Software & Systems Modeling (2019) 18:1795–1807
https://doi.org/10.1007/s10270-018-0697-y

SPECIAL SECTION PAPER

Handling index-out-of-bounds in safety-critical embedded C code using model-based development

Gunter Blache[1]

Received: 15 March 2017 / Revised: 18 July 2018 / Accepted: 21 September 2018 / Published online: 9 October 2018

Abstract
Embedded C code for safety critical systems faces some substantial challenges: like every other embedded SW code it must be efficient in terms of code size, data size and execution time, but it must also behave safely under all circumstances, without a user or operator who could handle the errors. One kind of problem is array accesses where the index is outside the specified value range. The C language does not specify the behaviour in such cases, which clearly violates the requirements for safe code. In this paper, the approach of the model-based development tool "ASCET" is explained, and the experiences of three case studies that describe the adoption of index protection by the users are presented.

Keywords Domain-specific languages · Functional safety · Software adaptation · Embedded software · Automotive engineering

1 Introduction

Tools for model-based software development have existed for several years now, and have been widely adopted in the industry for some domains. In this paper, the topic is the adoption of a new feature in the model-based development tool "ASCET" [7], which is mainly used in the automotive industry for the development of engine control algorithms and braking assistance (ABS, ESP). The introduction continues to describe the relevant properties of the tool, the concepts of functional safety and embedded systems. The second section discusses how the problem of index-out-of-bounds can be handled, and the third section discusses some related work. The following three sections describe the case studies, with a conclusion in the last section.

This paper is based on a conference submission [3]. It has been enhanced in the areas of the introduction and the related research, and a third case study has been added. Please also note that the handling of the index-out-of-bounds problem is described to motivate the suitability for the described purpose. It is not innovative with respect to other research. The main contribution of the paper is the three case studies, highlighting the issues that can arise in a real-world application.

1.1 Model-based development with ASCET

The modelling language of ASCET is designed to support the engineering of control algorithms in the automotive domain, where C is still the most frequently used language. It avoids many of the problems of the C language, like pointer arithmetics, operator precedence and uninitialized data. It also adds some special features that are important for the domain, like arithmetic of fixed-point integer values, special data structures to hold approximations of functions and a memory model suitable for embedded microcontrollers. The language contains several kinds of specification, including data-flow-oriented diagrams, textual representations and state machines.

The typical user is a control engineer who has limited knowledge about the technical details of embedded systems. The representation of control algorithms as data-flow-oriented diagrams is suitable for the domain and also used during education and in other tools.

The development process typically starts with a model that represents the control algorithm and is calculated with floating point values. It is run on the local PC to verify the correct behaviour. After that, the model is handed over to a software specialist who adds annotations to make the model

Communicated by Dr. Jörg Kienzle and Dr. Alexander Pretschner.

✉ Gunter Blache
gunter.blache@etas.com

[1] ETAS GmbH, Stuttgart, Germany

executable in an embedded device: fixed-point representations for the arithmetic values, data-consistency and real-time scheduling and assigning the data to the appropriate sections in the memory (like FLASH and RAM, where multiple sections may exist with different capacity and access latencies). Using the terminology of the Object Management Group, this is the transformation from a platform-independent to a platform-specific model [14]. The annotated model is then used to generate the code for an embedded device. The code is usually stored in a repository together with the model and later retrieved to build an executable. In most organizations, the application is split into multiple models and integration is done on the level of the C code.

The development of ASCET started in 1995, and the first embedded control unit (ECU) containing generated code went into production in 2000. At the time of writing, about 7500 developers are using ASCET, and an estimated 450 million ECUs are on the roads and rails that use ASCET generated code.

For the purpose of this paper, two model elements need to be described in more detail: system constants and arrays.

System constants are scalar elements of a Boolean, enumeration or integer type and used to model variants that are resolved by the compiler. Common examples are the presence or absence of a turbo charger, the activation of a customer-specific feature or the number of cylinders in an engine. In the generated C code, system constants appear as preprocessor `#define` directives.

Arrays in ASCET are very similar to other C-like languages. The index range can be of fixed size, or set by a system constant. (The array is of variant size.) In the latter case, the actual size of the array is not known before compile time. The size of all ASCET arrays is, however, known at compile time at the latest, enabling static allocation. It is not possible to calculate the size of ASCET arrays at runtime or change the size of an array programmatically.

In addition, regular integer variables also exist to index an array, e.g. as loop variables or counters. All integer variables also have a defined value range that may be a subset of the data type range.

1.2 Concepts of functional safety

In safety-critical systems, a failure could potentially result in injury or death of persons. Such systems exist in multiple domains, like medical, aviation, railway and automotive. They must be developed according to the state of practice, which is defined by standards like the general ISO/IEC61508 "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems" [9] or the specific ISO 26262 "Road vehicles - Functional safety" [11] for the automotive industry.

The standards categorize the components into different levels depending on their criticality and recommend engineering methods throughout the development cycle to achieve safety. The safety integrity levels (SIL) for ISO61508 reflect probability and consequences, and the automotive safety integrity levels (ASIL) for ISO26262 reflect probability, severity and controllability. The reason for this difference is the fact that most critical systems are operated by professionals that can be educated and are expected to follow established rules and procedures. For the automotive domain, the average driver of a car can be any type of person. To reflect this, the aspect of controllability has been added that describes how likely it will be that the conductor stays in control of the vehicle when a certain failure occurs.

What also strikes the automotive industry more than other domains is the cost pressure due to the high volume of cars sold, and this also applies to ECUs, where savings of a few Euro per unit (e.g. using less memory or a weaker micro controller) can mean several million Euros over the production life time of an ECU. For the software part, this means that memory footprint and execution time should be optimized without sacrificing safety.

The relevant part for the topic of this paper is Part 6 Section 8.4.4 of ISO26262 [11], which addresses the implementation of a software unit. Among the properties to be achieved, there is robustness, which is defined as "the ability to provide safe behaviour at boundaries" (Part 1 Section 1.100). Although this is certainly intended to be understood in a broad context, it can be taken quite literally when talking about array accesses and the boundaries of the index.

