

Delft University of Technology
Master's Thesis in Embedded Systems

MetaC - Embedded Software specific extensions for the C programming language

A.C. Stolwijk



MetaC - Embedded Software specific extensions for the C programming language

Master's Thesis in Embedded Systems

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

A.C. Stolwijk
a.c.stolwijk@student.tudelft.nl

April 25, 2016

Author

A.C. Stolwijk (a.c.stolwijk@student.tudelft.nl)

Title

MetaC - Embedded Software specific extensions for the C programming language

MSc presentation

April 28, 2016

Graduation Committee

Prof.dr. K.G. Langendoen	Delft University of Technology
--------------------------	--------------------------------

Dr. G.H. Wachsmuth	Delft University of Technology
--------------------	--------------------------------

Prof.dr. E. Visser	Delft University of Technology
--------------------	--------------------------------

Abstract

Embedded Software appears in a variety of systems and products. The software for these systems have special requirements. Firstly, embedded software needs to be very robust as it is usually deeply tucked away and not very visible to the user. Secondly, embedded systems have dedicated hardware the software has to work on, and finally embedded systems can have real-time constraints. The C programming language is the most popular programming language for these kinds of systems and meets the requirements.

Two disadvantages of C are that it is hard to create higher-level abstractions to solve the problem at hand of the programmer, and it is easier to create bugs compared to more modern languages.

To solve these disadvantages we propose the *MetaC* language, which extends the C language with domain specific extensions tailored for embedded systems. The MetaC compiler compiles MetaC code, including the C language and the extensions, to C code.

What extensions can be designed to be helpful for embedded software developers? MetaC implements the C language and adds the following extensions: 1) A bit-fields extension for declaring a bit-fields layout with names and use those names to manipulate the bits instead of using logical operators. 2) State machines to encode the state behavior of a system with new syntax and semantics. That makes it possible to generate state machine diagrams. 3) A concurrency extension for communication between concurrent processes with channels using CSP-style semantics. The extension can generate a model for the PAT model checker. 4) An error handling extension adding error handling constructs that are missing in the C language. Functions can return a new type that indicates the function can return an error. The type system forces programmers to handle the errors of those functions.

The MetaC compiler is implemented using the *Spoofax Language Workbench*, which also provides an Integrated Development Environment (IDE) with common IDE features. The design goals are to implement the extensions in a modular way, to allow separate development of extensions, and to integrate into C as much as possible to give a C feel to the extensions. A *BaseC* module implements the C compiler while separate modules implement the extensions. The modules are composed into the final MetaC compiler.

What are problems implementing MetaC with Spoofax? To reach the design goals several issues had to be solved implementing MetaC with Spoofax. These issues include determining the precedence of new expression operators, composition of scoping rules for new language constructs, and reusing existing name binding rules for new extensions.

Preface

This thesis is the result of the work I did for the past year in order to finish the Embedded Systems master's programme.

I wanted to improve the Embedded Software development process. In Embedded Software development C is the most used programming language. However after taking functional programming courses, using modern high-level programming languages, I wondered if some of the nice development experience could be transferred to the Embedded Software world.

Without a clear idea yet I asked around at the various groups and fortunately the Software Engineering group had a project that could be interesting: Adding extensions (Domain Specific Languages), for Embedded Software, to C using their Spoofax Language Workbench. This sounded like something that would fit my own goal, and with the help of the Embedded Software group for knowledge of embedded systems I could start working. The result is a new programming language based on C with four extensions and an implemented compiler in Spoofax. Hopefully this this work can help developers, in one form or another, improve their development process.

I would like to thank Koen Langendoen and Guido Wachsmuth for their supervision, ideas, and reviewing my work. I would also thank my family for all their support during my studies.

Arian Stolwijk
Delft, The Netherlands
April 25, 2016

Contents

Preface	v
1 Introduction	1
1.1 The MetaC Approach: Extending C	2
1.2 Approach using Spoofox	3
2 Related Work	5
3 Bit Fields	9
3.1 Background	9
3.2 Specification of Bit Fields Declarations	11
3.3 Bit Fields usage	12
3.4 Implementation	13
3.5 Evaluation	15
4 State Machine	17
4.1 Background	17
4.2 Specification	18
4.3 Implementation and generated code	22
4.4 Generating State Chart Graphs	25
5 Concurrency	27
5.1 Background	27
5.2 Code Example	28
5.3 Definition of Processes	32
5.4 Running processes in Parallel: par	33
5.5 Channels for communication between processes	33
5.6 Buffered Channels	34
5.7 Selection of a channel: alts	34
5.8 Verification	36
5.9 Implementation and C Runtime	39

6	Error Handling	43
6.1	Background	43
6.2	Error Function Definition	45
6.3	<code>attempt-fail</code> statement: usage of error functions	46
6.4	Implementation	47
7	Conclusion	51
7.1	Future Work	53

Chapter 1

Introduction

Embedded Software is software that runs on a system that is dedicated for a specific function. This system is embedded in a bigger, typically mechanical, device. Embedded software appears in a wide range of devices, from cars, avionics, factory controllers to smart home appliances. The software controls what a device should do: the software in a thermostat will enable or disable the heating depending on the configured temperature.

The C programming language is very suitable for writing Embedded Software. That makes it the most-used language for Embedded Software. Around 80 percent of the Embedded Software is written in C or C++ [4].

Embedded Software has a couple of requirements that are not common in other fields of software development. Firstly, in contrast with software for Desktops or Servers, embedded software is often deeply tucked away in the device and has a limited user interface. This makes it very important that the software is robust as it will be costly to restart or upgrade the system. For example, in the case of a serious bug in the software for cars, the manufacturer needs to recall many cars to the garage costing millions of euros.

Secondly, since the software is made for specific purposes, and the devices are often mass produced, the hardware is also specific for the device. To reduce costs the hardware will only just meet the minimum necessary requirements, such as CPU abilities, power and available memory. The software is closely tied to the hardware as it usually controls physical parts of the system. The programmer has to keep this under consideration when developing the software.

Finally, a requirement for many Embedded Software is that it should be predictable, meet real-time deadlines and be deterministic. A car airbag should be precisely on time, not too early and not too late, otherwise the execution of the software is at fault causing serious damage.

The C programming language meets the requirements. It provides very good support for writing efficient, low-level code. Because of the popularity

of C many platforms have very good tool support, such as compilers and debuggers.

Unfortunately using C has many disadvantages as well. First of all it is hard to make good abstractions that clearly describe the problem at hand. Examples include data flow blocks, state machines or physical units. High-level graphical modeling tools, such as Simulink, MetaEdit+ and LabView, provide a nice graphical method where different types of components can be connected to each other. However with these tools it is difficult to use lower level hardware features, write C when that would be the best solution, or integrate with legacy code.

Most other higher-level programming languages do not meet the requirements for Embedded Software. A language such as Java has a non-deterministic virtual machine with a Garbage Collector, which is very useful but can make the software unsuitable for certain real-time tasks. Likewise, a popular language as Python uses an interpreter which causes overhead, meaning that more powerful hardware is required to do the same task.

Furthermore, while C is flexible and efficient, the language does not prevent the programmer of making bugs, for example when using wrong integer sizes or void function pointers. Tools such as static analyzers, for example *linters*, can detect various classes of defects while *Valgrind* and debuggers can help programmers find bugs when the software is run, but those tools do not prevent making mistakes in the first place.

1.1 The MetaC Approach: Extending C

Instead of taking a completely different approach by abandoning C altogether, the solution followed in this thesis is to extend the C programming language with domain specific language extensions. These extensions are sometimes referred to as Domain Specific Languages (DSLs). The extensions should provide new programming constructs that help Embedded Software programmers solve their programming problems. If the programmer's problem involves a state machine, the programmer can code the solution with a state machine extension with syntax to create a state machine, the programmer does not need to create the state machine with `switch` statements. The programmer does not have to think anymore how to translate states and transitions of a state machine to C code, but can define the states and transitions directly, which closes the gap between the problem and the implementation code.

An advantage of using language extensions is that analyzing the program becomes easier as the extension serves a specific purpose. A compiler could extract all states and transitions out of a state machine and do specific analysis such as finding terminal states.

Additionally, other artifacts can be generated from extensions. Depending

on the type of extension the compiler could generate diagrams or models that can be analyzed by model checkers.

An already existing solution that uses language extensions is *mbeddr*, an extensible C-based language and IDE for embedded software development [16]. As will be explained later in Chapter 2 it uses projectional editing with the MPS workbench. Völter et al. executed a case study that showed extensions can help the development process, resulting in cleaner code, and reduced testing and integration effort [15].

In order to make it possible to see if extensions can help when implemented in another context, this thesis describes *MetaC* that implements different extensions in a different way. Instead of the MPS language workbench Spoofax will be used. Spoofax is a textual Language Workbench for developing Domain Specific Languages with IDE support [8].

Research Questions: In this thesis the following questions will be answered:

- What could additional language extensions be? *mbeddr* already implements several extensions, so what are other extensions that could provide new abstractions and/or prevent bugs?
- How can C language extensions be implemented using Spoofax? *mbeddr* uses a projectional editing instead of a textual parser, which avoids grammar ambiguities when composing different extension grammars. Also how well does Spoofax provide support for building extensible languages?

1.2 Approach using Spoofax

Spoofax is a language workbench for development of textual Domain Specific Languages with IDE support [8]. Spoofax makes it possible to define a parser for a language using SDF3. The parser generates an Abstract Syntax Tree (AST). This tree can be desugared to get a consistent tree that is easier to work with during the next analysis and generation phases. With the AST and by defining analysis rules Spoofax can resolve references and do type checking using the Spoofax Name Binding and Type system meta languages. Transformations can transform an AST to a modified AST, for example an AST with AST nodes of extensions to an AST containing C AST nodes only. Stratego is the transformation language used in Spoofax. Finally Spoofax can generate code from the resulting AST.

A design goal of *MetaC* is that extensions are built in different modules that are composed to the final *MetaC* language. That would make it possible to add, modify, or remove extensions later, without interfering with the other extensions.

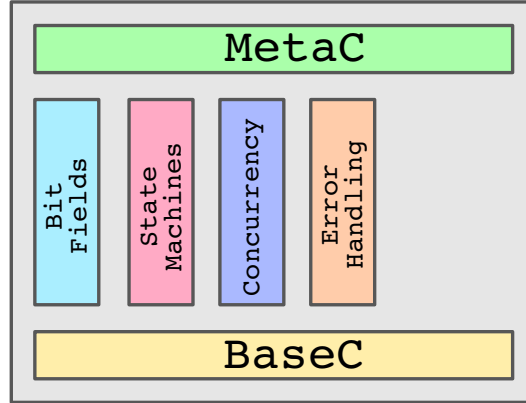


Figure 1.1: BaseC is the basis of MetaC, extensions are implemented separately on top of BaseC. MetaC composes everything to the final system.

Before it was possible to build extensions, the most essential module for the basis C language was implemented as the *BaseC* module. It defines the standard C syntax and analysis, such as resolving variable references. Each language extension is a separate module. The MetaC module imports BaseC and all modules of extensions and composes them into the final language (see Figure 1.1). Separate modules for extensions create a clear separation so they can be developed independently according to the open-closed principle: open for extension, closed for modification [10].

