mbeddr C Extension Guide

Markus Voelter[1], Bernd Kolb[1], Daniel Ratiu[2], and Bernhard Schaetz[2]

[1] itemis AG
[2] Fortiss GmbH

Abstract. The mbeddr approach to embedded development is based on modular, incremental extension of the C programming language using the MPS language workbench. This paper provides an overview over how to extend the language. The paper assumes a basic familiarity with MPS and some familiarity with *using* the mbeddr C language, e.g. as explained in the user's guide. The paper outlines a set of challenges in embedded software development and then shows examples how mbeddr extensions of C can address those challenges. The main section of the paper then shows how to build such extensions.

Table of Contents

1 Introduction

1.1 Specific Challenges in Embedded Development

This section provides an overview over some of the typical challenges encountered in the development of embedded systems. We label these challenges C_1 through C_6 so we can refer to the challenges from the later sections.

C_1: Abstraction without Runtime Cost Abstractions that fit the problem domain are important for modular and maintainable software. Functional and object-oriented languages are a good first step in this direction, but embedded software can make use of additional paradigms: state-based, reactive, time-triggered and data flow are examples often used. Abstractions specific to the application domain in which the embedded system will be used are also common. In contrast to general purpose programming however, these abstractions should come with minimal runtime cost, because the target environments for embedded software are often severely resource constrained. This means that many of the abstractions have to be resolved statically during translation, resulting in an efficient C implementation. This constraint also rules out approaches based on runtime reflection or runtime meta programming for building custom abstractions.

C_2: Static Checks and Verification To build safe, secure and real-time systems, various forms of static analysis are used, from the compiler's type checks to sophisticated model checking approaches. Detecting problems at runtime is often too late — the device may be deployed in a non-accessible environment, or may have done harm to the real world already. As a consequence of C's "permissive" type system and the fundamental challenge of verifying turing-complete languages, verifying C programs is very expensive. To make use of verification, those parts of a system that have to be verified should be isolated from the rest, and expressed in a formalism that makes the relevant verification feasible. An example formalism is state machines. However, these parts of the system still have to be integrated with the rest of the system, usually written in C. Extracting these parts into a completely separate environment, such as a state chart modeling tool, results in integration problems.

Note how this requirements, together with the previous one, rules out the use of libraries. Libraries don't provide static translation of abstractions, validation and verification of the code as well as IDE support. Using the preprocessor to create statically translated abstractionsis is also not a utilizable option as this leads to brittle and unmaintainable code.

C_3: C considered Harmful While being efficient and flexible, especially for low-level, machine-dependent code, some of C's features are often considered harmful. Unconstrained casting via `void` pointers, using `int`s as Booleans or data structures like `union`s can result in hard-to-detect runtime errors. Consequently, these features of C, as well as others, are prohibited in many organizations. Standards such as MISRA-C limit the language subset to what is considered *safe*. However,

most C IDEs are not out of the box MISRA-C aware, so separate checkers have to be used and they may or may not be integrated with developers' tools. This makes it hard for developers to use the safe language subset of C efficiently.

C_4: Inclusion of Meta Data Non-trivial embedded systems often associate various kinds of meta data with program elements. Examples include physical units, data encodings and value ranges for types, access restrictions, memory mappings and access frequency restrictions for (global) variables as well as trace information from code to other artifacts, typically requirements. These meta data often form the basis for checking and analysis tools. Since C programs cannot express these meta data directly (except as comments or `pragma`s, with the obvious drawbacks), they are stored separately, often in XML files, and related back to the program via qualified names. While there may be tool support to check the consistency of these name references and to navigate between code and meta data, scattering of the information leads to unnecessary complexity and maintainability problems.

C_5: Tool Integration The diverse requirements, as well as the limitations of C discussed to far, often lead to the use of a wide variety of tools used for a single embedded development project. While separate tools for managing requirements or early design artefacts arguably make sense, this cannot be said for many other concerns in the system. These include most of the meta data discussed above as well as models expressed in alternative formalisms to support abstraction and model checking. Most COTS tools are not open enough to facilitate seamless and semantically meaningful integration with other tools leading to significant accidental complexities.

C_6: Product Line Support Most embedded systems are developed as part of product lines. This leads to two problems. First, each product line, or domain, comes with its own set of abstractions. If those can be made available to the developer directly, writing code for that domain becomes much simpler, and checks and verifications become more meaningful. We have already discuss these limitations in building custom abstractions above. A second challenge is the support for product line variability where certain (parts of) artifacts are only included in some of the products in the product line. This variability typically cross-cuts all the artifacts and tools, yet still has to be managed efficiently. Today this variability on code level is often implemented with the preprocessor leading to the problems discussed above and preventing static analysis of variant consistency.