The authors of these standards were aware that the programming languages used in practice were not designed for high-integrity software development and therefore recommend to use a "safe subset" of such languages. A popular one is defined by the rules in MISRA-C [1], which in this context formulates directive 4.1 to minimize array bound errors. In general, the MISRA-C guidelines restrict the usage of language features which have an undefined, unspecified or implementation-defined behaviour, and also the features that are difficult to understand or negatively affect readability. Examples would be the evaluation order of side effects, the size of an integer (16 or 32 bit), integer promotion and conversion and the comma operator.

Another topic is related to the integration of multiple components on one ECU: the operation of one component must not be degraded through the interference with other components. This interference can take place in different ways, for example, consuming too much CPU time, blocking critical resources or writing into the memory of a different component. The standard therefore requires the isolation of components from each other. This isolation of the different sections of memory can be achieved by a suitable configuration of the memory management unit (MMU). However,

small microcontrollers do not include a MMU, or the cost of reconfiguration when transferring control from one component to another can be prohibitive for context switch latencies needed in a multi-threaded hard real-time environment. The other option is to avoid the problem by construction, which means a restricted use of pointers and a safe way of accessing the contents of an array (which technically is also pointer arithmetic in C).

For code generated from ASCET models, pointer arithmetic is not used, because the modelling language is restricted to only access array elements through an index, and this is also the only way that is used in the generated code. The safety of an array access can therefore be reduced to ensure that the index is in the valid range. In addition, access through pointers is safe by construction, because all pointers are initialized in the model to refer to a statically allocated object (either on definition of the pointer, or using a data-flow analysis to ensure initialization before first usage). To support handling of arrays of different sizes, fat array references are used that also carry the size of the referenced array to ensure a safe access. Other kinds of interferences are not addressed by the modelling language and need to be taken care of using other methods (e.g. schedulability analysis using worst-case execution times [8,19] or safe communication [15]).

It should also be noted that the ISO standard does not require "correct" behaviour, e.g. in a mathematical sense. Instead, it emphasizes that the safety requirements to prevent hazards must be documented, complete and verified, e.g. using coding guidelines, static analysis, manual review or testing (see [2,16]). The handling of exceptional cases can therefore take place in any way, as long as it is suitable to be verified against the safety requirements, which at least means that it is defined and deterministic.

1.3 Restrictions for embedded systems

Embedded systems come in various flavours, and the ECUs referred to in this paper are characterized as follows: they are real-time applications with functions that must run frequently (with inter-arrival times measured in milliseconds) and have hard deadlines (equal to the inter-arrival period or less). In addition, there are the usual resource constrains with respect to memory and performance. This can also be seen later, as the large majority of index variables have an 8-bit data type.

One consequence is that heap storage allocation is not used, because memory allocation introduces non-determinism that is difficult to account for in offline real-time analyses, and handling of an out-of-memory error is in conflict with the real-time requirements. Instead, all data are allocated statically or on the stack. While this is sufficient for the target domains, this is too restrictive for more recent technologies like processing of radar or video data, or entertainment systems (radio, MP3 players, DVD players).

2 Handling index-out-of-bounds

Accessing an array is nothing new, and various approaches are used to handle the index-out-of-bounds:

1. Make it the user's problem: this approach is taken in C, where the behaviour is simply "undefined" (see e.g. [10] appendix J.2). Expert programmers sometimes tend to exploit the fact that the access is to the "next" memory location, but this also depends on alignment, packing and padding which may change in the future (with a new compiler version or a different microcontroller architecture). In addition, memory may not even exist at these addresses, which could result in a trap. Some of the input/output devices may also be mapped to memory and reading to, or writing from such addresses may have an effect on the complete system, which adds one more failure mode to array accesses.
2. Throw an exception: this approach is taken in Ada, Java and C#. This is also a flavour of the first approach, but with a defined behaviour.
3. Prove that the problem does not exist: perform a code review or run a static analysis to ensure that the index is always inside of the bounds.
4. Handle the errors explicitly, e.g. check the index before each access and implement error handling code.
5. Handle the errors implicitly, e.g. map the invalid index values to valid ones by limitation $\texttt{min}(\texttt{max}(\texttt{i}, 0), \texttt{arraySize} - 1)$.

In the context of embedded C code for safety-critical systems, the first two options are unsuitable: undefined behaviour must be avoided, and exception handling is not possible in C. In addition, throwing an exception adds an edge to the control-flow graph, which can lead to a large increase in testing effort if branch coverage is required.

The other options are suitable and can also be used also in combination, e.g. using the static code analysis to ensure the consistent application of the explicit handling.

The options have an effect on development effort, though. If errors are handled explicitly with a follow-up manual review step, there is a risk of a late finding of a missing protection (which requires to repeat the development cycle once more). Even if static analysis is used, the experts need to qualify the findings of the "potentially unsafe" category and take appropriate action. In any case, the effort needs to be repeated for every change, and as long as humans are involved, there is also the risk of human error.

If errors are handled implicitly by an automated code generator, there is no additional effort required. Users may still want to run static code analysis to protect themselves against tool defects, but the analysis report only needs to be checked that no definitely or potentially unsafe code is detected. The

finding of an unnecessary protection is not critical, as long as the added execution time does not violate the real-time constraints. The number of late findings will be near zero.

It is also possible to handle the errors explicitly, and add an implicit handling if necessary. Again this can also be checked by static code analysis and will have nearly no late findings.

2.1 Index limitation

Limitation on the assignment to integer variables has been a property of the ASCET modelling language since the beginning: the value range of all variables is explicitly defined in the model, and an assigned value is limited to that range.

Regarding array indexing, ASCET up to version 6.1 followed C and used the first approach, placing the responsibility on the user: he needed to add explicit checks as necessary. Since Version 6.2, ASCET supports the combination of the options (4) and (5): the user can handle the error cases explicitly, and the tool can add the index limitation to ensure a valid index.