Abstractions in the MetaC extensions should *feel* familiar for C programmers and integrate nicely into the C language. The rationale is that existing C programmers should be able to quickly pick up the new syntax. This means reusing existing C syntax, and designing new syntax that will be familiar to C programmers.

The MetaC compiler will translate code using extensions to regular C code. This output can then be used by a C compiler to compile for the target platform. This makes building the MetaC compiler a lot simpler and makes extensions easier to implement, while specialized C compilers are very good at compiling to machine code.

Outline The structure of this thesis proceeds as follows. First related work is discussed in Chapter 2. Then chapters 3, 4, 5, and 6 describe implemented extensions. Chapter 3 is a small extension for working with bit manipulations. Chapter 4 describes an extension for defining and using State Machines, a similar extension mbeddr has. In Chapter 5 a non-trivial concurrency extension is described with a C API as target. An error handling extension is described in Chapter 6 that should make error code returning functions safer. Finally Chapter 7 concludes this thesis.

Chapter 2

Related Work

MetaC is inspired by several other projects. In the following sections are structured accordingly, one for each project.

mbeddr A big inspiration of this work is *mbeddr*. Mbeddr is an extensible C-based language and IDE for embedded software development [16]. Embedded software typically has reliability, safety, efficiency and real-time constraints as requirements. The *programming* approach using C or the *modeling* approach using tools that provide higher-level abstractions both have disadvantages. *Domain specific* languages (DSLs) are increasingly used in embedded software, but most real-world systems cannot be described completely and adequately with a single modeling tool or DSL. The mbeddr approach is a much tighter integration between low-level C code and higher level abstractions specific to embedded software.

The authors define five challenges: 1) making abstractions without runtime costs, 2) make C safer, 3) add program annotations, 4) static checks and verification, and 5) process support. Mbeddr addresses these challenges with an extensible version of the C language. Implemented extensions are state-machines, components, modules, physical units, unit tests, requirement traceability, and product line variability. The extensions are not limited to these extensions, but the mbeddr system is designed to be extensible for other extensions, which include project-specific extensions.

To provide an extensible language system as well as an IDE that is aware of the extensions mbeddr uses the JetBrains MPS technology. MPS is a language workbench, similar to Spoofax, for defining languages. Its most important characteristic is that it is a projectional editor. Instead of a parser parsing text into an AST a projectional editor edits the AST directly. The projection engine then generates a representation that is shown to the user. This representation can be text, in case of editing C code in mbeddr, decision tables, mathematical formulas, graphics, or something else the AST could represent. Since no grammars are used, no syntactic ambiguities can result

from two independently developed languages. The challenge for projectional editing is to make the editing experience convenient and productive. When the AST is saved as a file the AST is stored as an XML file, that makes it harder to work with tools such as version control.

In [15] Voelter et al. report on an industrial case study using domain specific extensions for C with mbeddr. The goal is to provide evidence to what extent language extensions supported by mbeddr are useful for the development of embedded software. The authors test the following aspects:

Complexity The developers of the case study project thought in terms of extensions and suggested additional extensions. The mbeddr component extension helps to structure the overall architecture. The used extensions facilitate in strong static checking, improve readability and help to avoid low-level mistakes.

Testing The components extension of mbeddr used in the case study improved testability, leading to 80% test coverage for core components, much higher than state-of-the-practice in industry. Custom extensions made hardware independent testing possible, so the system could be tested in a continuous integration process.

Overhead The overhead of extensions can be categorized into three categories: 1) without runtime footprint, 2) similar footprint as manually written C code, and 3) requires more sophisticated code structures. For the use case the performance overhead was low enough to meet the requirements. The components extension enables deployment of only the required functionality, minimizing the binary size.

Effort The effort for integration and testing is lower than usual in embedded software. In total, including implementation, the effort is similar to what could have been achieved with C.

All these tested aspects seem to be positive. This can be explained because mbeddr is specifically designed to achieve these benefits. The extensions improve C incrementally, so the developer can use the appropriate language features.

The developers of mbeddr have chosen MPS because of its support for modular language extensions and flexible notation. To a degree those features are also available in other language workbenches and they expect similar results when building something like mbeddr in those workbenches.

To support the findings of this case study, the authors think additional studies are necessary using mbeddr-based systems but also other extension-based approaches in embedded software are necessary. The work on MetaC aims to provide a starting point with an alternative implementation as a C-based language with extensions for embedded software.

nesC is a language based on C with a specific domain in mind: embedded networked systems. The nesC language uses an event-driven programming model for flexible concurrency and a component oriented-application design. Using this model the compiler can perform static analysis to detect race-conditions or perform aggressive function inlining. Components are statically wired together and post tasks for the scheduler to execute. Tasks are atomic, non-preemptable, for other tasks, which allows nesC to avoid data-races. However, tasks can be preempted by *events* (usually interrupts). Atomic code sections are a code blocks where interrupts are disabled so further data-races can be avoided.

Contrary to other languages with a similar execution model, nesC is an extension of C. This makes it possible to use the well known low-level features necessary to access hardware and to interact with existing C code. NesC provides help writing safer and more structured code.

The nesC extension to C is very specific. A similar extension could be built using the mbeddr or MetaC approach. Then the mbeddr or MetaC toolset can be used, and the extension could be used alongside other extensions.

Hume is another Domain Specific Language approach for Real-Time embedded systems that is not based on C but is an external DSL [6]. The language is based on a combination of λ -calculus and finite state machine notations with a syntax similar to Haskell. Its goal is to provide high level abstractions while maintaining properties necessary for real-time systems, such as determinancy and bounded time constraints. The main abstraction is boxes, that can be wired together, like nesC components. The language is layered. A meta programming layer simplifies the creation of boxes and wires, for example for different types or repetition of boxes. Exceptions that are raised must be handled by the surrounding box which is checked by static analysis.

Both nesC and Hume are domain specific languages for embedded software. Like mbeddr and MetaC, nesC extends C, but only with one *specific* extension, while mbeddr and MetaC could potentially add the nesC extension as well.

Hume, and other DSLs such as Feldspar [1], have a completely different syntax than C. This allows compilers to do more static analysis to guarantee safety, or to do more optimizations. A drawback is the interaction with legacy code that becomes harder to integrate or to migrate from.

Mbeddr tries to be a generic solution for specific extensions to C. Its authors have chosen the MPS language workbench to implement it, but that forces developers to use the MPS IDE as well. Though mbeddr has shown that extensions are a viable solution for embedded software development, with increased abstractions in the code and reduced integration and testing effort. MetaC takes these ideas but has an alternative, textual implementation with some different extensions.

Chapter 3

Bit Fields

The first MetaC extension is for bit manipulations. Bit manipulation is a technique often used when writing C programs, especially when working closely on the hardware, as in Embedded Software. Getting the bit manipulations correct the first time can be tricky to program. This extension should make it easier for the programmer to work with bit fields.

A similar extension is not available in mbeddr. This extension is relatively straightforward to implement, as reads and writes of Bit Fields can easily be translated to the related bit manipulations.

3.1 Background

Bit manipulation is a way to read and write specific bits at a given memory location. A memory location can be connected to peripherals, like sensors or actuators. For example reading from a certain memory location can give the current value of a sensor. Alternatively, writing values to a certain location can configure peripherals. Bit manipulation can also be useful dealing with boolean data in a memory efficient way as only one bit is needed for each value.

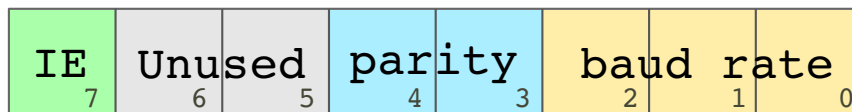


Figure 3.1: Layout of register bits of an serial interface.

The principle of working with bits is that bits are ordered accordingly to a given layout. Manipulating bits in C requires logical operators such as the *and*, *or*, *complement*, *exclusive or* and *left* and *right shift*. Figure 3.1 shows an example of the layout of some serial interface (UART) registers. Figure 3.2 shows the traditional method how bits can be manipulated using the existing operators in C.

```

#define INTERRUPT_ENABLE (1 << 7)
#define PARITY_SHIFT 3
#define PARITY (3 << PARITY_SHIFT)
#define BAUD_RATE (7 << 0)
int main() {
    /* the memory to modify */
    unsigned int flags = 0;
    /* enable bit 7 */
    flags |= INTERRUPT_ENABLE;
    /* disable bit 7 */
    flags &= ~INTERRUPT_ENABLE;
    /* toggle bit 7 */
    flags ^= INTERRUPT_ENABLE;
    /* test the bit */
    if (flags & INTERRUPT_ENABLE) {
    }
    /* set multiple bits to a new value: 2 */
    flags &= ~PARITY;
    flags |= 2 << PARITY_SHIFT;
    return 0;
}

```

Figure 3.2: Traditional method of manipulating bits in C.

C programmers will understand the code in Figure 3.2 quite easily, but unfortunately it is easy to make small mistakes that the programmer will not notice directly. Muscle memory could cause writing the *logical and* (`&&`) operator instead of the *bitwise and* (`&`) operator. The compiler will happily compile the code because it does not understand the intent of the programmer, for example when he or she wants to test for a bit using the expression `flags & INTERRUPT_ENABLE`.

When working with groups of bits the operations become more complex since multiple bits have to be modified. An operation such as “set the middle two bits to a value of 2” can require some puzzling by the programmer to get right.

Standard C structures can be used as a solution. When using structures, the number of bits that need to be used can be specified. However the C programming language specification only guarantees that the compiler must reserve this many bits for the field. It does not guarantee that the bits are ordered accordingly to a certain layout. If the compiler wants, it can put padding bits between fields to make other operations faster. It could place a field at the 8th bit instead of the 6th. Given this freedom for the compiler, this method cannot be used when the layout of the bits is significant.

```

#include <assert.h>

// define the layout of the fields
bitfields UART {
    // 'interrupt_enabled' is the name of the field
    interrupt_enabled: 1;
    unused: 2;
    parity: 2; // 2 is the number of bits
    baud_rate: 3;
};

int main() {
    // declare a variable of bitfields layout UART
    UART a;
    // set field 'interrupt_enabled' to 1
    a.interrupt_enabled = 1;
    // assert that the bit has changed
    assert((a & (1 << 7)) != 0);
    // field 'parity' with 2 bits becomes 2
    a.parity = 2;
    // reading a bit field
    printf("%d\n", a.parity);
    return 0;
}

```

Figure 3.3: a bit fields extension example.

Figure 3.3 shows an example of the usage of the proposed Bit Fields extension. First the layout of the Bit Fields are declared using the `bitfields` keyword. This declaration contains the fields with their names and the number of bits each field has. At another location the Bit Fields declaration can be used by declaring a variable with the type as the name of the Bit Fields declaration. In this example this is `UART`. Finally by using the *member operator*, `a.interrupt_enabled`, the bits can be assigned or read.

