, the consumed space in the form of LUTs and registers differs slightly, except for the unsigned multiplication operation, which we discuss in a moment. The same goes for the maximum clock frequency that has a maximum of 10 MHz for the signed addition operation.

The overflow detecting unsigned multiplication operation has a significantly larger amount of LUTs, as during the synthesis process the *lpm divide* megafunction is used. Megafunctions are programmable logic devices (PLD) that describe a certain functionality, e.g. integer multiplication. These functional blocks are ready-made, pre-tested and augment hardware designs so the functionality has not to be implemented again. For unsigned values Quartus Prime includes the *lpm divide* block automatically while for signed integer division it does not. This decision is based on the analysis of the RTL code. For this reason, the amount of LUTs is significantly higher.

8 Discussion

In this section we discuss the proposed overflow detection pattern and the results of the evaluation.

The detection pattern, shown in Section 6, leads to arithmetic functions that are no longer closed, since the type of the input values is no longer the type of the output values. This prevents the cascading of these operations, which is possible with their corresponding basic arithmetic operations. We addressed this issue by providing an *option* monad, as seen in Section 6.1. This monad closes the operations implementing the proposed arithmetic overflow detection pattern. The closure of functions enables the cascading of those operations which results in the description of more complex calculations similar to the cascading of basic arithmetic operations.

According to the results of the evaluation, the impact on the speed and consumed space by replacing the basic integer arithmetic operations with the corresponding ones that detects the overflow depends on the used operation and the integer type. The difference between the basic arithmetic operations and their corresponding overflow detecting operations for unsigned addition and signed multiplication is even negligible.

In general, this opens a trade-off between safety oriented and performance oriented hardware designs. The additional overflow checks clearly have an impact either on the consumed space or on the maximum clock frequency. But, it depends on the concrete hardware design whether the safety aspect is important enough to except this impact or not. This might not always be the case and the concrete values regarding the speed and consumed space highly depends on the chosen FPGA. Note that in larger hardware designs the arithmetic integer operations represent only a small part of the entire functionality. The impact of the overflow detecting arithmetic operations compared with their corresponding basic arithmetic operations become negligible.

In general, our discussion shows that the overflow detection pattern proposed in this work is applicable. The maximum frequency and the consumed space for the overflow detecting arithmetic operations are slightly slower or even negligible. The only exception is the overflow detecting unsigned multiplication operation, but the Quartus Prime synthesis tool chooses to use the *lpm divide* megafunction automatically during the synthesis process which was omitted for the other operations. But, even in this case, the difference regarding the maximum clock frequency is only slightly slower than in the other implementations used for the evaluation.

9 Conclusion

In this work, the *semantic gap* between the infinite integer types of a SysML/OCL specification and the finite integer types of SystemC was addressed. The issue of this *semantic gap* might lead to arithmetic overflows in the SystemC model which are unknown in the specification, as explained in Section 2.2. This gap motivates our work, and we addressed it by the proposal of an alternative approach which extends the work [5] already published by the authors.

We use the proof assistant Coq [4, 10] in combination with the CompCert integer library [19] to close this gap. The CompCert Integer library describes both signed and unsigned finite integer types of arbitrary sizes as *dependent types* [17, 7]. We utilized this library to describe finite integer types in Coq. This description enables the specification of arithmetic integer operations that verifiable detect overflows, as described in Section 4. These descriptions result in a generalizable pattern for detecting overflows in arithmetic integer operations.

Furthermore, we provide a method to close the functions that implements the proposed detection pattern, as described in Section 6. This allows the cascading of operations implementing the proposed overflow detection pattern, analog to their corresponding basic arithmetic operations to describe more complex calculations.

We evaluated the proposed overflow detection pattern by comparing basic arithmetic operations with their corresponding overflow detecting operations in terms of their maximum clock frequency and consumed space. These values were gathered from an FPGA synthesis process, explained in Section 7, which uses the synthesize process introduced in this work [6]. This evaluation opens a trade-off between safety oriented and performance oriented hardware designs, as additional safety checks clearly have an impact on the consumed space and maximum clock frequency, but the impact is sporadically negligible. For this reason, we evaluated the proposed approach to address the *semantic gap* between infinite and finite integer types as promising.