The reason was the positioning of ASCET as tool to develop software for safety-critical applications, and the array indexing was the last kind of runtime error without tool support. The expected benefits for the user are:

1. Front-loading of checks, avoiding long turnaround times if the defect is detected late. This contributes to work efficiency.
2. Reduced model complexity when explicit handling of the index-out-of-bounds problem can be removed. The models should become more understandable and also more maintainable.
3. In addition to code, documentation is generated from the models that is used in the downstream process of calibrating the software to a particular vehicle model. This documentation should become more readable in the same way as the model becomes more understandable.

By default, all array accesses are protected by adding an index limitation. To reduce the negative impact of the added checks on the code efficiency, an analysis has been added to identify those accesses where no additional protection is required. The analysis is a standard data-flow analysis to recognize the typical model structures used for explicit handling of index problems. It keeps track of the value ranges for each variable (taking the value ranges as specified in the model as a starting point) and uses interval arithmetic on operators. This is sufficient for arrays of fixed size. Additionally, relations that are derived from if and loop conditions are exploited to handle arrays of variant size.

The analysis decides if an array access is always safe, potentially unsafe or definitely unsafe. In the definitely unsafe case, an error is reported. In the potentially unsafe

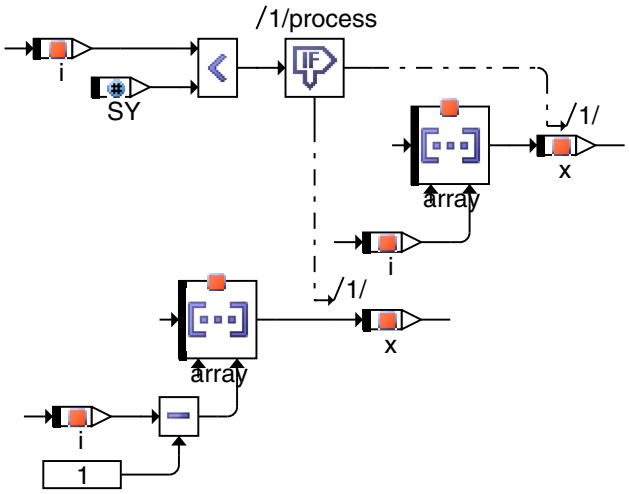

Fig. 1 Example block diagram model

```
1   if (i < SY) {
2       x = array[i];
3   } else {
4       x = array[i-1];
5   }
```

Fig. 2 Example textual model

case, a warning is reported. A separate option enables the generation of a limitation of the index value against the lower and upper bounds.

To enforce the explicit handling by the user, the warning can be promoted to an error, so that code will only be generated if all array accesses are considered safe by the tool. If index limitation is the dominant way of explicit handling, this can be taken care of in the code generator, and the warnings can be used in a review to check if additional handling is required. The safety property of isolation (see Sect. 1.2) is maintained in both cases. Please recall that correctness in a mathematical sense is not required. Instead, the behaviour needs to be deterministic and correct with respect to the safety requirements. If both the index limitation and the data flow analysis are nevertheless considered unsuitable or insufficient, both can be disabled.

As an example, consider Fig. 1. The drawing only shows the behaviour part of the model, since this is the relevant part for the example. The model also contains information that the variable `i` and the system constant `SY` are integer values with a value range of `[0..255]`, initial values of 0 and 10, respectively, and the array is declared to have `SY` elements, with a maximum of 10. The base type of the array and the variable `x` are both floating point types.

The ASCET block diagrams are a data-flow-oriented specification, where each object (like variables, system constants and arrays) and operators are represented as a block with the according pins. Variables and system constants have a read-

pin on the right-hand side, and variables have an additional write-pin on the left-hand side. The less-than-operator has two operand pins on the left-hand side and a result-pin on the right-hand side. For the subtraction operator, the operand pins are on the left and the bottom side, and the result is retrieved from the right-hand pin. The array blocks shown here have a pin on the right-hand side to read an array element, whose index is connected to the left of the two pins on the bottom. The right-hand pin and the right bottom pin can be used to write an array element. The execution order is explicitly defined, and the first statement is the if-statement, which is sequenced as number 1 in the process. Depending on the result of condition, either the then-path on the right-hand pin is followed using the dashed connection, or the else-path at the bottom. In both cases, multiple blocks can be connected, and they are then executed in the specified sequence. The block diagram is semantically equivalent to the textual model shown in Fig. 2.

When analyzing the if-block, the relation `i < SY` is derived from the condition and used in the array access connected to the then-pin on the right-hand side of the if-block. With this information, the access is definitely safe and does not need to be protected.

The opposite situation is observed on the else-pin on the bottom of the if-block: the relation `i >= SY` is derived. But in this case, the variable `i` is not used directly, and the analysis needs to deal with the subtraction. The information about relations is discarded, and interval arithmetic is applied to yield a result interval of `[−1..254]`. This case is considered to be potentially unsafe, and a warning is reported.

We decided to not place a focus on the analysis of loops like heuristics for loop invariants. The loop body is only analyzed for variables that are not assigned (so the information at loop entry can be used), and the loop condition is used locally for the loop body. This was not expected to be a major problem because loops occur only occasionally in control algorithms for real-time systems.

Regarding the typical model structures for explicit handling, it is quite difficult to get good input from customers: for reasons of protection of intellectual property, the models are only shared to a very limited extent with the development team to perform an analysis. Asking questions also has little benefit, because the contact person has no detailed knowledge of what users have implemented in potentially hundreds of models during the last years. In addition, at the time of requirements engineering for a specific version, the customers are still working on the introduction of the previous version and have little immediate motivation to spend what seems to be additional effort.

As can be seen from the above description, the analysis is implemented in a conservative way. The first reason is that one additional limitation is not a critical problem, while a missing limitation is. It is clear that critical defects would undermine the trust in the tool, which will contradict the whole argument about safety. The second reason is that generated code is almost always also reviewed for correctness, and if the analysis is too elaborate, it will be difficult to understand its decisions.