3.2 Specification of Bit Fields Declarations

The Bit Fields declaration is designed to look like C `struct` or `union` declarations. A new Bit Fields declaration is defined by the `bitfields` keyword. A `bitfields` declaration must have a valid identifier as name. The declaration declares a new type of this name and is used to refer to this Bit Fields declaration elsewhere in the program. After the name a list of fields,

the layout, follows, enclosed by braces. Each field has a name as identifier, followed by a colon “:” and an integer to indicate the number of bits the field can hold. Each field is ended with a semicolon “;”.

Field names have to be unique for each Bit Fields declaration. However two different Bit Fields declarations can both have a field with the same name.

The positioning of the bits is laid out as shown in Figure 3.1. The bits are aligned to the right. For `bitfields UART` from Figure 3.3 the last field, `baud_rate`, will be placed at the three right-most positions, followed by the `parity` field, that will be placed at locations 3 and 4, the `unused` field at 5 and 6, and finally `interrupt_enabled` at position 7.

3.3 Bit Fields usage

A variable can be declared with the Bit Fields name as variable type: `bitfield_type variable_name;`. Optionally an initial value can be given to the declaration. This value should be an integer literal. The variable can also be declared as pointer, so a pointer to a number or even another Bit Fields variable can be assigned to the variable. Figure 3.4 shows a few different possible variations.

```
bitfields A { a: 2; b: 4; c: 2; };
void f() {
    A a; // without initializer
    A b = 0x1F; // arbitrary number
    A *c = &a; // pointer to another number or bitfield
}
```

Figure 3.4: Examples of declaring Bit Fields variables.

Setting and getting values to and from Bit Fields is done using the same operator as structures or unions already known in C with the member operator “.”. The left hand side of the operator needs to refer to a Bit Fields variable while the right hand side of the operator should match one of the fields listed in the Bit Fields declaration of that variable.

If the member expression happens on the left side of an assignment expression, it will be used as an assignment. The right hand side of the expression will be written to the relevant bit positions the field refers to.

If the member expression happens elsewhere, it is used to retrieve the data. The expression will evaluate to the value in the memory the bits of the field of this expression refers to.

Bit Fields variables declared as pointer, `A *b;`, would need to dereference the variable first, so `*b` refers to the Bit Fields, and `(*b).field` would

refer to some bits. Instead of writing `(*b).field`, it is possible to use the `b->field` operator, just like structures in C. Figure 3.5 gives a complete example of the usage of Bit Fields pointers.

```
#include <assert.h>

bitfields A { x: 4; y: 4; };

int main() {
    int number = 0;
    A *b = &number;

    b->x = 1;
    assert(number == 0x10);

    b->y = 4;
    assert(number == 0x14);

    return 0;
}
```

Figure 3.5: Using the `->` operator.

3.4 Implementation

The only new syntax for Bit Fields is the Bit Fields declaration with the names of the fields and the corresponding sizes.

During the desugaring phase, the start and end positions of each field are determined. In the case of Figure 3.1 field `baud_rate` would have range `0...2`, field `parity` range `3...4` and so on. The desugared Bit Fields declaration will also contain the total number of bits.

In the analysis phase the Bit Fields declaration defines itself as a new `typedef` type, so type names will refer to the Bit Fields declarations. Each field defines itself and is scoped by the Bit Fields declaration. The range and size of the field are stored as properties so they can be used when the field is referenced.

The fields are referenced by the default C member operators `a.b` and `a->b`. Resolving fields depends on the type of the left side of the expression. In BaseC it first looks if it is a structure, otherwise it will try to use it as a union, see Figure 3.6. This should be altered to include Bit Fields.

To keep the implementation modular, BaseC should not include Bit Fields code, thus the default Spoofox Name Binding language solution will not be

```

Field(e, Identifier(field)):
  refers to Field field in Struct s
    where e has type Struct(Identifier(s))
  otherwise refers to Field field in Union s
    where e has type Union(Identifier(s))

```

Figure 3.6: Spoofox Name Binding code for for resolving member expressions.

```

field-lookup = register-field-lookup(|
  "rewrite-struct-name",
  NablNsStruct(),
  NablNsField())
task-rewrite: ("rewrite-struct-name", Struct(Identifier(s))) -> s

```

Figure 3.7: Stratego code to register a new candidate.

good enough. The BaseC name binding can use Stratego with the underlying name binding API directly. At the `nabl-use-site` the adjusted BaseC name binding code tries to collect all registered candidates in a modular fashion. The essential differences between each candidate are the namespaces of the enclosing construct, the field namespace, and a mapping from the type to the type name, for example `Struct(X) -> X`.

Registering candidates happens by creating a `field-lookup` strategy that is called from the BaseC code with a list of already registered candidates. This strategy uses `register-field-lookup` that was added to the BaseC Stratego API. This strategy will fail if the candidate is already in the list, otherwise it will succeed and return a new list with the extra candidate. The result is that the BaseC strategy collects a list of methods defined at different places how the field expression can be used.

3.4.1 C Code Generation

```

bitfields UART { /* ... */ };  typedef uint8_t UART;

```

Figure 3.8: Compilation scheme of a `bitfields` declaration.

When the code analysis analysis is complete the C code can be generated. Bit Fields declarations are transformed to `typedef` declarations, as shown in Figure 3.8. The name of the declaration is the same as the Bit Fields declaration name. This way the type can still be used in other places like function parameters or cast expressions. The integer type used for this `typedef` depends on the total number of bits in the Bit Fields declaration. If

the number of bits has a maximum of 8, the type is `uint8_t`, for 16 `uint16_t`, for 32 `uint32_t` and for 64 `uint64_t`.

```
UART a = 31;                                UART a = (UART) 31;
```

Figure 3.9: Compilation scheme of a Bit Field variable declaration.

Figure 3.9 shows how a variable declarations is transformed. If it has an initial value, that value is casted to the correct type to prevent C compiler warnings. For example this allows overlaying 32 bit integers with a Bit Fields declaration with only 8 bits. If the variable declaration does not have an initial value, the declaration is changed so the initial value of 0 is assigned. This should prevent confusions when the C compiler does not do this.

To retrieve bits the number is shifted to the right first, and then masked to filter only the relevant bits. For a field with the third and fourth bits, that translates to `(a >> 2 & 0x3)`, where `a` is the variable.

For setting the third and fourth bits `a = (a & ~(0x3 << 2) & 0xFF) | ((1 & 0x3) << 2)` is used to set them to value 1. At the left-hand side of the logical-or expression the bits are unset. The right-hand side makes sure the value will fit in the two bits and shifts the value to the correct location. The result is ORed and written to the variable `a`.

3.5 Evaluation

The Bit Fields extension can be a useful extension for programmers when working with bits. The generated code is quite small, but to get the analysis and code generation working some workarounds were necessary with the Spoofox implementation. The `register-field-lookup` works so far because all types are different (structures, unions, or bit fields) the order they are registered is not relevant. If it would be relevant composing the field lookup analysis would be harder because different extensions should not know about each other, so MetaC would need to resolve the precedence.

Chapter 4

State Machine

Almost all embedded software contains some state-based behavior [9]. This chapter discusses a MetaC State Machine extension that provides special syntax for expressing state machines in code. Mbeddr provides a similar extension. The syntax is very similar, both can generate diagrams of a state machine declaration.

4.1 Background

State Machines model the behavior of a system with a finite number of *states*. A State Machine can only be in one state at the time. An *event* can trigger a transition to move to another state.

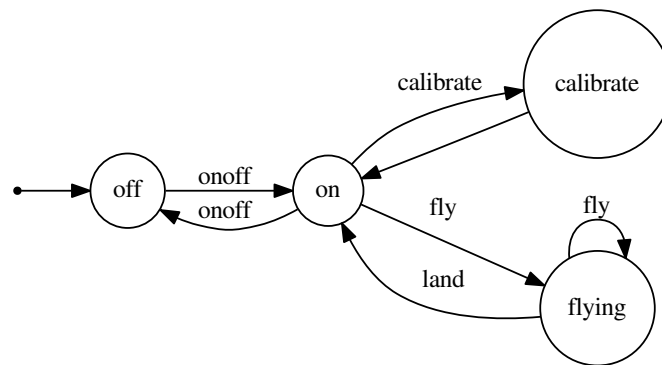


Figure 4.1: A State Machine graph of a simple Drone State Machine.

An example of a system where a state machine can be used is a drone. A state machine diagram is shown in Figure 4.1 The drone has multiple operational *states*: off, on, calibrating and flying. *Events* are the inputs of the system from the outside, for example from the person controlling the drone. Some events for this system are pressing the on-off or calibrate buttons and

new flight info (speed, rotations). An event can trigger a *transition*, the calibrate event will trigger a transition from the on state to the calibrate state.

Traditionally a state machine is implemented in C with `switch` statements. With growing complexity this might lead to deeply nested conditional statements that are difficult to understand or test.

The state machine extension provides new syntax to define a state machine with states and transitions between states. The extension provides a clear framework to structure the state machine code. The advantage compared to runtime solutions, such as libraries, is that it enables more possibilities for static analysis. Additionally there is support for generating state chart diagrams.

4.2 Specification

Figure 4.2 shows an example of the syntax of the state machine extension. It shows a state machine describing a drone. The idea is to have a similar construct as structures or functions, but for state machines. Inside the construct the states and events are listed and in each state the transitions are listed.

```

statemachine Drone {
    init off;

    uint8 calibrated = 0;
    uint32 speed = 0;

    event onoff();
    event calibrate();
    event fly(uint32 input_speed);
    event land();

    state off {
        entry { calibrated = 0; }
        on onoff -> on;
    }
    state on {
        on onoff -> off;
        on calibrate -> calibrate;
        on fly [calibrated == 1] / speed = input_speed; -> flying;
    }
    state calibrate {
        entry { calibrated = 1; }
        on -> on;
    }
    state flying {
        on fly [calibrated == 1] / speed = input_speed; -> flying;
        on land -> on;
    }
}

```

Figure 4.2: A simplistic drone with four states and for events.

The initial state is the **off** state. The **onoff** event will trigger the transition to the **on** state. The **on** state has three transitions, back to **off**, to the **calibrate** state or to the **fly** state, which is only possible when the local variable **calibrated** equals 1. If the **calibrate** event triggers the transition to **calibrate** the new state of the state machine will be the **calibrate** state. The **entry** block sets the local variable **calibrated** to 1 to indicate it has calibrated the drone. This state has only one transition, without an event, which will be executed immediately after the entry block is done, and will transition back to the **on** state. Now the **calibrated** variable equals 1, the transition to **flying** can be taken, but while taking the transition, the transition action is executed, which will assign the **input_speed** parameter to the **speed** variable. Finally the **flying** state will be exited when the **land** event is triggered.