References

[1] Accellera. “Accellera Systems Initiative Inc SystemC Synthesizable Subset.” In: Version 1.5.7 (2016).

[2] Guido Arnout. “SystemC standard.” In: *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2000, pp. 573–578.

[3] *CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell.* 2010, pp. 714–721.

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

[5] Fritjof Bornebusch et al. “Integer Overflow Detection in Hardware Designs at the Specification Level.” In: *8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2020.

[6] Fritjof Bornebusch et al. “Towards Automatic Hardware Synthesis from Formal Specification to Implementation.” In: *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2020.

[7] Edwin Brady, James McKinna, and Kevin Hammond. “Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types.” In: *Trends in Functional Programming (TFP)*. 2007, pp. 159–176.

[8] Achim D. Brucker and Burkhart Wolff. *The HOL-OCL Book.* USenglish. Tech. rep. 525. ETH Zurich, 2006.

[9] Jordi Cabot, Robert Clarisó, and Daniel Riera. “Verification of UML/OCL Class Diagrams using Constraint Programming.” In: *First International Conference on Software Testing Verification and Validation, ICST*. 2008, pp. 73–80.

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

[11] Zack Coker and Munawar Hafiz. “Program transformations to fix C integers.” In: *35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013*. 2013, pp. 792–801.

[12] Patrick Cousot. “Formal Verification by Abstract Interpretation.” In: *NASA Formal Methods - International Symposium, NFM*. 2012, pp. 3–7.

[13] Patrick Cousot et al. “The ASTREÉ Analyzer.” In: *European Symposium on Programming*. 2005, pp. 21–30.

[14] Pascal Cuoq et al. “Frama-C - A Software Analysis Perspective.” In: *International Conference on Software Engineering and Formal Methods*. 2012, pp. 233–247.

[15] Will Dietz et al. “Understanding Integer Overflow in C/C++.” In: *ACM Trans. Softw. Eng. Methodol.* 25.1 (2015).

[16] Manuel Fähndrich and Francesco Logozzo. “Static Contract Checking with Abstract Interpretation.” In: *International Conference on Formal Verification of Object-Oriented Software*. 2010, pp. 10–30.

[17] F. K. Hanna and Neil Daeche. “Dependent types and formal synthesis.” In: 1992.

[18] Jan Kuper, Christiaan Baaij, and Matthijs Kooijman. “Exercises in Architecture Specification Using CλaSH.” In: *Forum on Specification and Design Languages (FDL)*. 2010.

[19] Xavier Leroy et al. “CompCert – A Formally Verified Optimizing Compiler.” In: *Embedded Real Time Software and Systems (ERTS)*. 2016.

[20] George C. Necula et al. “CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs.” In: *European Joint Conferences on Theorey and & Practice of Software*. 2002, pp. 213–228.

[21] OMG. “Object Management Group Object Constraint Language (OCL).” In: Version 2.4 (2014).

[22] OMG. “Open Management Group System Modeling Language (SysML).” In: Version 1.6 (2019).

[23] Nils Przigoda, Robert Wille, and Rolf Drechsler. “Analyzing Inconsistencies in UML/OCL Models.” In: *Journal of Circuits, Systems, and Computers* 25.3 (2016).

[24] Mathias Soeken et al. “Verifying UML/OCL models using Boolean satisfiability.” In: *Design, Automation and Test in Europe (DATE)*. 2010, pp. 1341–1344.

[25] Andrés Takach. “High-Level Synthesis: Status, Trends, and Future Directions.” In: *IEEE Design & Test* 33.3 (2016), pp. 116–124.

[26] Philip Wadler. “Monads for Functional Programming.” In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*. Ed. by Johan Jeuring and Erik Meijer. Vol. 925. Lecture Notes in Computer Science. 1995, pp. 24–52.

[27] Tim Weilkiens. *Systems engineering with SysML / UML - modeling, analysis, design*. Morgan Kaufmann, 2007.