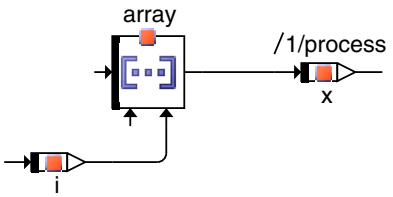

Fig. 3 Simple access to an array

2.2 Variant handling at compile time

Variant handling is an important topic for the domain. There are functional variants, where a case distinction is added to the model and the data structures are not affected. And there are also structural variants, where different data structures must be allocated. Variants are resolved at compile time, because, on the one hand, no unnecessary code and data should be in the final executable, and, on the other hand, it should be possible to select the variant as late as possible, while still using the static allocation.

One example is an array whose size is defined by a system constant. This affects the declaration of the data structures, but also the access to the array. A simple example is shown in Fig. 3. Like in the example from Fig. 1, the variable `i` and the system constant `SY` are integer values with a value range of `[0..255]`, initial values of 0 and 10, respectively, and the array is declared to have `SY` elements, with a maximum of 10. The base type of the array and the variable `x` are both floating point types. The executable named `process` has only one step, which is the storage of the array element at index `i` into the variable `x`.

The generated code (see Fig. 4) is straightforward: the system constant is lazily defined with the default value, the value is checked to be in the allowed interval, the array is defined using the value of `SY` for the number of elements, and the array access contains a limitation. (Since the index cannot be negative, a limitation against the upper bound is sufficient.)

The check of the system constant in line 4 is added for two reasons: to have an early error in case the value is less than 1, instead of waiting for the compiler to produce an error on the array declaration, which may be more difficult to analyze. And to avoid the accidental allocation of a very large array, which will not be detected earlier than at link time for the final executable, if the available memory is exceeded.

Fig. 4 Generated code for an access to a variant array

```
#ifndef SY
#define SY 10
#endif
#if (SY < 1) || (SY > 10)
#error The system constant SY must be between 1 and 10,...
#endif
...
unsigned long array[SY];
...
x = array[((i < SY) ? i : (SY - 1U))];
```

The analysis and code generation for array accesses is designed to provide safe and efficient code by recognizing manual protection in the models. To avoid long turnaround times, potential problems are highlighted as soon as possible (during code generation or compilation). Execution time of the analysis itself is a secondary issue.

3 Related work

3.1 Research

Both static code analysis to detect runtime failures and code optimization in compiler construction have triggered research that can be applied to the index protection problem.

The work of Kolte and Wolfe [12] proposes different optimization strategies to remove unnecessary checks or merge similar checks. The area of application is either a programming language that mandates these checks (e.g. Ada) or where checks are added optionally for added safety (e.g. C, Fortran). The reaction for a failed check is a trap in this case, and no other options are discussed. The different optimization strategies are implemented in a research compiler for Fortran, and experimental results for various benchmark programs are computed.

The proposed algorithm is much more sophisticated than the one presented in this paper, also building on existing work for induction variables to handle loops. The second special strategy is the optimization of weaker checks: when a trap is executed if `i >= 5||i >= 10`, then the second condition can be removed. This is not applicable to the limitation of index values, because the two upper bounds result in two different limited index values.

Widening the scope, Rugina and Rinard [18] not only deal with static detection of array bounds violations and elimination of array bounds checks, but also pointer arithmetic with the presence of dynamic memory allocation and recursion. A prototype implementation has been tested on various benchmark programs for C and Cilk, a parallel version of C.

Since both dynamic memory allocation and recursion are not used in the domain of real-time embedded software, the design of the analysis is overly complex for ASCET. Please recall that one of the design goals is the freedom from defects that lead to missing protections. The support for loops is similarly weak, and the algorithm has the restriction of variables with an unsigned value range.

Buffer overruns are a serious problem for the security of networked applications and can also be detected statically. In Wagner et al. [21], an approach is developed that is applied to string operations in C. It does not include an analysis of the control flow or pointers, but still provides useful results. The analysis is supposed to be carried out during the development to be able to fix the vulnerabilities prior to the deployment of the code. The focus is therefore on the analysis performance, accepting undetected problems.

The described analysis is even more basic than the one implemented in ASCET. Following the argument that few detected vulnerabilities are better than none (in case the complete analysis takes too long to be unusable), undetected problems are accepted, which is not an option for safety-critical applications as outlined earlier.

3.2 Other languages and tools

Beyond research, there are also static code analysis tools in practical use, e.g. the Polyspace Verifier [4]. This tool uses abstract interpretation to analyze C code for the presence of runtime errors, including array accesses. The analysis is very elaborate, and this tool is also used to analyze ASCET generated code. The output annotates the C code with the analysis result: safe, unsafe, unreachable or potentially unsafe. Findings of the last category obviously need to be further analyzed. However, code may be unreachable due to defensive code generation (uncritical) or due to errors in the program logic, also leading to follow-up analysis and probably changes to the model. Due to the conservative design of the index protection analysis in ASCET, unreachable code is reported by the checker, resulting in a conflict between understandable code and minimizing effort of follow-up analysis.

Another modelling language that is also targeted at safety-critical embedded applications is Lustre. This language is supported by the tool SCADE developed by Esterel and mainly used in the aviation and railway industry [5], [6]. In V4 of the language, the support for arrays has been added as described in the PhD thesis [17]. The language is not only designed to generate models for special hardware like FPGA

and embedded C code, but also to enable formal verification. To this end, the language has been revised in V6 to simplify the verification of models with arrays [13].

The arrays in Lustre are restricted to have a fixed size that is known at the time the model is compiled for some target platform. Indexing of an array is also restricted to an index expression with a value that is constant, so that the question of index-out-of-bounds can be decided statically. To compensate for this limitation, the language lifts all operators as element-wise operations on arrays, and also introduces new operators to aggregate all array elements, etc. This eliminates the need for loops to a large extent, and the access to array elements is correct by construction.