The following paragraphs will explain the different elements in more detail.

statemachine declaration

The **statemachine** keyword declares a new state machine type. After the

keyword follows a valid identifier indicating the name of the state machine. The initial state, events, variables and states are declared inside the following accolades. The programmer can declare the initial state with the **init** keyword, name of the initial state and a semicolon. The initial state declaration must come first, then events and variables can be defined without a particular order and finally the states can be defined.

event declaration

An event is the input of the state machine by the user of the state machine. Events can be declared inside the **statemachine** block by the **event** keyword followed by a identifier which is the event name. The **event** keyword is necessary to distinguish from function declarations as **int foo()**;. The following parenthesis indicates the event parameters in a similar way as function parameters: a parameter type with a parameter name, and multiple parameters separated by commas. The user of the state machine can pass values into the state machine, for example the new flying speed of the drone.

state declaration

Each state the state machine can have is declared with the **state** keyword. The keyword is followed by the name of the state, which should be a valid identifier. The content of the state is surrounded by accolades after the name of the state. Transitions to other states are declared inside the accolades, as well as an *entry action*. An entry action is a code block (compound statement) that is executed when the state machine enters this state.

on declaration for transitions

Inside the accolades of the state a programmer can define the transitions from this state to another state. After the **on** keyword there are three optional items: an event name, a *guard expression* and a **transition action**. The transition is ended with an arrow, **->**, the name of the new state and a semicolon.

The event name is optional. If an event name is given, the event parameters do not need to be repeated, as they are defined at the top of the state machine already. Multiple transitions from different states can be triggered by the same event, which would mean duplication of the parameter list or conflicting parameter types at the different transitions. Therefore the event parameters are defined once and are referred to by only the event name.

A special type of transition is when the transition has no event name. This is called an epsilon transition. This transition is taken directly after executing the **entry** block, if the guard expression is **true**.

A guard expression is a boolean expression between square brackets. Only if this expression evaluates to **true** the transition is taken. The event parameters can be used inside this expression. Using guard expressions it is

possible that the same event triggers another transition. The transitions are executed sequentially, so if multiple expressions are true for the same event, the first transition is taken.

Transition actions execute some code statement when the state machine executes a transition to a new state. This can be useful to set some variable or to show something on a display. The transition action is denoted by a forward slash and a C statement. This denotation is taken from UML state machines. The parameters of the event can be used inside this statement.

Figure 4.3 shows how the state machine of Figure 4.2 can be used.

```
#include <assert.h>
void f() {
    statemachine Drone drone;

    drone<|onoff();

    // can't go to flying if it's not calibrated
    drone<|fly(10);
    assert(drone|>speed == 0);

    // calibrate, directly go to the on state again
    drone<|calibrate();
    assert(drone|>calibrated == 1);

    // now we can go to flying
    drone<|fly(20);
    assert(drone|>speed == 20);

    drone<|land();

    drone<|onoff();
}
```

Figure 4.3: Syntax for using the Drone State Machine.

In Figure 4.3 an instance of the state machine is created by declaring a new variable of the `statemachine Drone` type. Events can be triggered by the `<|` operator, while the state machine variables can be read from the state machine by using the `|>` operator.

State machine variable declaration

The programmer can use the state machine type in a variable declaration

using `statemachine` with the name of the state machine as variable type. This initializes the state machine as well.

Trigger state machine events

The state machine does not change without events from outside. The programmer can trigger events with the `state_machine<|event(parameters)` expression. The state machine variable is on the left side of the expression and the event name is on the right side, followed by the event arguments. The expression can be read as: input this event with parameters into the state machine.

Accessing state machine variables

The variables that are declared inside the state machine can be accessed outside the state machine as well using the `state_machine|>variable`. All variables of the state machine are visible, though accessing variables should be done responsibly by the programmer, and a future `private` or `static` keyword could enforce that.

4.3 Implementation and generated code

Syntax and Analysis

The syntax of state machines is defined using SFD3 as a new declaration, meaning that everywhere where declarations can be declared in a C program a state machine type can be declared as well. Declaring a state machine variable is defined as a declaration too.

Triggering state machine events and accessing state machine variables are defined as post-fix expressions, like the `a++` expression. Expressions have a certain precedence or priority; the `*` operator binds stronger than the `+` operator. In SDF3 the context-free priorities defines the relative priorities between productions to avoid grammar ambiguities. With the priorities it is possible to use the same `Expr` sort declarations for all types of expressions. However, to follow the C grammar specification BaseC uses different sorts for the different types of expressions: binary, unary, post-fix, etc. The state machine expressions are added as post-fix expressions where the left-hand side can only contain other post-fix expressions. This avoids ambiguities. However extensions could add other new expressions, for example new binary operators that result in grammar ambiguities, as is done in the fictive example in Figure 4.4. The ambiguity is solved by the context-free priorities, but if another extension adds another operator the context-free priorities need to define the relative priorities between the two operators. Other extensions do not know about each other, so defining the context-free priorities would need to be done in the main MetaC module that includes all extensions.

```

context-free syntax
  AExpr.At = [[AExpr] @ [AExpr]] {left}
context-free priorities
  { UnaryExpr.Negate }
  > { left: AExpr.At }
  > { AExpr.Add }

```

Figure 4.4: SDF3 Grammar with fictive @ operator results in an ambiguous grammar.

The state machine expressions also have a different syntax than one might expect. The member expression, `a.b`, using the dot, might be more logical for new users of MetaC. This would require analysis that is currently not supported by the Spoofox name binding language. Files from BaseC and files from extensions should be able to contribute name bindings for the field expression depending on the type of the left-hand side. Separate syntax will parse to another AST node so a separate name binding rule can handle the name binding of this node.

In BaseC the top AST node, `Program()`, scopes the variable and function namespaces. Variables and functions are known in BaseC. The state machine name binding definitions define the namespace for state machine, which BaseC has no knowledge of, so BaseC cannot scope the state machine namespace. This would result in an unwanted *duplicate declaration* error when defining a state machine of name *x* in two separate files.

The solution for the scoping problem is to define the `nabl-get-scope` strategy in Stratego in BaseC using the Spoofox Stratego API directly rather than in the name binding language. Instead of returning a list of name binding namespaces directly, it collects the results of a custom `get-program-scope-nabl-namespace` on the `Program()` AST node. This `get-program-scope-nabl-namespace` can be declared in the modules of any extension as Stratego composes multiple definitions of this strategy with the choice, `<+`, operator, where the result of the first succeeding strategy is returned. Now the state machine module can define the Stratego rule using the stratego code shown in Figure 4.5. The `nabl-get-scope` executes a search, and because of the `get-program-scope-nabl-namespace` definition in the state machine module it succeeds and returns the name binding state machine namespace, after which it is concatenated with the list of namespaces of variables and functions which is the eventual result of `nabl-get-scope`.

```

get-program-scope-nabl-namespace:
  StateMachine(_, _, _, _, _) -> NablNsStateMachine()

```

Figure 4.5: Stratego rule to define the scoping rule of the state machines.

C Code Generation

The generated C code consists of four elements: a structure with data containing the current state and state machine variables, a transition function that updates the state machine state when taking a transition and executes the *entry* code block, an *exec* function that is called when an event is executed, which may trigger transitions and execute transition actions. Finally an *init* function that initializes the state machine, by setting the initial values of the state machine variables and executing the entry code of the initial state.

```
state on {
  on onoff -> off;
  on fly
    [calibrated == 1]
    / speed = input_speed;
    -> flying;
}

case 1: {
  if (__sm_event__ == 0 && 1) {
    __sm_cont__ = 1;
    __sm_event__ = -1;
    Drone_transition_to_state(sm, 0);
    break; }
  if (__sm_event__ == 2
      && sm->calibrated == 1) {
    sm->speed = input_speed;
    __sm_cont__ = 1;
    __sm_event__ = -1;
    Drone_transition_to_state(sm, 3);
    break; }
  break; }
```

Figure 4.6: Compilation scheme of state in the `switch` of the `exec` function.

The most important function is the *exec* function. This function takes the state machine object and an event number as arguments. A while loop makes sure that epsilon transitions, transitions without an event, are automatically taken: when this function executes and a transition is taken, it will reset the event number and execute the entire function body again so possible epsilon transitions are executed too. A `switch` statement is inside the while loop. Figure 4.6 shows how a state is translated into a case statement in the `switch` statement. Depending on the current state the transitions from that state are checked with `if` statements. If the event number argument matches the transition, and the guard expression is true, the transition is taken by executing the `transition_to_state` function which modifies the state machine object with the new state. Epsilon transitions do not have an associated event, so only the guard expression, if available, has to be evaluated to true.

The local variables inside the state machine can be referred to as normal variables inside entry statements, transition actions and transition guards. In the C code they are stored in the state machine structure object. Therefore the variable expression should be changed from `local_variable` to a pointer member expression, `sm->local_variable`. However not all variables should be transformed, for example variables from a higher scope, like global variables. A variable expression is transformed if the *nabl uri*, the location

of the AST node, of the state machine is equal to the parent scope of the variable declaration an identifier refers to.

When a state machine is initialized by declaring a state machine variable, a variable for the state machine structure is created. This structure is passed into the *init* function, which initializes the local state machine variables and executes the initial transition to the initial state.

<pre> statemachine Drone drone drone< fly(20); </pre>	<pre> Drone drone; Drone_EventParams drone_event_args; Drone_init(&drone); (drone_event_args.input_speed = 20, Drone_exec(&drone, 2, &drone_event_args)); </pre>
--	---

Figure 4.7: Compilation scheme of initializing a state machine and triggering an event.

Figure 4.7 shows how a trigger expression is translated into the *exec* function. Each event is numbered and will be passed to the *exec* function call together with a pointer to the state machine structure and the event parameters. The parameters are stored in a separate structure and a pointer to this structure is passed into the *exec* function so the transition guards and actions can use these values. Inside the *exec* function the values are assigned to variables from the arguments structure so the variable references do not have to be transformed when the event parameters are used.

4.4 Generating State Chart Graphs

One of the advantages of creating an abstraction for state machines with new syntax is that analyzing the code becomes easier. By explicitly declaring the states and transitions it becomes not only very clear for the programmer, but the compiler can generate other artifacts than just code, such as diagrams. The programmer can press the *generate diagram* button in the MetaC IDE to automatically generate a diagram. This diagram can help programmers to analyze and better understand the system to make sure everything is implemented as intended.

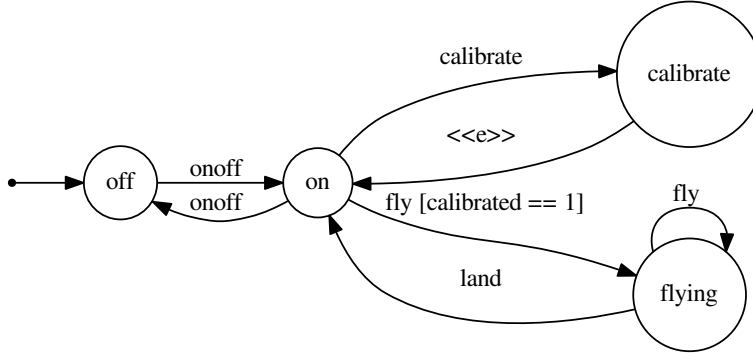


Figure 4.8: A generated graph from the Drone State Machine code.

The Figure 4.8 shows the graph generated from the code of Figure 4.2. Each circle depicts a state and the arrows between the circles are the transitions. If a transition is triggered by an event, the event name is shown as the arrow label. For epsilon transitions the label is `<<e>>`. If the transition has a guard, the expression is added to the label between square brackets. The initial state is represented with a transition that is only connected with one side to this state. Terminating states, without outgoing transitions, have a double circle in the diagram.

