Delft University of Technology

Model-Driven Software Development

MetaC

Spring 2013

Students:
Daco Harkes
Ioana Jivet
Dario Nesi
Mircea Voda

Professor:
Dr. Eelco Visser

Chapter 1

Introduction

While the C programming language provides good support for writing efficient, low-level code, it is not adequate for defining higher-level abstractions relevant to embedded software. This project addresses that problem and proposes a language inspired by mbeddr [3] that provides object-oriented functionalities to embedded software developers. The main advantage of MetaC over mbeddr is that MetaC is a textual language rather than a projectional language.

MetaC results so a domain specific language oriented to the development of reliable and highly maintainable software, equipped with extensions such as State Machine Modelling and Runtime Message Reporting that exempts programmers from writing boilerplate code.

The language has been built on the solid basis of BaseC, that is described in detail in this document, and implemented via Stratego/XT [2] and Spoofax [1] technologies.

Issues as architecture-independence and extensibility have been tackled in the Spring of 2013 at the TU Delft University, during the course of Model-Driven Software Development, by four students leaded by Professor Eelco Visser and supported by the Software Engineering Research Group for one semester.

This documentent is structured as follows. In the next section, a Conceptual view of mbeddr is given, with references to this project, in order to evidence strengths of both solution. In section 3, BaseC implementation details are described, with a description of the standard development process used to obtain executable software starting from MetaC programs, and focusing on aspects of syntax and name binding. Sections 4 and 5 contain two DSLs for State Machines Modelling and Runtime Message Reporting , along with implementation and usage aspects. The last section describes future improvements. The appendices contain a quick reference is provided to be used as handy manual.

Chapter 2

Concepts of MetaC

MetaC is inspired by mbeddr. mbeddr is a language that provides support for developing embedded software. The main difference between mbeddr and MetaC is that mbeddr uses a projectional editor while MetaC is a textual language.

mbeddr enables C programmers to use abstract concepts that are not allowed in standard C as State-Machine-based reasoning or object-oriented modularity. Those concepts can be heavily used for tailoring software to the embedded domain to produce software more maintainable and fixable. Moreover, features that are prohibited in the majority of standards have been removed, in order to keep the software stable and reliable.

mbeddr has been designed with MPS [4]. Its approach is radically different from the other embedded development tools, third parties can use the same mechanisms for building their own extensions that were used to implement C and the existing extensions. MPS has a projectional editor, directly manipulating the AST instead of parsing text, which makes building extensions to the syntax easier.

Also, mbeddr embody tools to support key aspects of the Software Engineering Process as Requirement Verification and Documentation. Via MPS it provides functionalities as such as Model Checking and Contract verification.

MetaC is a textual language instead of being a projectional language. The main advantage of this is to not need any specific environment to be used in its core functionality and to be able to use plaintext based tools such as version control with the source code.

MetaC provides support to a big variety of data type that are platform independent, as mbeddr, but also allows old-style C constructs that experienced programmers can exploit to produce very efficient code.

Also, object-oriented reasoning is enabled by the mbeddr-like concept of module and this way C is enriched by features as encapsulation and inheritance that programmers can use in code in an elegant fashion.

The State Machine Modelling and Runtime Message Reporting extensions are ported from mbeddr. This way the effort for code migration from mbeddr and maintainability are minimized. Software writtin in mbeddr can be easily ported to MetaC, if the all extensions used in mbeddr are available.

Chapter 3

BaseC

BaseC is a language based on C but abstracting over tradtional .c and .h files. MetaC is BaseC plus extensions, and will be covered in the next chapters. In this chapter BaseC is described in detail.

3.1 Syntax

The top level concept in MetaC programs are modules. Modules act as namespaces and as the unit of encapsulation. The classical separation between .c and .h files doesn't exists in MetaC. The code resides in .mc files that get transformed to .c and .h files during the generation phase. A module can import other modules. The importing module can then access the exported contents of imported modules. Module contents can be exported using the keyword exported. All other syntax is simply C.

A basic module is shown in code snippet below.

```
module HelloWorld {
    int32 main() {
        return 0;
    }
}
```

3.2 Name binding and type checking

The namebinding for BaseC follows the same rules as basic C. However, the scope of entities declared inside modules without the "exported" specifier is limited and these entities are not visible outside the module.