The idea to lift operators to array types could also be applied to ASCET, and the strategies for efficient code generation to eliminate unnecessary array copies would also be beneficial. The extension to arrays of a system constant size is straightforward. It could, however, be difficult to present the concept of higher-order functions (like an operation on an array that takes an initial value and an operator for the array elements that can be used to compute min, max, sum, etc., of all array elements) to the existing user base in a way that is easy to understand and use.

The most frequently used tool in this domain is Simulink developed by The Mathworks [20]. The focus lies on the simulation, but code generation is also supported. The models support multi-dimensional values, and a "Selector" block exists to extract a single value or a subset of the values. In the case of an index violation, the simulation stops with an error message. While this is a sensible reaction for simulation, this is usually not desired for an executable in an embedded device.

The two code generators "Simulink Coder" and "Embedded Coder" that are also sold from The Mathworks both do not generate any index protection. So if a model is extensively simulated without an error, it can be expected to not have an array index that is out-of-bounds. If additional safety is required, static code analysis and explicit handling (e.g. using saturating blocks) can be applied.

In general, the value of a simulation is reduced if the results are not identical to the generated code on the ECU. In this case, the different behaviour is obvious and therefore not a problem in practice. ASCET does not have a separate simulation engine: when running a model on the local PC, code is generated, compiled and then executed. In this case, only compiler defects or different hardwares (like floating point units) lead to differences in behaviour between simulation and the ECU.

3.3 Summary

The index-out-of-bound problem can be treated in a number of ways, depending on the intended use case. Analysis can be fast and inaccurate or very elaborate. Modelling languages can be designed to avoid this problem completely or leave it for the user to find a solution.

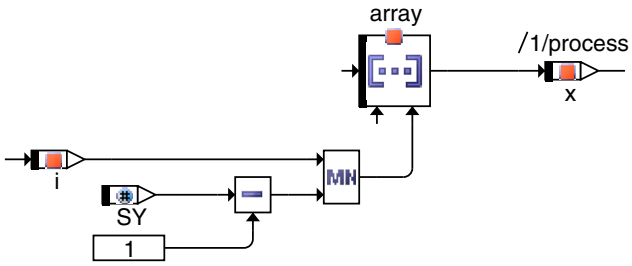

Fig. 5 Index protection in the model

```
1   _t1 = (sint16)SY - 1;
2   _t1 = (((sint16)i <= _t1) ?
3         (sint16)i : _t1);
4   if (_t1 >= 0)
5   {
6     if (_t1 < (sint16)SY)
7     {
8       _t2 = _t1;
9     }
10    else
11    {
12      _t1uint8 = SY - 1U;
13      _t2 = (sint16)_t1uint8;
14    }
15    _t1 = _t2;
16  }
17  else
18  {
19    _t1 = 0;
20  }
21  x = array[_t1];
```

Fig. 6 Generated code for index protection in the model

4 Case Study I

When planning the introduction of ASCET 6.2 in one customer organization, the key users agreed that the array index protection would improve the safety of the generated code and decided to enable it. After the testing phase has been finished successfully and the new version was made available to the end-users, a lot of negative feedback about increased code size and execution time was reported from the end-users.

4.1 Example: index protection in the model

Since the users have already worked for several years without the index limitation during code generation, the models were full of explicit handling of index values. One typical way is very similar to the generated limitation, namely the usage of a *min*-operator in Fig. 5. This was not recognized

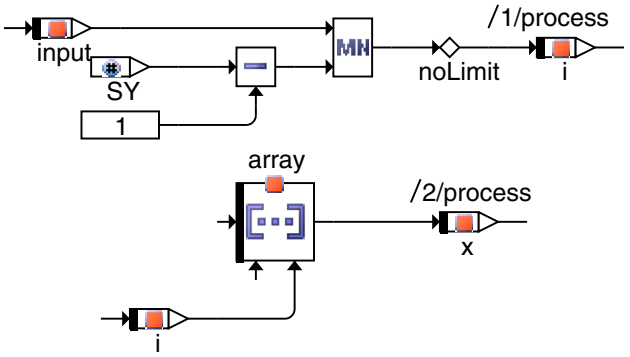

Fig. 7 Manually suppressed limitation in the model

```
1   _t1 = (sint16)SY - 1;
2   _t1 = (((sint16)_input <= _t1) ?
3       (sint16)_input : _t1);
4   i = (uint8)_t1;
5   x = array[((i < SY) ? i : (SY - 1U))];
```

Fig. 8 Generated code for manually suppressed limitation

by the analysis, resulting in a duplicate limitation, which is certainly safe, but inefficient, see the code in Fig. 6.

The result is indeed surprisingly large compared to the model, for a number of reasons. Both the system constants `SY` and the index variable `i` are declared to have an interval of `[0..255]`. The interval of `SY-1` is therefore `[−1..254]` and requires a signed type (line 1). The result of the *min*-operator also has an interval of `[−1..254]`, which requires a limitation against both the lower bound, which is 0 (lines 4, 19), and the upper bound, which is `SY-1` (lines 6, 12, 13). The upper bound of an array is ensured to be positive. Due to type casting according to MISRA-C:2012 rule 10.8 [1], a temporary variable is inserted on the assignment of an unsigned value to a signed variable (line 12). This extra assignment enforces the cascaded structure of `if`-statements (lines 4, 6) that could otherwise be generated as conditional operators with a comma operator. For reasons of readability, the use of the comma operator is discouraged by MISRA-C:2012 rule 12.3 (see [1]).

The root cause of this problem was an incomplete requirement, which did not list the *min*-operator as a possible explicit protection. It was straightforward to add this case of explicit handling to the analysis.

4.2 Example: Manually suppressed limitation

Limitation is not only applied for array index values, but more prominently on the assignment to integer variables to avoid overflows (in this case from − 1 to 255). Users, especially if they have a strong C background, tend to tweak the model and want to avoid unnecessary limitations also in this case. This is achieved by the special noLimit-operator[1] as shown in Fig. 7. As outlined in the previous example, the value of the `min`-expression may be negative, which would lead to a limitation on the assignment. The effect of the noLimit-operator can be seen in line 4 of Fig. 8: the value is directly assigned, without a further limitation.