To generate the diagram the AST of the state machine is transformed to the DOT language [5]. In the DOT language the states are listed with the correct shape: a circle for a non-terminating state and a double circle for a terminating state. Then all transitions between the states are listed using the event name as label. Guard expressions use the pretty-print Stratego strategy generated by Spoofax and are added to the transition label.

Finally the diagram is generated using the default DOT tools.

Chapter 5

Concurrency

It is common that embedded systems have multiple components that run continuously alongside each other and need to communicate to work together. The concurrency extension adds language features to create parallel processes and synchronous channels for communication.

5.1 Background

Embedded systems often consist of multiple concurrent components, Real Time Operating System (RTOS) tasks or interrupt service routines (ISR), that have to communicate with each other. A component has to let other components know something happened or share some data. Interrupts are usually the initial source of data that needs to be processed by the system, flowing into the various components.

One of the easiest methods to share data between components is using global variables. However this introduces the risk of concurrency bugs. Maybe the most common type of bugs occur when a data value that is multiple machine words long is read by a task, and is changed by another task at the same time. As the read is not atomic, the final result may contain some data from before the write by the other task, and some new data from after the write that makes the result invalid. When global variables are used it is important to protect the variable with a mutex or disable interrupts temporarily to avoid concurrency problems [9]. Using mutexes manually can sometimes be hard to use and is prone to bugs because there is no control of their proper usage. Also there is no connection between the data controlled by the mutex and the mutex.

The concurrency extension adds features for defining concurrent processes and channels to make communication between concurrent processes easier to program. Processes can read from and write to channels to communicate with each other. When a process is ready to write to a channel and another concurrent process is ready to read from this channel the communication is

performed and both processes continue with further execution.

The extension design is influenced by Communicating Sequential Processes (CSP) [7]. CSP is a language for defining the interaction of concurrent systems. It has influenced many languages since, including Occam [11] and Go [12].

As CSP is also used as a process algebra, it is applied as a method for specifying and verifying concurrent systems. As concurrency bugs do only appear so often, tests might not catch the bug, but formal verification can find those issues. Using new syntax features to define the processes and the communication between processes, verification model specifications can be generated directly from the MetaC code.

The extension introduces new syntax for processes and channels that do not have a direct equivalent syntax in plain C or mbeddr. Instead the extension uses a small C runtime library.

5.2 Code Example

The following example shows a simple controller that could be found in a drone. During development the drone receives data through a *RS232* connection. When data is received the RS232 ISR is triggered. Depending on this data the drone should change the settings of the four rotors.

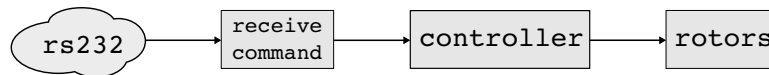


Figure 5.1: A diagram of the example system.

Figure 5.1 shows a diagram of the example system. When the RS232 ISR is serviced it passes a value to the `receive_command` processes that decodes commands, the decoded command is then passed to the `controller` processes that translates the command into rotor settings which are sent to the `rotors` process that controls the rotors.

The following paragraphs, will briefly explain the code of the example while the following sections will explain the features of the concurrency extension in detail.

```

int main() {
    chan<SetPoints> setpoint;
    chan<RotorSettings> rotors;
    set_interrupt(RS232_ISR, &rs232_isr);
    par {
        receive_command(rs232_channel, setpoint @ STANDBY_ENABLED);
        controller(setpoint, rotors);
        rotors(rotors);
    }
    return 0;
}

```

Figure 5.2: Running processes concurrently.

In Figure 5.2 the three processes are started concurrently with a **par** statement. The statement is used in an ordinary C function that happens to be the **main** function of the program in this case. Besides starting the processes, the channels that provide the communication between the processes are declared as well. These channels are then passed into the processes as the process arguments. The @ sign in the arguments list of **receive_command** call separates process arguments and the initial values of the *state variables* for the process, as will be explained shortly. Furthermore, the function sets an interrupt service routine to receive the data from the RS232 connection.

```

typedef struct {
    uint8 rotor1, rotor2, rotor3, rotor4;
} RotorSettings;

typedef struct {
    uint8 yaw, roll, pitch, speed;
} SetPoints;

process controller(
    chan<SetPoints> setpoints_channel,
    chan<RotorSettings> settings_channel) {
    RotorSettings settings;
    SetPoints setpoints;
    setpoints_channel ? setpoints;
    calculate_rotor_speeds(&settings, &setpoints);
    settings_channel ! settings;
    controller();
}

process rotors(chan<RotorSettings> settings_channel) {
    RotorSettings settings;
    settings_channel ? settings;
    set_rotor_speeds(&settings);
    rotors();
}

```

Figure 5.3: A drone rotors controller. Two processes communicating over a channel.

Figure 5.3 shows the code of the `controller` and `rotors` processes. The `controller` process has two parameters: both are channels, one with `SetPoints` values, specifying what the yaw, roll, pitch and speed of the drone should be. The other channel is for `RotorSettings` values that are the rotation speeds for each of the four rotors. Inside the body of the process `SetPoints` values are read from the `setpoints_channel` using the `?` expression. This expression blocks the execution of the process until a value is available and writes it to the `setpoints` variable. The `calculate_rotor_speeds` function is a library function that calculates the rotor settings depending on the setpoints. In the following step of the process these rotor settings are written to the `settings_channel` using the `!` expression. Like the `?` expression this expression blocks until there is another process that can read from this channel so the communication can be executed. The final step of the process is calling the process, which will create a loop that causes

the process body to run again. This method of looping helps the compiler analyze the program and generate verification models (Section 5.8).

The `rotors` process has a channel of `RotorSettings` values as parameter, the `?` expression reads the value and writes the value to the `settings` variable, which is then passed into a library function `set_rotor_speeds`.

```
buffered[2] chan<uint8> rs232_channel;

void rs232_isr() {
    /* may have received > 1 char before before ISR
       is serviced */
    while (PERIPHERALS[RS232_CHAR]) {
        uint8 c = PERIPHERALS[RS232_DATA];
        rs232_channel ! c;
    }
}

process receive_command(
    chan<uint8> c,
    chan<SetPoints> setpoint_channel @
    uint8 standby) {
    uint8 current_command, value;
    static SetPoints setpoint;
    c ? current_command;
    c ? value;
    if (current_command == STANDBY_CMD) {
        setpoints_reset(&setpoint);
        standby = value;
    } else if (standby == STANDBY_DISABLED) {
        switch (current_command) {
            case 0: setpoint.yaw = value; break;
            case 1: setpoint.roll = value; break;
            case 2: setpoint.pitch = value; break;
            case 3: setpoint.speed = value; break;
        }
    }
    setpoint_channel ! setpoint;
    receive_command(standby);
}
```

Figure 5.4: Receiving RS232 data.

Channels can also be declared globally so any function, including ISRs can write values to channels, as shown in Figure 5.4. The channel is buffered, that means writes to the channel do not block. This is very important for ISRs as the execution of this function should be as fast as possible.

The `receive_process` is a separate process that reads from a channel

of characters and translates this into setpoints. A command consists of two bytes, the first byte determines which value needs to be changed, and the second byte is the value. These bytes are read with the two `read (?)` expressions. Depending on the value of the `standby` state parameter, the process is either in standby or not. If the system is in standby, it will only consider the *standby command*. Otherwise it can receive the new setpoint values. The `static setpoint` variable persists when the process is looped, when receiving a new value the old value of the object should be updated. At the end of the process the `receive_command` process is called to create a loop. The new standby value is passed as argument to set the new standby state of the process.

The recursive call and state parameters are designed to help formal verification, discussed in Section 5.8, as this code structure declares loops and state variables that can be translated to a verification model.

5.3 Definition of Processes

Defining a process is done using the `process` keyword, as shown in Figure 5.5.

```
process P(int a, int b @ int c, int d) {
    printf("%d + %d = %d\n", a, c, a + c);
    P(c + a, d + b);
}
```

Figure 5.5: A simple process declaration.

This process prints two numeric sequences. Variables `a` and `b` are not changed, while `c` and `d` are updated by the recursive call.

Processes can be instantiated by the `par` keyword described in Section 5.4. Like normal C functions, a process can have parameters, so when instantiating a process, it is possible to pass data into the process, for example channels to communicate with other processes or any other data.

In embedded systems processes often run indefinitely using an infinite loop. To capture this pattern processes can *call* themselves. This creates a recursive loop. The state variables of the loop are listed in the process parameter list after the `@` sign. If a process calls itself to recurse only the new state variable values have to be passed in the call.

Sometimes it is not possible or necessary to use state variables to keep the state during multiple loops of a process, for example when allocating an object that does not change, but needs to persist during the various cycles of the loop. In this case it is possible to use the `static` storage specifier to make the values of variables persist after a process has recursed, as shown in example Figure 5.6. To initialize such a value, or to only execute some code

once, the programmer can create a label called `process_init` followed by a (compound) statement.

```
process f() {  
    static int x;  
    process_init: { x = 0; }  
    x++;  
    f();  
}
```

Figure 5.6: Usage of the `static` specifier in a process.

5.4 Running processes in Parallel: `par`

The `par` statement runs the processes concurrently (see example in Figure 5.2). The processes are listed inside the curly brackets. The programmer can pass arguments to the processes, similarly to normal function calls. Arguments can be pointers to channels. If a process has state parameters the programmer needs to pass the default state values. The arguments and default state values are separated by the `@` sign.

The `par` statement is a blocking statement. Only when all processes are finished the execution continues. Because of this it is not necessary to join the processes (threads) at a later point.

5.5 Channels for communication between processes

Communication between processes happens over channels. Channels are the wires between the processes passing values to each other.

A channel is declared as a normal variable with a `chan<T>` type. The type is parameterized, so only values of `T` can be read from or written to the channel. An example would be `chan<int> ints;` for declaring a channel of integers called `ints`.

Writing to a channel is denoted as `channel ! expression`. This is a binary expression with on the left side a reference to a channel variable and on the right side an expression of the same type of the channel. By default the expression blocks until another process can read the value from the channel. However writing to a buffered channel does not block. The expression does not have a result value.

Writes to a channel are strictly by value, a reader can modify the value at the output but that will not change the value at the input. The programmer can still pass a pointer to a channel, but as with passing a pointer to a function the programmer has to understand that the consumer may change

the value of the object the pointer is pointing to. This semantic choice has two reasons: firstly it is similar to C function calls, where the arguments are copied into function call; secondly as with function calls it is still possible to pass a pointer if that is necessary. To illustrate, in Figure 5.3 the **rotors** process could modify one of the fields of the **settings** variable but, as it is passed by value, the original **settings** variable in the **controller** process has not changed.

The expression **channel ? variable-reference** reads a value from the channel. The left hand side of the expression is a reference to a channel and the right hand side is a reference to a variable that has the same type as the channel. The expression blocks until another process writes to the channel. The value that the other process writes to the channel is then assigned to the variable reference. The result of this expression is the received value: **channel ? var == var**.