Unlike usual C IDEs, MetaC provides **type checking** during editing, and not only after an explicit compilation. The type checking is performed on statements, expressions and declarations with initialization, that include variables, numerals and function calls. Additionally, the return types and parameter types of function calls are also type checked.

Like in C, **typedefs** define synonym types to already existing types. In MetaC, when the type checking encounters a synonym type, it recursively extracts the base type, which is then used in the type checking.

3.3 Mapping to C and Building

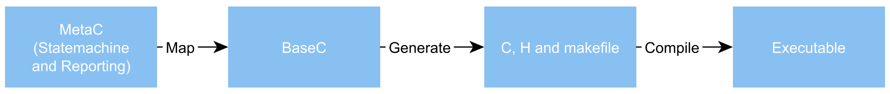

Figure 3.1: The transformation steps in compilation of MetaC

In the bigger picture the mapping from BaseC to C is called the generate step as is shown in Figure 3.1. MetaC is mapped to c and h files together with a make file. Every module generates one c file and one header file. External modules only generate a header file (see next subsection for that). This mapping is based on the way mbeddr maps its modules to c, h and makefile.
The c corresponding to a MetaC module contains:

- Its corresponding header file
- The header files of the imported modules
- Declarations
- Function definitions
- Structs, Unions and Enums

The accompanying header file contains:

- standard ifndef define
- all external functions

The makefile contains:

- All object file targets for all modules

For multiple files the imports are chased iteratively as is shown in Figure 3.2. A check is done to ensure modules referenced more than once is only included once in this process. Cyclic references are solved the same way.
After the c files, the headers files and the makefile are generated make with gcc is called, which compiles the program into an executable. This can bee seen in Figure 3.3 which also shows the last part of the complete build process.

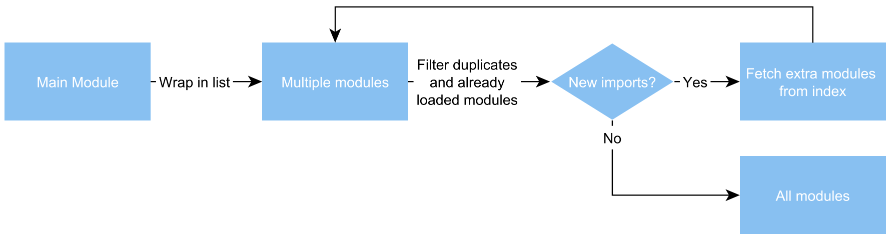
Main Module
Wrap in list
Multiple modules
Filter duplicates and already loaded modules
New imports?
Yes
Fetch extra modules from index
No
All modules

Figure 3.2: Tracing imports

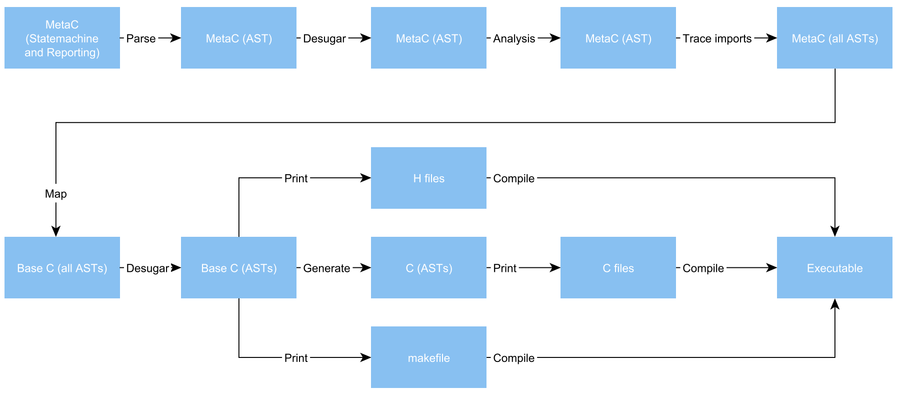

Figure 3.3: The complete build process of MetaC

3.3.1 Example program with two modules

Here we show an example with two modules. The MetaC files (*.mc) show the source code, and the other files the generated c, h and make-file(s) which are fed to gcc.