It should be noted that a model like this is fragile: if the array size is changed to depend on a different system constant, the noLimit-operator may mask an overflow if `SY` can then become 0 again. It is a case-by-case decision of the user that depends on the outlined risk, the available measures of verification, and the expected benefit (which may be relevant if the code is executed very frequently, e.g. once every 1 ms).

Since the analysis was designed to be conservative, the information gathered on the operand of the special noLimit-operator is not used. The effect is that the index in the array access is again limited, even though protection has already been done explicitly in the model.

The removal of the noLimit-operator would enable the analysis to omit the index protection, but at the same time add a limitation on the assignment to `i`. To avoid this, the declared interval of `SY` would need to be changed to `[1..255]`, since an array cannot have 0 elements. This will improve the result interval of the `min`-operator to `[0..254]` and not trigger a limitation on the assignment to `i`. However, the system constants are (as indicated by the name) global to the system, and a change in the properties therefore has a global effect, which consequently is prohibitively expensive.

The other option, which is more likely to be accepted, would be to improve the analysis to infer that system constants used as array sized cannot be 0 (which is safe due to the added check as outlined in Sect. 2.2), and that the information from the operand of a noLimit-operator can be further used, if the operator does not result in an overflow.

4.3 Discussion

The generated code is clearly more complex than expected (especially in Fig. 6), which affects readability and possibly efficiency in a negative way.

Different opinions exist if readability is a concern for generated code. Drawing an analogy, typical C programmers do not read the intermediate assembly language output. On the other side, the transition towards model-based development is not complete yet. The quality of the generated code is ready for production use, and it is therefore not changed manually before being compiled. But organizations check the generated code into the source code repositories instead of re-generating it from the models and they apply methods developed for manual coding like code reviews and coverage analysis. In this setting, readability is an important property.

[1] The operator is called “implementation cast” in the product

	No protection	With protection	Increase
C16x	18 Bytes	32 Bytes	70 %
MPC56x	48 Bytes	84 Bytes	75 %

Fig. 9 Comparisons of code size

The effects on code efficiency are not so easy to estimate. During compilation, the system constant is replaced with a concrete value. A data flow analysis can then propagate this value and enable the optimizations to reduce the code. For embedded systems, the compilers may not include the most recent technologies, and developers of safety-critical systems usually stick with the default configuration to protect themselves from compiler defects, which may exclude the more complex optimizations. This means that the compiled code may be less efficient than the result of state-of-the-art high-performance compilers. Taking the example in Fig. 6, the effect is a lot less dramatic than in the C code and basically what would be expected for a duplicate limitation. Taking compilers for a 16-bit target (Infineon C16x,Tasking VX 2.2 compiler released in 2005) and a 32-bit target (Freescale MPC56x, Windriver 5.6.0 compiler released in 2007), it can be seen in Fig. 9 that the extra code is not more expensive than the limitation in the model for the embedded targets, although it takes a lot more lines of code. The effect therefore is real, but not as big as it may be indicated by the C code.

The issues were discussed between two groups within the customer organization: the advocates for safety and representatives of the user community. The decision was to disable the index limitation for the following reasons:

- The index limitation was already part of the models, and further static analysis of the generated C code established that the limitations were sufficient. The correctness of the models was also ensured by reviews and testing.
- The users did not want to spend effort to change the models to achieve the efficiency they already had, because the analysis was not sufficiently thorough. In contrast, they argued that the tool is there to make the life of the users easier. Since each change must also be verified through reviews, testing etc., this is a strong argument.
- The perceived negative impact on the efficiency required an either-or decision. It was not feasible to enable the limitation and adapt the models over time (e.g. when other changes were made anyway).

Unfortunately, these discussions were conducted internally, without involvement of experts from the tool vendor. There were no opportunities to correct the perception of unacceptably high loss of efficiency or to suggest supporting measures. It may well be that code efficiency is only a secondary aspect, and the real issue is code change, since the switch from one version to another (or a configuration change) is verified by code comparison. In this approach, the number of code changes directly translates into effort and cost of change. What makes matters worse is the decision to only have one tool version in service. While this minimizes the administration effort, it requires all projects to switch the tool version at the same time.

Independent of the actual root cause of the rejection, the analysis should be improved to address the areas where the biggest measurable impact on efficiency and/or change in generated code is observed to increase the probability of feature adoption. Taking one step back, it could also be beneficial to offer a model refactoring to remove unnecessary manual limitations. This would address the double-limitation problem and also provide the benefit of reduced model complexity.

5 Case Study II

A second customer has a different workflow. The ECU software is split into multiple models according to the functional structure of the software to facilitate distributed development. The interface between the models consists of data (like vehicle speed or motor temperature). Function calls across model boundaries are not supported. The compatibility of the data definitions (like data types) is checked outside of ASCET, where an overview of all models is available.

To perform global optimizations of the data structures, a special description file is used. The code and the data structure description is generated separately for each model and stored in a repository. When the configuration of an ECU is defined (i.e. which models are part of the build), a custom tool takes the descriptions of all models that are used in an ECU and generates globally optimized data structures. The optimizations, for example, omit data that are unused in the selected variant and reduce the data copies to ensure data consistency.

This extra step of data structure generation was also used to add features that could also be part of ASCET, but was implemented in the custom tool due to schedule and priority conflicts. One enhancement included the ability to annotate arrays with an additional size that overrides the information in the model for the data structures description. The annotation was more expressive than the product solution in version 6.2, in particular allowing the size to be calculated from constants and system constants, e.g. redefining the size for an array with 10 elements as `SY * 2 - 1`.