The syntax using the **?** and **!** operators is taken from the original CSP paper [7] and subsequent languages.

5.6 Buffered Channels

In embedded software, interrupt service routines (ISR) are triggered when something happens. Interrupts can fire at any point in time and ISRs are typically small. They should not block. By default, writing to a channel blocks, that is not desired in ISRs. Adding a **buffered** storage specifier to the declaration of a channel makes the channel non-blocking. **buffered chan<int>** creates a buffered channel of **ints** with a default buffer size of 1. In general **buffered[N] chan<T>** creates a channel with values of type **T** and a buffer size of **N**. If the reader of the channel is too slow, the oldest value is overwritten, so it is still the task of the programmer to find an appropriate buffer size. If the buffer is empty, reading from the channel still blocks. Figure 5.4 already showed an example how it can be used.

5.7 Selection of a channel: **alts**

In the example of Figure 5.3 all the setpoints, like roll or speed are in one structure **SetPoints**. Instead the programmer could choose to create separate channels for each setpoint, one for the speed, one for yaw, one for pitch, and one for roll.

```

process controller2(
    chan<int> yaw,
    chan<int> roll,
    chan<int> pitch,
    chan<int> speed,
    chan<RotorSettings> settings_channel) {
    static SetPoints setpoints;
    int x;
    RotorSettings settings;
    alts {
        case yaw ? x: setpoints.yaw = x;
        case roll ? x: setpoints.roll = x;
        case pitch ? x: setpoints.pitch = x;
        case speed ? x: setpoints.speed = x;
    }
    calculate_rotor_speeds(&settings, &setpoints);
    settings_channel ! settings;
    controller2();
}

```

Figure 5.7: A process where the `alts` statement selects a value from one of the channels.

Figure 5.7 shows the controller process rewritten to use multiple channels. The `alts` statement can be used to do something when any of the channels receives a value. If one of the channels, `speed`, `roll`, `pitch`, or `yaw`, can be read, a value is read from the channel and the statement after the colon is executed.

The `alts` statement looks similar to a `switch` statement but the cases each have a channel. A `case` has the form of `case <guard> <communication>: <statement>`. The `guard` part is optional. If it is given, inside square brackets, the expression should not be 0 for the `case` to be selected. The `communication` part can have two forms: just a reference to a channel or a channel read expression. Finally if there is no guard or the guard is *true*, and the channel is ready to communicate, the `statement` is executed.

The first channel that is listed in the cases and is ready for input will be selected. If it has a read expression, the read expression will assign the value from the channel to the variable. Then the statement will be executed.

The `alts` statement will block until one of the cases is executed.

5.8 Verification

An advantage of CSP is that formal verification tools exist to verify the behavior of a program. One tool is PAT [13], which can simulate and check for certain properties, such as deadlock freeness and reachability.

Models for the PAT program are written in CSP#, their CSP language to model systems. The definitions of processes and channels in the concurrency extension of MetaC can be translated to CSP#. This makes it possible to analyze a generated model of the actual implemented code.

```
1  #define STANDBY_ENABLED 1;
2  #define STANDBY_DISABLED 0;
3  #define STANDBY_CMD 0;
4  channel chan_rs232_channel 2;
5  channel chan_setpoint 0;
6  channel chan_rotors 0;
7
8  rotors() = chan_rotors ? settings -> rotors();
9
10 controller() = chan_setpoint ? setpoints ->
11                chan_rotors ! 1 -> controller();
12
13 receive_command(standby) = chan_rs232_channel ? current_command ->
14                             chan_rs232_channel ? value ->
15                             (if (current_command == STANDBY_CMD) {
16                                 chan_setpoint ! 1 -> receive_command(value)
17                             } else if (standby == STANDBY_DISABLED) {
18                                 chan_setpoint ! 1 -> receive_command(standby)
19                             });
20
21 ISR() = chan_rs232_channel ! STANDBY_CMD ->
22         chan_rs232_channel ! STANDBY_DISABLED -> Skip;
23
24 PAR_0() = rotors() ||| controller() ||| receive_command(STANDBY_ENABLED) ||| ISR();
25
26 #assert PAR_0() deadlockfree;
```

Figure 5.8: The generated CSP# model, slightly adjusted for readability.

Figure 5.8 shows a generated CSP# model. At the top some constants and the used channels are defined. As channels in CSP# are defined globally, the channel references passed into the arguments at **par** statements are traced through the processes. The number after the channel name defines the buffer size.

Then, from line 8, the processes are declared. All irrelevant information from the MetaC code is filtered with only the communication as result. In CSP# reading from and writing to a channel has almost the same syntax as the MetaC version. The **rotors** process reads a value from the **chan_rotors** channel after which in the final step is recursively calling the process to

accept a new value again.

In the `receive_command` function the state variable, `standby`, is translated to a `CPS#` parameter. It reads two values from the `chan_rs232_channel` for the command type and the value. The `if` statement is translated as well in the `CSP# if` on line 15. This determines whether it writes some value to the setpoint channel and updates the standby state (line 16) or writes a value to the setpoint channel and recurses with the current standby value (line 18).

The `ISR` process constantly writes a command to disable standby by writing two values to the `chan_rs232_channel` channel. `Skip` is a process that terminates directly, and is only used to create valid `CSP#` syntax.

On line 24 the `PAR_0` process is created. This corresponds to the `par` statement from Figure 5.2. The last line is a default assertion to check whether the system is deadlock free.

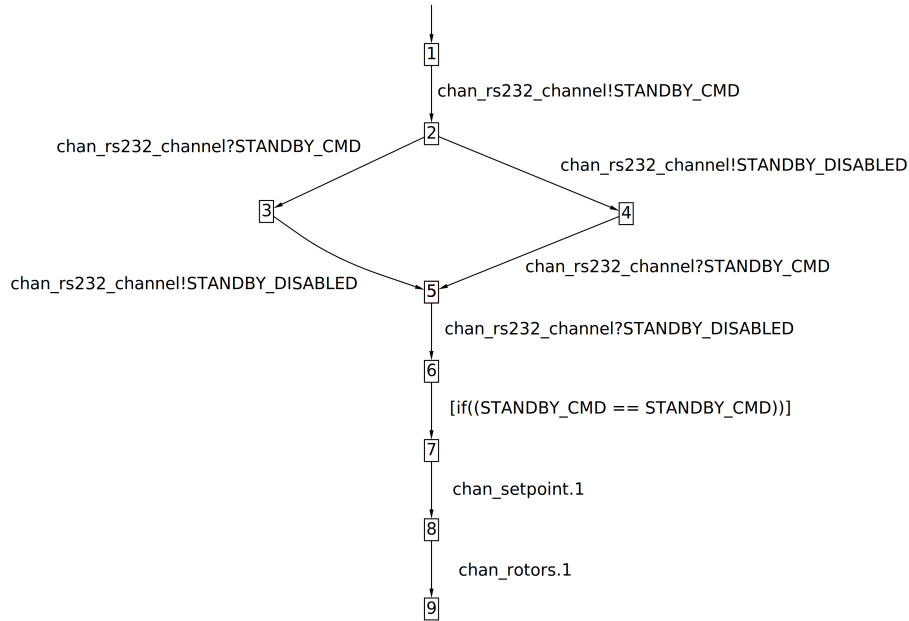


Figure 5.9: The states of the simulated `CSP#` model.

When simulating the `PAR_0` process with `PAT` the resulting image looks like Figure 5.9. Each square represents a different state of the system and the arrows are events that lead to a new state of the system.

What can be seen in this figure is that first the `STANDBY_CMD` value is written to the RS232 channel. At this point two events can happen in two different orders: either the value is directly read by the `receive_command` process, or the second value is written to the RS232 channel. This is possible

because the processes run concurrently and the RS232 channel is buffered. After either of the events happened, the other event is executed as well.

After these two events, which happened in two different orders, the second value from the RS232 channel is read. The `current_command` is substituted by `STANDBY_CMD` in the `if` (line 15). Then a value is written to and read from the setpoint channel (lines 16 and 10), so the controller can write another value to the rotors channel (line 11) that is read by the rotors process (line 8).

Modeling constraints As C code can be mixed and matched with the CSP constructs, special attention is necessary by the programmer to generate a useful model.

- Values written over a channel are assumed to be integers. Boolean and integers are the only data types PAT supports by default. Other types work fine, but are not considered in the model.
- The state of a model can be passed as *state parameters*. These parameters are defined as `process name(t1 param1, t2 param2 @ s1 state1, s2 state2)`. The state variables will be in the model. Changing the state can be done by calling the process as `name(new_state1, new_state2)` at the end of the process. The process will be executed again with the new state.
- The test expression of `if` statements can only depend on state variables or simple arithmetic expressions. Only then the model can make a deterministic choice, otherwise the model will be nondeterministic, meaning the statement could be executed or not. Consequently, the model will be able to reach certain states, that might falsify the properties the programmer likes to test, while in practice those could not happen.
- The same constraint holds for guard expressions of `alts` cases.
- Communication should not be done in C loops, instead recursive calls to execute the process again must be used.
- Other statements, that are not relevant or known to the model are, ignored.
- Model checking only works with a finite state space. So a recursive call with a continuously increasing counter will not work, as there will be a infinite number of states which a model checker (usually) cannot handle.

Currently the Concurrency MetaC extension does not warn the programmer if one of these constraints are violated. Adding a warning to an `alts` guard

when variables other than state variables or constants are used would be an example. Preventing false positives would be the challenge here in order not to irritate the programmer.

5.9 Implementation and C Runtime

The concurrency extension adds special syntax for defining processes as declarations, just like functions and variables. The syntax for channel variables is defined as a *type specifier*, just like the default types such as `int` or `float`. The read and write expressions syntax is defined as expressions, while the `par` and `alts` are defined as statements, just like `if`.

The only new name bindings are for referencing processes, for example from the `par` statement. However there are two special cases where the name binding language was not sufficient and the Stratego name binding API had to be used. Firstly, the processes need to be scoped for the current file. This is the same problem as described for state machines in Section 4.3. A strategy `get-program-scope-nabl-namespace` is also defined for processes so BaseC picks up the process namespace if the file contains processes. Consequently the processes are scoped in the enclosing file. Secondly, the recursive function call to loop the process is a function call expression where the function name is basically a variable reference. In this case the reference should refer to the process. A similar problem was solved in Section 3.4, where a Stratego strategy tries to find all possible namespaces to resolve the reference by building up a list by calling the strategy `define-variable-ns`, which can be declared at any place of any extension. The result is that Spoofox tries to look for variables, functions and processes.

The types of the write and read expressions are found by looking up the type of the variable reference. If the type of the variable is not a channel the type system will generate an error. For the write expression the type of the value that should be written to the channel is also checked, so it will not be possible to write a strange type into a channel. For reading values from a channel the type of the variable the value should be assigned too is also checked against the channel type.