test52.mc

```
module test52 imports factorial{
    exported int32 main(int32 argc, string[] argv) {
        printf("5! is %d \n", fact(5));
        return 0;
    }
}
```

factorial.mc

```
module factorial{
    exported int32 fact(int32 x){
        if (x <= 1)
            return 1;
        return x * fact(x-1);
        }

}
```

test52.h

```
#ifndef TEST52
#define TEST52

#include <stdint.h>

int32_t main (int32_t argc, int8_t * argv []);

#endif
```

test52.c

```
#include "test52.h"
#include <stdlib.h>
#include "factorial.h"
int32_t main (int32_t argc, int8_t * argv [])
{
   printf("5! is %d \n", fact(5));
   return(0);
}
```

factorial.h

```
#ifndef FACTORIAL
#define FACTORIAL

#include <stdint.h>

int32_t fact (int32_t x);

#endif
```

factorial.c

```
#include "factorial.h"
#include <stdlib.h>
int32_t fact (int32_t x)
{
   if((x <= 1))
      return(1);
   return((x * fact((x - 1))));
}
```

makefile

```
CC=gcc
CFLAGS=-std=c99
ODIR=./bin
_OBJ_test52=test52.o factorial.o
OBJ_test52=$(patsubst %,$(ODIR)/%,$(_OBJ_test52))

all: removeStuffFromLibraries clean test52
.PHONY: removeStuffFromLibraries all clean
removeStuffFromLibraries:

$(ODIR)/%.o: %.c
              mkdir -p $(ODIR)
       $(CC) $(CFLAGS) -c -o $@ $<
debug: CFLAGS +=-g
debug: clean test52
test52: $(OBJ_test52)
       $(CC) $(CFLAGS) -o $@ $^
clean:
       rm -rf $(ODIR)
```

Note that all header files include *stdint.h* and that all .c files include *stdlib.h* because these are needed for the test programs without external modules. In the future these should be removed.

3.4 External: standard C libraries

Standard C libraries can be included by using external modules. These external resources can be imported by using the resource keyword. External modules are not compiled themselves, only the libraries they reference are imported. The remainder of this section contains one example with the resulting c code showing how this works in practise.
test50.mc

```
module test50 imports c_stdio{
   exported int32 main(int32 argc, string[] argv){
      int32 a = 3;
      printf("%d",a);
      return 0;
   }
}
```

c_stdio.mc

```
external module c_stdio resources stdio.h {
   void printf(string formatString, int32 arg1);
}
```

test50.h

```
#ifndef TEST50
#define TEST50
#include <stdint.h>
int32_t main (int32_t argc, int8_t * argv []);
#endif
```

test50.c

```
#include "test50.h"
#include <stdlib.h>
#include "c_stdio.h"
int32_t main (int32_t argc, int8_t * argv [])
{
   int32_t a = 3;
   printf("%d", a);
   return(0);
}
```

c_stdio.h

```
#include <stdio.h>
```

makefile

```
CC=gcc
CFLAGS=-std=c99
```

```
ODIR=./bin
_OBJ_test50=test50.o
OBJ_test50=$(patsubst %,$(ODIR)/%,$(_OBJ_test50))

all: removeStuffFromLibraries clean test50
.PHONY: removeStuffFromLibraries all clean
removeStuffFromLibraries:

$(ODIR)/%.o: %.c
               mkdir -p $(ODIR)
       $(CC) $(CFLAGS) -c -o $@ $<
debug: CFLAGS +=-g
debug: clean test50
test50: $(OBJ_test50)
       $(CC) $(CFLAGS) -o $@ $^
clean:
       rm -rf $(ODIR)
```

Chapter 4

Reporting DSL

The implementation of the reporting DSL is relatively straightforward. The message constructor has the form *Message(id, paramList, modifier, messageText)* where messageText is a string representing the contents of the message.

The example below shows the parsing of a message declaration.

```
INFO HelloWorld() active: "Hello, World!"
```

```
Message(
   Identifier("HelloWorld"),
   [],
   MessageActive(),
   String("\"Hello, World!\"")
)
```

During indexing, the last term of the constructor (the message text) is stored in the index as **MsgText()** data. This term (messageText) is later retrieved from the index during the to-basec phase (see the complete build process of MetaC) in order to translate **Report** statements into **printf()** function calls. The messageText is used as the parameter for the **printf()** call.