2 Solution Approach

We address the challenges desribed above with an extensible version of the C programming language. The following section briefly discusses language extension in general and explores which kinds of extensions are necessary to address the challenges C_1 to C_6. Section 2.3 then shows how we address the challenges with the extensible C.

2.1 Language Extension

In this chapter we distinguish various approaches of language modularization and composition. Traditionally, languages have been combined by using an approach we call *referencing*. The partial programs written in different languages reside in their own files and refer to each other with references, often using qualified names. This approach is useful if the various partial programs describe concerns of the system that should be separated. For example, describing the deployment of components to hardware elements should be separated from the definition of these components, because the two concerns are specified at different times during system development, and the same set of components should be deployable to *different* hardware configurations without changing the component definition. However, for many other extensions, separation into files and cross-referencing does not work well. Syntactic integration between the concerns in necessary. We identify two approaches that provide syntactic integration: in language *embedding* we compose independent languages: the embedded language has no dependency on the language it is embedded in. In language *extension*, the extensions depend on the extended language, for example, by inheriting from language concepts defined in the base language. The extensible C language described in this paper mainly uses language extension: the extended languages depend on C as the base language.

Language extension as we understand it provides deep syntactic and semantic integration, as well as an IDE that is aware of the language extensions. It is much more than a macro system or an open compiler. The language extensions are not defined using meta programming (an approach where the base language uses its own abstractions to define the extensions). Instead we use a *language workbench*, a tool that supports the flexible definition, extension, composition and use of multiple languages. A language extension defines new syntax, type system rules and semantics. Typically the semantics of a language extension are defined by a transformation back to languages at a lower abstraction level. For example, an extension that provides *state machines* is transformed back to a `switch/case`-based implementation in C. At the end of this chain, the tree representing a pure C program is transformed to text and submitted to existing C compilers. Fig. 1 shows the process.

It is unlikely that a development organization is able to agree on *one common* set of domain-specific extensions to C. Also, the resulting language may become big and unwieldy. Thus, extensions have to be *modular*. They have to be created without modifying the original core language, and unintended interactions between independently created extensions must be taken care of. Users must be able to include only those language modules they actually need.

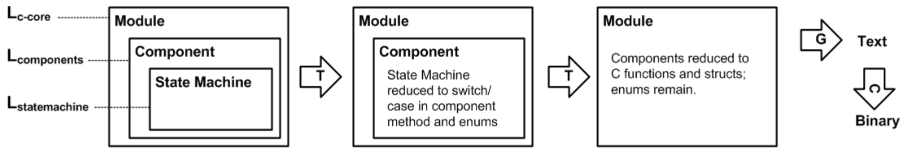

$L_{c\text{-}core}$
$L_{components}$
$L_{statemachine}$
Module
Component
State Machine
T
Module
Component
State Machine reduced to switch/ case in component method and enums
T
Module
Components reduced to C functions and structs; enums remain.
G
Text
C
Binary

Fig. 1. Abstractions expressed in higher level languages (e.g. state machines or components) are progressively reduced to their lower-level equivalent. At the end, C text is generated that is subsequently compiled.

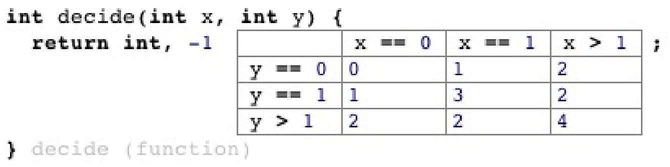
2.2 Ways to extend C

We argued above that to make language extension useful, the language syntax, type system, semantics and IDE support need to be extensible. In this section we discuss in which specific ways C needs to be extensible to make the overall approach feasible.

W_1: Top Level Constructs New top level constructs, on the level of functions and `struct`s are necessary. Examples include test cases, state machines, or interfaces and components. This enables the integration of new programming paradigms relevant in particular domains.

W_2: Statements New statements, such as an `assert` or `fail` statement in test cases, should be supported. If statements introduce new blocks, then variable visibility and shadowing must be handled correctly. Statements (and other extensions) may have to be restricted to a specific context; for example the the `assert` or `fail` statements must *only* be used in test cases.

W_3: Expressions New expressions must be addable. An example is the decision table expression that represents a two-level decision tree (Fig. 2) as a two dimensional table.



```
int decide(int x, int y) {
  return int, -1
```

	x == 0	x == 1	x > 1
y == 0	0	1	2
y == 1	1	3	2
y > 1	2	2	4

```
;
} decide (function)
```

Fig. 2. A decision table expression. It evaluates to the value in the cell for which the row and column header boolean expressions are true. If none is true, the default value `-1` is applied.