The generated C code uses a library called *libcsp* [2]. It is a library that provides CSP features as a C API. This resembles the syntax of this extension, so code generation becomes a bit easier: all the complexity is inside the library, so the code generation only has to generate the variable parts. Internally the library uses POSIX threads. Using a library means adding a dependency, and the API has to be included. When the code generation detects a process is declared in a file, a C `#include <csp.h>` and `#include <process.h>` are added to the top of the generated file.

```

process f(int a @ int b) {
    static int x;
    process_init: { x = 0; }
    int y = a + b;
    f(31);
}

#include <csp.h>
#include <process.h>
typedef struct {
    signed int a, b;
} f_ProcessArgs;
void f (Process *process) {
    f_ProcessArgs *args = process->args;
    signed int a = args->a;
    signed int b = args->b;
    signed int x;
    { x = 0; }
    __csp_process_recurse: {
        signed int y = a + b;
        b = 31;
        goto __csp_process_recurse;
    }
}

```

Figure 5.10: Compilation scheme of a process.

Figure 5.10 shows how a process is transformed into C. This function has one `Process` parameter that contains all process information. This includes a pointer to the process arguments, that is declared above the function. When the process calls itself somewhere, a label is created. This label is used with a `goto` statement at the location of the call. Just before the call the arguments are assigned to the corresponding variables.

```

struct X b;
(
    CSP_chanInCopy(
        a,
        &b,
        sizeof (struct X)
    ),
    b
);

struct X b;
a ? b;

```

Figure 5.11: Compilation scheme of a channel read.

Figure 5.11 shows how a channel read is translated into C code. The C code calls a libcsp library function `CSP_chanInCopy` with the channel, a pointer to the destination variable, and the number of bytes that need to be copied from the channel to the variable: the size of the channel type. The entire expression is a comma expression, with the last item a variable reference to the destination variable, this leads that the result of the entire expression is the value from the channel.

```

struct List *b;
struct List *temp_var;
(
    temp_var = b->next,
    CSP_chanOutCopy(
        a,
        &temp_var,
        sizeof (struct List*)
    )
);

```

Figure 5.12: Compilation scheme of a channel write.

An example of how a channel write is translated is shown in Figure 5.12. If the channel type is a 32 bit integer the `CSP_chanOutInt32` library function is used, where there are also variations for 8, 16 and 64 bits. For other value types the generic `CSP_chanOutCopy` is used. The passed arguments are the variable reference to the channel object, a pointer to a variable containing the value to write to the channel and the number of bytes of the value. If the value expression of the write is just a variable expression, that variable reference can be passed as second argument, however if it is a more complex expression the expression needs to be saved in a temporary variable first, before a pointer to that temporary variable can be passed as argument.

```

void g () {
    Channel *a = NULL;
    {
        Channel __a_csp_channel;
        if (a == NULL) {
            a = &__a_csp_channel;
            CSP_chanInit(a,
                CSP_ONE2ONE_CHANNEL, 0);
        }
        Process f_proc_1;
        f_ProcessArgs f_proc_1_args;
        f_proc_1_args.a = a;
        f_proc_1_args.b = 2;
        f_proc_1.args = &f_proc_1_args;
        ProcInit(&f_proc_1, f, NULL, 0, 0);
        ProcPar(&f_proc_1, NULL);
        ProcInitClean(&f_proc_1);
        if (a == &__a_csp_channel) {
            CSP_chanClose(a);
            a = NULL;
        }
    }
}

```

Figure 5.13: Compilation scheme running processes with `par`.

The `par` statement runs processes concurrently. Figure 5.13 shows an example how the code is transformed. The `ProcPar` library function is the function to spawn and join the threads. Each argument is a pointer to a process object. At this location the arguments passed to the process are assigned to a structure object, and the process object is initialized using the `ProcInit` function, that also assigns a pointer of the process function to the object.

If channels are passed as process arguments, some channel initialization code is generated as well. As channels can be declared globally, where an initial value can only be a constant expression, they should be initialized before they are used. The channel is cleaned up after after it is used.

```

                                CSP_Alt_t alt;
                                Channel *clist[3] = {c, d, e};
                                /* assign guards to channels */
                                CSP_altInit(&alt);
                                char selected = CSP_priAltSelect(
alts {
                                &alt, clist, 3);
                                CSP_altClose(&alt);
                                case [a < b] c ? x: ; switch (selected) {
                                case d: ; case 0: /* chanInCopy(...) */
                                case [1] e: ; case 1: /* chanInCopy(...) */
                                break;
                                case 2: /* chanInCopy(...) */
                                break;
                                }
                                }

```

Figure 5.14: Compilation scheme selecting channels with `alts`.

Figure 5.14 shows the code generation of `alts` statements. The main function is the `CSP_priAltSelect` which accepts a list of channels. The function blocks until one of the channels is ready to communicate. When guards are added to the cases the code generation becomes more complex. For each guard a function that returns the guard expression and a structure is generated. All the variables that are used by the guard are stored in the structure. The function and structure with variables are assigned to two different fields of the channel. If one of the channels might communicate `libcsp` calls the guard function with the variables object and if the function result is `true` the channel can be selected.

Chapter 6

Error Handling

Error handling is a key component of a reliable software system. It allows the system to detect errors and handle them correctly, for example recovering the error or by signalling an appropriate error message. Despite its importance, error handling is often the least well understood, documented and tested part of a system [3]. C does not have explicit error handling, instead typically programmers use a convention of *return codes*: functions return a special value the caller of the function has to check. The error handling extension adds special syntax for error handling allowing the compiler to do static checks.

6.1 Background

The C convention of error handling is that functions return a special value, for example `-1`, `NULL`, or another previously defined constant. The programmer using this function has to check whether this special value was returned by the function, and if so, handle the error appropriately. However the compiler will not prevent the programmer from skipping the error check.

```

/* read char from rx fifo, return -1 if no char available */
int getchar() {
    int c;
    if (optr == iptr)
        return -1;
    c = fifo[optr++];
    if (optr > FIFOSIZE)
        optr = 0;
    return c;
}

```

Figure 6.1: An example of a function, reading from an array, returning -1, to indicate that there are no new values.

```

void process_input(Process *proc) {
    int c1, c2;
    c1 = getchar();
    if (c1 == -1)
        goto fail;
    c2 = getchar();
    if (c2 == -1)
        goto fail;
    process_keys(proc, c1, c2);
    /* easy to forget 'return' to prevent the
    error handling code to always run */
    return;
fail:
    printf("could not read from the buffer\n");
}

```

Figure 6.2: How the `getchar` function is used and how the check for errors is typically done.

Figure 6.2 shows an example where input is being processed. The `getchar` function of Figure 6.1 returns a character code, or -1 if the buffer it reads from is empty. The `process_input` gets two characters or prints a message if one of the two characters is not a valid character. C code structured similarly lead to the *#gotofail* security bug in Apple's iOS operating system [14].

More modern languages have error handling constructs: Java, amongst others, has `try-catch` blocks where thrown exceptions are caught. In some functional languages `Maybe` or `Either` objects are returned by functions. In combination with pattern matching the programmer is forced by the compiler to *unpack* the object and handle all possible cases, including the error case.

The error handling extension adds new syntax for handling errors. Using this extension the programmer is forced to handle the error. Instead of propagating errors up the call stack, a simpler variant is implemented here. Instead of throwing an error that is caught and handled somewhere on the call stack, the function returns an error object the user of the function must handle directly. Firstly this forces the programmer to handle errors quickly and, secondly is more practical to implement as transformations can be kept locally.

Although other modern languages have first-class error handling support, Mbeddr does not have a similar language extension yet.

6.2 Error Function Definition

The error handling extension adds special syntax for function return types to indicate that a function could return an error. Instead of a normal return type `T`, for example `int`, the return type of a so-called *error function* is `MaybeError<T>` or `MaybeError<T,E>`. `E` is the type of an optional error value the error function can return.

```
/* read char from rx fifo, return Error if no char available */
MaybeError<int> getchar() {
    int c;
    if (optr == iptr)
        return Error();
    c = fifo[optr++];
    if (optr > FIFOSIZE)
        optr = 0;
    return c;
}
```

Figure 6.3: Modified `getchar` function using `MaybeError<int>` as return type.

Figure 6.3 uses the `MaybeError<int>` as return type. Instead of returning `-1` a new error object `Error()` is returned to indicate the function cannot read new values, otherwise the function is identical. An important effect of this change is that the `getchar` function cannot be used directly in expressions or assignments anymore as it would not type-check: the programmer has to use the function inside an `attempt-fail` statement as explained in Section 6.3.

If a function can fail due to multiple causes, a value can be passed to the `Error()` object, for example `Error(BUFFER_EMPTY)`. For this variation the function return type should be of the form `MaybeError<T,E>` where `E` is the return type of `BUFFER_EMPTY`, for instance `MaybeError<int,unsigned char>`.

Later, where this function is used, the user can retrieve this value and handle the error appropriately.

6.3 attempt-fail statement: usage of error functions

As using error functions directly will not type check, as `MaybeError` is the return type, the programmer can use something special to use the functions: the `attempt-fail` statement. This statement consists of an `attempt` code block and one or multiple `fail` code blocks.

```
void process_input(Process *proc) {
    attempt {
        int c1 ?= getchar();
        int c2 ?= getchar();
        process_keys(proc, c1, c2);
    } fail {
        printf("could not read from the buffer");
    }
}
```

Figure 6.4: Modified `process_input` function with an `attempt-fail` to retrieve the values from `getchar` or handle errors appropriately.

Inside the blocks the programmer can use normal C code. However inside the `attempt` block an *error variable declaration* can be used. This is like a normal C variable declaration: a type specifier, name identifier, and an initial value. The difference is that instead of a `=` sign a `?=` sign is used, and that the initial value is a function call to the error function. Also the variable type is `T` from `MaybeError<T>` instead of `MaybeError`. If an error variable declaration is used outside an `attempt` block, the compiler can parse the code, so there will not be a syntax error, but during analysis the compiler will generate an error as feedback to the programmer.

The variable declaration has a slightly different syntax to indicate that it is not a normal variable declaration: it does more than only assigning the return value to the variable. If the function call results in a failure, it will not assign the variable with the result, but will jump to a `fail` block.

Inside `fail` blocks the programmer can handle the errors. The `fail` block in Figure 6.4 shows the generic `fail` block, which will be used by default. Because there can be multiple error variable declarations inside the `attempt` block, with multiple different error value types, the different `fail` blocks can handle the different types of the error values. The programmer can add a parameter to the `fail` block: `fail (E e) {...}`, This block will handle the error of

a variable declaration that uses an error function where the return type is `MaybeError<T,E>`. For example `fail (int err) { printf(strerror(err)); }` will handle the error of a function returning `MaybeError<char,int>`, as the `int` error value matches the fail block parameter type. If no corresponding fail block could be found, the compiler will generate an error as feedback to the programmer.

```
enum full_error {full};
enum empty_error {empty};
MaybeError<void,enum full_error> push(int32 x) {
    /* implementation */
}
MaybeError<int32,enum empty_error> pop() {
    /* implementation */
}
void f() {
    attempt {
        void p ?= push(1);
        int32 last ?= pop();
    } fail (enum full_error e) {
        printf("The stack is full\n");
    } fail (enum empty_error e) {
        printf("The stack is empty\n");
    }
}
```

Figure 6.5: Different `fail` blocks that will handle different errors.