4.1 Example

```
module HelloWorld {
   exported int32 main(int32 argc, string[] argv) {
      report m.HelloWorld();
      return 0;
   }
   messagelist m {
      INFO HelloWorld() active: "Hello, World!"
   }
}
```

The report statement:

```
report m.HelloWorld();
```

will be translated to:

```
printf("Hello, World!");
```

Chapter 5

State Machine DSL

The State Machine DSL is the main C extension implemented for MetaC. This extension allows the definition and usage of State Machines inside regular MetaC code. The MetaC State Machine DSL closely follows the mbeddr implementation of State Machines.

5.1 Syntax

The syntax for the State Machine DSL follows the one used by mbeddr and can be divided into State Machine definition syntax and State Machine operation in BaseC syntax.

5.1.1 Definition

A State Machine is defined by using the following construction. The State Machine name has to be a valid identifier according to BaseC, and the initial state must be defined as a state inside the State Machine.

```
statemachine statemachine_name initial = initial_state.
```

A State Machine has several components:

- *inner variables*, declared using the keyword **var**;
- *State Machine states*, declared using the keyword **state**;
- *transition triggering events*, declared using the keyword **in**;
- *outer events*, declared using the keyword **out**;

In and out events must be declared before they can be used in states to describe transitions. Their declaration is the same as for function prototypes, except for the missing return type. In addition, the out events have to be bound to an external function, called when the outer events are triggered.

```
state beforeFlight {
    entry { points = 0; }
    on next [tp->alt > 0] -> airborne
    on report[]-> beforeFlight {printf("STATE: beforeFlight");}
    exit { points += TAKEOFF; }
}
```

State machine states have three elements, all of them optional.

- *entry* clause defines a block of instructions executed when entering the state;
- *exit* clause defines a block of instructions executed before leaving the state;
- transitions, marked by the keyword on. In the general syntax for transitions, the transition condition is expressed between square brackets as a boolean expression, with the parameters of the transition triggering event, declared at the State Machine level. The instructions that need to be executed during the transition are listed in an optional block.

```
on <trigger_event> [ <condition> ] -> <target_state> <block?>
```

5.1.2 Integration with BaseC

In BaseC, State Machines are defined inside modules, next to structs and functions. Similarly to structs, a State Machine definition generates a new type. Further, one can declare variables that have State Machine types previously defined.

```
module AnalyzeFlight{
    statemachine analyzeFlight initial = beforeFlight {
        ...
    }
    exported int8 main(int32 argc, string[] argv) {
        statemachine analyzeFlight smvar;
        return 0;
    }
}
```

Handling State Machine variables is done through three predefined functions: two translate into BaseC statements (sminit and smtrigger), and one into a BaseC expression (smIsInState).

- **sminit(smvar);** initializes the internal variables of the State Machine and sets the initial state. The only parameter points to a variable of a State Machine type.
- **smtrigger(smvar, report());** sends an event to the State Machine. The two parameters point to the State Machine variable and the in event that needs to be analyzed.
- **smIsInState(smvar, statename);** checks whether the State Machine's current state is the one indicated by the second parameter. The *statename* parameter is an identifier describing an existing state of the State Machine.

5.2 Type checking and name binding

Since the State Machine DSL is built upon BaseC, most of the typechecking and name binding are inherited from BaseC. The additional name binding is related to states, events, inner variables and outer events to functions defined in BaseC.

In the State Machine definition, no additional type checking is needed, other than the one inherited from BaseC. However, type checking rules are necessary when the State Machine DSL is integrated into BaseC. A State Machine definition generates a type, so it has to be integrated in the type system, similarly to structs. Further type checking is done on the call parameters of the predefined functions for State Machine handling.

5.3 Mapping to BaseC

The State Machine to BaseC mapping transforms the State Machine constructor into a list consisting of headerAST, initFunction and executeFunction.

The **headerAST** contains:

- an enum that is used to store the states of the State Machine
- an enum used to store the events that the State Machine can react to
- a struct that contains internal variables defined by the State Machine + a special field (__currentState) that is used to store the current state of the State Machine

The **initFunction** has the role of initializing the variables of the State Machine and the __currentState variable to their default values. The function has one parameter: a pointer to the State Machine variable that needs to be (re-)initialized.