This customer also upgraded from an earlier version to V6.2 and investigated if index limitation could be suitable, but encountered a problem: the code generator would use the size as specified in the model for limitation, while the actual data structures would have a different size, effectively limiting the array index against the wrong upper bound. In

```
// array is declared with 10 elements
// in the model
unsigned long array[SY*2 - 1];
...
// upper bound is wrong
x = array[(i < 10) ? i : 9];
```

Fig. 10 Mismatch between declared size and limit bound

```
#define SY2 ((SY)*2 - 1)
...
#ifndef SY2
#define SY2 10
#endif
...
unsigned long array[SY2];
...
x = array[(i < SY2) ? i : (SY2 - 1)];
```

Fig. 11 Calculated value of the system constant

the example (see Fig. 10), the array has been declared with 10 elements in the model, but the annotation specifies that the size is computed from the value of a system constant. Since the index analysis has no information about this annotation, it introduces a limitation against the upper bound as specified in the model, which may be too small or too big.

5.1 Discussion

In contrast to the first case study, it was not a list of more or less frequent cases that were not handled as expected, but one substantial point that made the usage of automatically generated limitations difficult to achieve. The root cause is not related to the index analysis itself, but the unavailability of the customer-specific annotation for the analysis.

The most simple solution is to make sure that the array size in the model is not too small and enable the limitation. This will give a correct lower bound, but an upper bound that is too big. Compared to the previous setting, this is safer for the lower bound, while not unsafer for the upper bound. Taking the example in Fig. 10, the code would be correct as long as `SY` < 5. If `SY` represents the number of cylinders in an engine, and the model is only developed for up to 4 cylinders, this is fine. But obviously, the benefit is only half of what it could be and the handling of the additional failure would require additional effort.

The next-best solution is to introduce a dummy system constant for the size, whose value is determined at compile time. This would effectively mean to exchange an annotation about the array size with an annotation about the calculation of a system constant value. From a practical perspective, the definition can be added as a compiler switch or added to a header file, due to the lazy definition of the system constant value that already exists in the generated code (see Fig. 11). The customer-specific annotations for the array size are sufficiently formal to enable an automated migration. This automation is not only nice to have to save the manual effort, it is definitely required to avoid the inevitable errors that occur during manual tasks.

While the two described solutions fit into the current setup and can be implemented without a product change by the customer himself in the short term, a product solution certainly is the preferred option. The impediment here is that the expressiveness for the variant size of an array is too limited, which was caused by the ignorance of this specific customer enhancement when the variant arrays were introduced.

There is also a tension between a product with tightly controlled features, and open interfaces that enable customers to implement their own requests without coordination with the core product development. Once such a customer-specific solution exists, it becomes difficult to change it into a product feature if there is no clear benefit, because this creates additional implementation and migration effort. One anti-pattern we observed in this area is a response along the line of "I know what I am doing" from the side of the customer: indeed they know what they are doing and know the boundary conditions, etc. However, it is likely that customer organization looses this knowledge over time, since customizing tools is not the core business. So the question is not if the customer knows what he is doing now, but how long he will continue to know, and how this compares to the expected usage time of the tool. The time span of knowing may be very small if an external consultant is given the task of customization, or be in the area of three to five years for employees.

6 Case study III

A third customer also agreed that index limitation is a suitable way to avoid the runtime errors and decided to enable the index limitation when migrating to ASCET V6.2. The overall situation was different to the other two customers: the models were built in a way that no index-out-bounds could occur, e.g. using arrays of fixed size with constant index values, and there was also no follow-up tool chain that interacts with the index protection. The models therefore did not contain any explicit handling to avoid runtime errors. Still this customer was very sensitive to the actual efficiency of the generated code, mainly in terms of execution time.

When activating the index limitation, most of the array accesses therefore did not need any additional protection. However, this customer also encountered the problem of unnecessary limitations, e.g. in the case of special loops. One example is an array that is effectively a matrix with a length and width that can be adjusted to the current configuration of the software (see Fig. 12). The array has a fixed size of 50,

```
1  sum=0;
2  idx=0;
3  for (cnt = 0; cnt < (length * width); cnt = cnt + 1)
4  {
5          sum = sum+array[idx];
6          idx = idx + 1;
7  }
8  return sum;
```

Fig. 12 Example loop for a manual workaround

and the variable `idx` is consequently defined with a value range of `[0..49]`, so that the array access is considered to be safe by the index analysis. The increment on `idx` then involves a limitation, which is reported by a code generator diagnostic as shown in Fig. 13.

The alternative of directly using the variable `cnt` to access the array would have the same semantics, since the values of `cnt` and `idx` are identical at the point of the array access. However, the loop condition has no apparent relation to the array size, so the index analysis would consider the array access as potentially unsafe, add a limitation to the index and report a warning.

By introducing the variable `idx`, the behaviour was preserved, but the diagnostics changed from a warning about an index limitation to an info message about a limitation on assignment. In this way, the model did comply with the modelling guidelines that index limitation should not be necessary, and the new info message could be handled in the existing review process.

The unnecessary index limitations as described in the example were very rare, so the overall impact on the efficiency was negligible. The benefit of the added safety did therefore outweigh the cost of an exceptionally added limitation.

6.1 Discussion

It is a common practice to analyze the compiler diagnostics and either change the source code to avoid a warning or to decide that a specific warning is not relevant for a specific project. Likewise, the diagnostics of a code generator are analyzed. In this case, the warning that an index limitation needs to be applied was not tolerated, because the accesses are safe by design, and such a warning would indicate a violation of that design. To handle the false positive warnings, the model was changed to instead produce the information about the limitation of a regular integer value. For this information, an existing process was in place to separate the intentional warnings from the accidental ones (which would result in increased run time) and could be reused.

This highlights that a successful adoption of a new feature not only depends on the functional behaviour, but also on pragmatic properties like diagnostic information that can be used to detect the places that need manual intervention.

In contrast to the case study II, this customer tried to use the tool with as little customization as possible, recognizing that customizations need to be maintained for a potentially long time. This made it easier to use the standard mechanisms for migration. Unlike the first case study, the switch to a new tool version was synchronized with the start of a new development project. The disadvantage is that multiple versions of the tool need to be maintained for different projects, but the clear advantage is that running projects do not need to cope with the impact of a version switch, while new projects can benefit from the most recent feature set.