Figure 6.5 shows an example of two different functions with different error value types: if the `push` call fails it will jump to the fail block that has `enum full_error` as parameter type. On the other hand the `pop` function call would jump to the other fail block with the `enum empty_error` parameter type.

In the example of Figure 6.5 the `p` variable is not used. An improvement would be that this code is an intermediate, desugared, form and the functions can be used directly in expressions, for example `push(pop())`.

6.4 Implementation

The syntax definitions of the `attempt` statement adds a new statement type. The error variable declaration is also defined as a statement. If it would be declared as a normal declaration it would also be possible to use it at the top level of a file, like a global variable, and that should not be

possible.

The `MaybeError` syntax is defined as a type specifier. That makes it possible to use it at the return type position of a function declaration.

During the analysis phase, the name binding declares the error variable declaration AST nodes as normal C variables. The error value parameter of a fail block is defined as variable only in the fail code block, that scopes the variable.

The fail blocks themselves are also registered to the name binding index engine, and if present, the parameter type is stored as property. These are used by custom error handling Stratego `nabl-use-site` strategies that are called by the analysis engine. When an error variable declaration is traversed by this strategy an analysis task looks up the type of the function call, and rewrites it to the type of the error value. Another task finds all registered fail blocks of the attempt block the error variable declaration is in. The stored parameter type property is then used to filter the found fail blocks against the error value type of the function call in the error variable declaration. Finally another task looks up if there is a registered default fail block (without a error value property). The result is either a fail block with matching parameter type, the default fail block, or no result at all. If there is no result a task is created to create a new error message informing the programmer.

C Code generation translates the special error handling constructs to normal C code.

```
MaybeError<int,char> f(int x) {
    if (x == 1)
        return Error('E');
    return 0;
}

typedef union {
    int value;
    char error;
} MError_f;
unsigned char f(
    MError_f *ret,
    int x) {
    if (x == 1)
        return (ret->error = 'E', 0);
    return (ret->value = 0, 1);
}
```

Figure 6.6: Compilation scheme of an error function.

Figure 6.6 shows how an error function is transformed to C code. When the code generator matches a function that returns a `MaybeError`, the function will be transformed into a new union type. The return type of the function will be changed to `unsigned char`. This return value will be either 1 or 0 to indicate success or failure. A parameter with the type of the union is prepended to the function parameter list. This union will hold the return

value if the function succeeds, if the function return type includes an error value type the union holds the error value when the function fails. Inside the function the expressions of `returns` are transformed too. `return Error()` is transformed to `return 0`, `return Error(value)` is transformed to a comma expression, the first expression sets the `error` field of the union object to the error value, and the last expression is the 0 constant. Other return values are considered successful and are also replaced by a comma expression, but instead set the `value` field and return the 1 constant.

```

attempt {
    int a ?= f();
    int b ?= g();
} fail (char e) {
    printf("f failed: %c\n");
} fail {
    printf("g failed\n");
}

char attempt_1_e;
{ MError_f __maybe_a;
  if (!f(&__maybe_a)) {
    attempt_1_e = __maybe_a.error;
    goto attempt_fail_1;
  }
  int a = __maybe_a.value;
  MError_g __maybe_b;
  if (!g(&__maybe_b)) {
    goto attempt_fail__;
  }
  int b = __maybe_b.value;
  goto attempt_finally;
}
attempt_fail_1: {
  { printf("f failed: %d\n",
    attempt_1_e); }
  goto attempt_finally; }
attempt_fail__: {
  { printf("g failed\n"); }
  goto attempt_finally; }
attempt_finally: {}

```

Figure 6.7: Compilation scheme of an attempt statement.

The `attempt` statements are transformed to variable declarations of the fail block parameters with a unique name, a block statement containing the contents of the attempt block, labeled block statements containing the contents of the fail blocks and, a label at the end.

An error variable declaration is transformed to a variable declaration for the union type that will hold either the success or error value. The function is called by a subsequent `if` statement. The function returns 1 or 0 so can be used directly in the `if` statement. The function call result is negated, so the `true` branch of the `if` statement is only executed if the result of the error function is an error. In the branch the error value from the union object is assigned to the previously declared error parameter, if the function has an error value, and the program jumps to the corresponding label of the fail block using a `goto`. If the function was successful the branch was

not taken, so the next declaration assigns the success value to a variable. At the end of the attempt block a `goto` is necessary to jump to the end to ensure the statements of the following fail blocks are not executed. Inside the fail blocks, variable references to the parameter are transformed to use the unique variable name given to the parameter. At the end of each fail block a `goto` ensures that the program jumps to the end of the `attempt-fail` statement.

Chapter 7

Conclusion

The C programming language fits the requirements for embedded systems very well, but it is also difficult to create higher level abstractions and to write bug-free code compared to other higher level modern languages.

Domain Specific extensions to the C programming language can help writing code for embedded systems. The extensions add new abstractions with new syntax that was not possible with just C. The new syntax can prevent programmers writing bugs. Mbeddr has already shown with a use-case study that extensions to C for embedded software is a viable solution: managing complexity with increased abstractions in the code and reduced integration and testing effort. MetaC takes the ideas from mbeddr but is implemented with the Spoofox Language Workbench as a textual language with some different extensions.

MetaC consists of the BaseC module and the extensions Bit Fields, State Machines, Concurrency, and Error Handling.

BaseC The BaseC module implements a C compiler with a C code generator. This module basis the extensions are built on.

Bit Fields The Bit Fields extension provides a new abstraction for bit manipulations. Instead of using logical operators directly a programmer can declare a Bit Fields layout and name the bits. By referring to this name he or she can read or write the bits.

State Machine The State Machine extension adds an abstraction to encode the state-based behavior, often found in embedded software. The new syntax and semantics provide a framework to structure state machine code. Due to this new abstraction the MetaC compiler can not only generate C, but also state machine diagrams the programmer can check to see whether the implementation is correct.

Concurrency The Concurrency extension provides MetaC concurrent processes and channels that help with communication between the different

parts of a system. The semantics are based on the CSP algebra that makes it possible to generate models for existing model checking tools. The concurrency extension can generate models for the PAT tool that can simulate the system and test for properties such as deadlock-freeness.

Error Handling Error handling is a key component of a software system.

The Error Handling extension enables functions to return errors that must be handled by the programmer when that function is used. This avoids the problem that programmes could forget handling errors when a function returns a special return value. The return value is a common practice for C due to lack of an error handling system.

A design goal of MetaC was that extensions built in different modules that are composed to the final MetaC language. Another goal was that the extensions should feel like C as much as possible to integrate into the language as much as possible. This caused several issues with implementing the extensions using Spoofax.

- Addition of new (binary) operators (as expressions) in multiple extensions can result in grammar ambiguities as the operator precedence is not clear. BaseC has multiple SDF3 sorts for different types of expressions so the State Machine post-fix expressions can be declared without ambiguities. Otherwise the relative context-free priorities need to be declared in the top level MetaC module.
- Specialized code had to be added to scope new constructs such as bit field layouts and state machines to the enclosing file. Spoofax does not support adding those new constructs to a certain scope. When defining scopes in Spoofax, for example a function body, all new constructs would need to be known in BaseC already, which scopes variables in the function scope. As BaseC has no knowledge of what extensions are added this problem needed to be solved otherwise. By using the Spoofax name binding API directly in BaseC, BaseC can collect the Spoofax name binding namespaces that are declared by the extensions.
- In C, the name of a field in a field expression `var.field` either refers to a declaration in a structure or union. This rule is defined in BaseC. When trying to reuse this expression for bit fields the default Spoofax solution using the Spoofax name binding language for resolving the field could not be used anymore as the rule had to be changed in BaseC, but BaseC has no knowledge of the extensions. Instead a Stratego rule tries to collect all possibilities, which is now declared inside the extension without modifying BaseC.

- In the Error Handling extension resolving the right fail block required special analysis that was not possible with the name binding language. The error variable declarations are resolved to the fail blocks by the type of the error value using the name binding Stratego API directly.

These problems made it harder to develop extensions in a modular way, but could all be solved either by workarounds in the MetaC implementation using custom Stratego code or by improving Spoofax with more support for modular languages.

BaseC and the extensions are implemented in the MetaC compiler using roughly 5000 lines of code of the various Spoofax languages. The compiler parses the C language and can compile code of extensions, including the code examples in this thesis, to C code.

7.1 Future Work

The mbeddr case study showed the approach of adding extensions to C can be a viable solution. The mbeddr authors expect similar results for similar projects as mbeddr. To test if this holds MetaC should be used in similar case study where the development effort, code size and complexity, and performance can be measured.

MetaC has not been used in real-world projects yet. To improve the implementation and find potential bugs in the compiler or the generated code MetaC should be used in real projects.

The developer experience of programmers using MetaC can be improved too. With the extensions using abstractions the compiler has more knowledge about the code. Using that knowledge, the editor can give smarter error and warning messages.

The MetaC compiler generates the C code that is run on the embedded device. When a programmer needs to debug the system with debugging tools he or she will debug (setting break points, stepping through) the generated C code. With some experience the generated C code can be manually resolved to the original MetaC code, but the ideal situation would be where the system can be debugged from the IDE with only MetaC code.

Bibliography

- [1] Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178. IEEE, 2010.
- [2] Beton. libcsp - a building mechanism for CSP communication and synchronisation in multithreaded C programs. *Communicating Process Architectures*, pages 239–250, 2000.
- [3] Magiel Bruntink, Arie Van Deursen, and Tom Tourwé. Discovering faults in idiom-based exception handling. In *Proceedings of the 28th international conference on Software engineering*, pages 242–251. ACM, 2006.
- [4] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *IEEE Computer*, 42(4):42–52, 2009.
- [5] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.
- [6] Kevin Hammond and Greg Michaelson. Hume: a domain-specific language for real-time embedded systems. In *Generative Programming and Component Engineering*, pages 37–56. Springer, 2003.
- [7] Charles Antony Richard Hoare. *Communicating sequential processes*. Springer, 1978.
- [8] Lennart C. L. Kats and Eelco Visser. The Spoofox language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, 2010.
- [9] Philip Koopman. *Better embedded system software*. Drumnadrochit Education, 2010.
- [10] Robert C Martin. The open-closed principle. *More C++ gems*, pages 97–112, 1996.
- [11] David May and Richard Taylor. Occam-an overview. *Microprocessors and Microsystems*, 8(2):73–79, 1984.
- [12] Rob Pike. Go at Google: Language design in the service of software engineering. <http://talks.golang.org/2012/splash.article>, 2012. [Online; accessed 23-March-2016].
- [13] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards flexible verification under fairness. volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

- [14] Arie van Deursen. Learning from Apple’s #gotofail security bug. <http://avandeursen.com/2014/02/22/gotofail-security>, 2014. [Online; accessed 5-April-2016].
- [15] Markus Voelter, Arie van Deursen, Bernd Kolb, and Stephan Eberle. Using C language extensions for developing embedded software: A case study. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 655–674, New York, NY, USA, 2015. ACM.
- [16] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an extensible C-based programming language and IDE for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140. ACM, 2012.