The **executeFunction** is the 'engine' of the State Machine. It has 3 parameters: a pointer to the State Machine variable, the name of the event and a (double) pointer to a void variable that is used to pass the event arguments to the State Machine. The executeFunction is structured as nested switch-case statements. The outer switch statement checks the current state of the State Machine. Each case statement matches a particular state and has as a 'body' another switch statement that checks the current event. The case statements of the event switch match only the events that can be received in that state and the body of these event case statements consists of the statements contained in the entry, exit and transition blocks of the state. First, the statements from the current state's exit block are generated, followed by the statements from the appropriate transition block, a statement that changes the value of the current state variable to the corresponding next state and finally, the statements found in the next state's entry block. The transition from one state to another can also be controlled with a boolean condition. If this is the case, the statement described above are generated inside an if statement block which checks that condition.

5.3.1 Example

State Machine definition

```
statemachine counter initial = start {
    readable var int8 current = 0
    in increment(int8 delta)
    in reset()
    state start {
        entry {
            current = 0;
        }
        exit {
            printf("exit start\n");
        }
        on increment[delta<15]->
            increasing {current +=delta;}
    }

    state increasing {
        entry{
            printf("in increasing: current: %d\n",current);
        }
        on reset[] -> start {printf("statemachine reset\n");}
        on increment[delta<5 && current<5]-> increasing {current
            +=delta;}
    }
}
```

headerAST

```
enum moduleName_sm_events_counter{
    counter__event_increment,
    counter__event_reset
};
enum moduleName_sm_states_counter{
    counter__state_start,
    counter__state_increasing
};
struct moduleName_sm_data_counter{
    enum moduleName_sm_states_counter __currentState;
    int8_t current;
};
```

initFunction

```
void moduleName_sm_init_counter(
    struct moduleName_sm_data_counter * instance){

    instance->__currentState = counter__state_start;
    instance->current = 0;
}
```

executeFunction

```
void moduleName_sm_execute_counter(
    struct moduleName_sm_data_counter* instance,
    enum moduleName_sm_events_counter event,
    void** arguments){
    switch ( instance->__currentState ){
        case counter__state_start:{
            switch(event){
                case counter__event_increment:
                {
                    if((((*((int8_t*)arguments[0])))<15))
                    {
                        printf("exit start\n");
                        instance->current +=
                            ((*((int8_t*)arguments[0])));
                        instance->__currentState =
                            counter__state_increasing;
                        printf("in increasing: current: %d\n",
                            instance->current);
                        return(-1);
                    }break;
                }
            }break;
        }
        case counter__state_increasing:{
            switch(event){
                case counter__event_reset:{
                    {
                        printf("statemachine reset\n");
                        instance->__currentState =
                            counter__state_start;
                        instance->current = 0;
                        return(-1);
                    }break;
                }
                case counter__event_increment:{
                    if(((((*((int8_t*)arguments[0]))) < 5) &&
                        (instance->current < 5)))
                    {
                        instance->current +=
                            ((*((int8_t*)arguments[0])));
                        instance->__currentState =
                            counter__state_increasing;
                        printf("in increasing: current: %d\n",
                            instance->current);
                        return(-1);
                    }break ;
                }
            }break; }}}
```

Chapter 6

Future work

MetaC is a solid base for future expansions. The language provides basic functionality with modules and has the reporting and State Machine DSLs. Some suggested features can be found on the issue tracker of MetaC [1].

6.1 Import all std. C libraries automatically

One suggestion to improve MetaC is to automatically generate all external modules wrapping C standard libraries. The glibc is however full of macros, which make it non trivial to extract the external function definitions. Discussing this feature can be done on the issue tracker [2].

6.2 Import custom C libraries

An important feature for MetaC is the use of external custom C libraries. This is important as developers can keep using existing systems and build MetaC on top of these systems. As with the standard C libraries this could be done automatically, too. Based on the amount of macros in these custom libraries this might either be easy or not. Writing a generic solution would mean ignoring macros, since tracing all macro logic is nearly impossible[3].