7 Conclusion

With ASCET Version 6.2, we introduced handling the problem of index values outside the array boundaries by limitation of the index value. Although the approach of limitation is appropriate for the given domain, two major customer organizations did not adopt the new feature. We have identified two root causes for the failed adoption of index protection:

- The analysis failed to recognize frequently used explicit protections in the customer models. This leads to double limitation and therefore inefficient code and/or code changes that need to be verified manually.
- The limitation was not correct in combination with customer-specific adaptions of the data structure generation that extended to capabilities of the core product.

On the other hand, there are positive things. Even though the analysis did not cover all typical cases of customer models, its design was suitable for the job: it is a common understanding of all users that a missing limitation is a critical defect, and we have not observed such a problem at all. In addition, the initially missing cases could be added to the analysis without design changes.

It also was a good decision to make the generation of the limitation optional. If it would have been mandatory, this could have made it impossible for the customers to adopt a newer version. Adding new features as options with a default behaviour as in the older versions is also a best practice for regular programming languages and compilers and should

```
INFO(IIa13): Interval mismatch to <idx> [0,49] := [1,50] (will be limited)
```

Fig. 13 Diagnostic information for limitation

therefore also be applied to modelling languages. For a similar reason, providing diagnostic information can also help to integrate a new feature in a development workflow.

The interesting questions is how these problems could have been avoided. There is certainly some room for improvement in the requirements engineering, but the major decisions were done correctly and the real world is always a little bit more complicated than anticipated. The main point is that the array index protection changes the abstraction of the modelling language: a behaviour that needed to be implemented manually by the users became part of the language. A late change in the semantics of the language is always difficult. Special care therefore needs to be taken to find a suitable abstraction early in the language development. Otherwise users are forced to implement workarounds, and there is a considerable inertia against spending effort to change working solutions.

We will continue to work with both customers to address the issues:

- Improve the analysis to reduce the detrimental impact on the efficiency of the generated code, while avoiding to make the analysis too complex.
- Integrate the external data structure generation with the index analysis.

The topic of safe access to arrays is well understood from a theoretical perspective. The issues encountered when applying the solution in a real-world environment may look trivial in comparison, but can nevertheless be substantial and make an adoption very hard. The mission of a tool vendor must be to overcome these problems as well to deliver an added value to the customers.

References

1. MISRA-C:2012 Guidelines for the Use of the C Language in Critical Systems. MISRA (2013, March)
2. Hocking, A.B., Knight, J., Anthony M.A., Shiraishi, S.: Arguing software compliance with ISO 26262. In: IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 226 – 231 (2014)
3. Blache, G.: Handling index-out-of-bounds in safety-critical embedded C code using model-based development. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, MODELS'16, pp. 143–149. ACM, New York, NY, USA (2016)
4. Boulanger, J.-L., Munier, P.: Industrial Use of Formal Methods: Formal Verification, Chapter 4: Polyspace. Wiley, New York (2012)
5. Esterel Technologies SA. Esterel (2017). http://www.esterel-technologies.com. Accessed 10 Mar 2017
6. Esterel Technologies SA. SCADE (2017). http://www.esterel-technologies.com. Accessed 10 Mar 2017
7. ETAS GmbH, Germany. ASCET (2016). www.etas.com/ascet. Accessed 22 July 2016
8. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and Precise WCET Determination for a Real-Life Processor, pp. 469–485. Springer, Berlin (2001)
9. International Organization for Standardization. ISO/IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. Geneva, Switzerland (2010)
10. International Organization for Standardization. ISO 9899:1999. Geneva, Switzerland (1999)
11. International Organization for Standardization. ISO 26262:2011, road vehicles—functional safety, 2011. Geneva, Switzerland (2011)
12. Kolte, P., Wolfe, M.: Elimination of redundant array subscript range checks. In: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, PLDI'95, pp. 270–278. ACM, New York, NY, USA (1995)
13. Morel, L.: Array iterators in lustre: from a language extension to its exploitation in validation. EURASIP J. Embed. Syst. **2007**(1), 059130 (2007)
14. Object Management Group: Model Driven Architecture Guide rev. 2.0 (2014). http://www.omg.org/cgi-bin/doc?ormsc/14-06-01. Accessed 17 Nov 2017
15. Poledna, S.: Optimizing interprocess communication for embedded real-time systems. In: RTSS'96: Proceedings of the 17th IEEE Real-Time Systems Symposium, p. 311. IEEE Computer Society, Washington, DC, USA (1996)
16. Palin, R., MIRA Ltd., UK, Ward D., Habli, I., Rivett, R.: ISO 26262 safety cases: compliance and assurance. In: Proceedings of the 6th IET International Conference on System Safety, pp. 1–6. IET (2011)
17. Rocheteau, F.: Extension of the lustre language and application to hardware design: the lustre-v4 language and the pollux system. Theses, Institut National Polytechnique de Grenoble - INPG (1992, June)
18. Rugina, R., Rinard, M.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. SIGPLAN Not. **35**(5), 182–195 (2000). https://doi.org/10.1145/358438.349325
19. Sha, L., Abdelzaher, T., Å rzén, K., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A.: Real time scheduling theory: a historical perspective. Real-Time Syst. **28**(2), 101–155 (2004)
20. The Mathworks Inc. Simulink (2017). https://de.mathworks.com/products/simulink.htm. Accessed 14 Mar 2017
21. Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: Network and Distributed System Security Symposium, pp. 3–17 (2000)

Gunter Blache is a product owner (SCRUM) and architect for the ASCET product. He received a German Diplom of computer science (comparable to a M.Sc.) at the university of Bonn (Germany) in 1998 and joined ETAS GmbH (located in Stuttgart, Germany) in 2005. His responsibilities include the requirements engineering of customer requests, planning of product releases and support of customer migrations to new versions of ASCET or introduction of the product in new tool chains. His interest is to improve the state of practice for the development of embedded systems, where engineers in many cases do not have background in computer science.