6.3 Variable argument function definitions

In order to support variable argument functions for standard C libraries these variable argument functions need to be supported in MetaC themselves [4]. Since functions like printf are often used this would be a very appreciated improvement. For developers of embedded software this might be less important since embedded programming is usually conservative in use of features for performance and reliability.

[1]`http://yellowgrass.org/project/MetaC`
[2]`http://yellowgrass.org/issue/MetaC/39`
[3]`http://yellowgrass.org/issue/MetaC/47`
[4]`http://yellowgrass.org/issue/MetaC/43`

6.4 Changing synonym types

Currently, defining struct and State Machine variables is done by using unambiguous type names. For example:

```
struct Person john;
statemachine counter varStatemachine;
```

An improvement would be to define struct and State Machine variables without using the **struct** and **statemachine** keywords. Currently, defining a variable in this way: **Person john;** will define a variable 'john' with the type *TypeSynonym(Identifier(Person))*. This can be extended to check if any State Machines or structs have been defined with that name (Person) and change the type of the variable accordingly if this is true. This is an interesting problem because it is characteristic to text-based editors and is typically not encountered in projectional editors.

6.5 Configurable reporting DSL

Another improvement to MetaC would be to make the reporting DSL configurable. Currently, all report statements are translated into printf. However, in an embedded environment, reporting errors or creating logging information with printf does not make sense for a lot of applications. Thus, the reporting DSL should be extended to translate report statements into statements that set memory flags or update different registers.

6.6 Units

Implementing a DSL that attaches physical units to variables, as described in thee mbeddr userguide, is another interesting option for future work. Such a DSL would test the flexibility of the current typing and typechecking systems.

6.7 Making pointers safe

Another future improvement of MetaC would be making pointer operation safe. This could be implemented by disallowing pointer operations in the BaseC language and implementing them and related checks in a separate DSL that needs to be activated explicitly.

6.8 Conclusion

In conclusion, there are a lot of possible future improvements for MetaC, both by improving or extending existing systems and by implementing new features or extensions. Currently, mbeddr has a wide range of extensions available that can serve as inspiration. These include pre- and postconditions, support for requirement tracing and product line variablity. However, the focus should be on implementing extensions that offer real value to the general user, as the usability of these extensions varies based on the user's domain and interest.

Bibliography

[1] Karl Trygve Kalleberg, Eelco Visser, Adrian Johnstone, and Tony Sloane. Spoofax: An interactive development environment for program transformation with stratego/xt. In *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07)*, pages 47–50, 2007.

[2] Eelco Visser. Program transformation with stratego/xt. In *Domain-Specific Program Generation*, pages 216–238. Springer, 2004.

[3] 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.

[4] Markus Voelter and Konstantin Solomatov. Language modularization and composition with projectional language workbenches illustrated with mps. *Software Language Engineering, SLE*, 2010.

Appendix A

Quick reference

A.1 Datatypes

In table below is reported a list of data types implemented in the syntax of MetaC, with reference to the equivalent standard C data type and size of the type.

MetaC	Standard C	Size [bit]
boolean	-	-
int8	char	8
int16	short	16
int32	int	32
int64	long long	64
uint8	unsigned char	8
uint16	unsigned short	16
uint32	unsigned int	32
uint64	unsigned long long	64
float	float	32
double	double	64
string	-	-

Table A.1: MetaC and Standard C data types

A.2 Non-decimal numbers

There are three types of non decimal representation of numbers:

- binary. Binary numbers can be expressed in the form *[0][b][0-1]+*.
- Octal. Octal numbers can be expressed in the form *[0][0-7]+*.
- Hexadecimal. Hex numbers can be epxressed in the form *[0][xX][0-9a-fA-F]+*.

Appendix B

Get up and running

The readme.md[1] file can be found on the github page of the project. It contains example files.

B.1 Get Spoofax

In order to build this project get Eclipse 3.7/3.8 and install Spoofax (http://metaborg.org/wiki/spoofax/download).

The nightly version of Spoofax is required to work with the template language files (*.tmpl).

B.2 Checkout project

Clone https://github.com/metaborg/metac.git in Eclipse-Spoofax.

B.3 First build

The first build is actually building the project. Everything should work out of the box, including the template language.

[1]`https://github.com/metaborg/metac`