W_4: Types and Literals New types, for example for matrices, complex numbers or numbers with SI units must be addable. Adding new types also requires defining new operators or overriding the typing rules for existing operators. New literals may also be required to instantiate those types. For example, a two-dimensional table notation could be used for instantiating matrices (similar to the one in Fig. 2).

W_5: Transformation Different transformations for existing language concepts must be possible. For example, in a module marked as `safe`, a `x + y` may have to be translated to calling `addWithBoundsCheck(x, y)`.

W_6: Meta Data Decoration It should be possible to add meta data such as trace links to requirements or product line variability specifications to arbitrary any program elements.

2.3 Using the Extensible C

In this section we show how to use the extensible C language and some example extensions to address the challenges discussed above (Section 1.1). How such extensions are added to the core language will be discussed in Section 3. We address each of the challenges C_n and ways of extending C W_n with at least one example each.

A cleaned up C (addresses C_3) To make C extensible, we first had to implement C in MPS. This entails the definition of the language structure, the syntax and the type system[3]. While doing this we changed some aspects of C. Some of these changes are a first step in providing a safer C. Others changes were implemented because it is more convenient to the user or because it simplified the implementation of the language in MPS. Here are some examples.

Our C implementation provides *modules* (Fig. 3). A module contains the top level C constructs (such as structs, functions, variables, etc.). These module contents can be `exported`. Furthermore modules can *import* other modules, so they can access the exported contents of the imported modules. While header files are of course generated, we don't expose them to the user. This is an example of a simplification of C — header files are in essence just C's workaround for distinguishing public and private aspects of a C file.

We intentionally do not support the *preprocessor*. The preprocessor is used for various tasks in C, some of them legitimate (e.g. header files, constants or simple macros) and other questionable (e.g. fine-grained `#ifdef`s for variability implementation). Our C implementation provides first-class support for the legitimate uses of the preprocessor. An example of this is the module system

[3] A generator to C text is also required, so the code can be fed into an existing compiler. However, since this generator merely renders the tree as text, with no structural differences, this generator is trivial and hardly worth mentioning.

```
module Calculator {
  exported int add(int x, int y) {
    return x + y;
  }
}

module CalculatorClient imports Calculator {
  int main(char*[] args, int argc) {
    return add(1, 2);
  }
}
```

Fig. 3. Modules form the top-level container in our C. They can import other modules, whose exported contents become visible to the importing module.

mentioned in the previous paragraph as well as the support for expressing variability discussed below. Removing the preprocessor and providing more specific support for the legitimate use cases will lead to more maintainable programs.

TODO: in general the pointer syntax is not very nice...

The syntax for function pointers is a bit convoluted in C, so we have cleaned up and extended it. In particular, we support lambdas (see Fig. 4).

```
  void main(char*[] args, int argc) {
    int[] anArray = {1, 2, 3};
    forEachElement(anArray, 3, :times2);
    forEachElement(anArray, 3, [a|3 * a;]);
  }

  int times2(int a) { return 2 * a; }

  void forEachElement(int[] arr, int len, (int)=>(int) f) {
    for (int i = 0; i < len; i++) { arr[i] = f(arr[i]); }
  }
```

Fig. 4. The function `each` is a higher order function takes takes a reference to another function as its third argument. That function has to take one `int` as the argument and return an `int`. From main we call this `each` with a reference to `times2` and with a lambda that multiplies by three.

`int` is not interpreted as Boolean by default. We have introduced a separate `boolean` type. An explicit cast operator is available to support interoperability with legacy code. This is an example of a change that makes C safer and more easily analyzable.

All type annotations, such as array brackets or the pointer asterisk have to be specified on the type, not on the identifier (so you have to write `int* a;` instead of `int *a;`). We also don't support several variable declarations in the same statement (as in `int a, b, c;`. While these descisions made is eaasier for us to implement C in MPS, it also improves readability of the source code in general.

Decision Tables (adressing W_3) are a new expression. An example is shown in Fig. 2. Since expressions need a type and a value, decision tables specify the expected type of the result expressions (`int` in the example) and a default value (`-1`). A decision table is basically a two-level `if` statement. It evaluates to the value in the cell whose column and row headers evaluate to `true`.

Unit Tests (addresses W_1 and W_2) are additional a new kind of top level constructs (Fig. 5). They are introduced in a separate *unittest* language that extends the C core. Unit tests are like void functions without arguments. The *unittest* language also introduces the `assert` and `fail` statements which can only be used inside of a test case.

```
module UnitTestDemo imports multiplier {
  exported test case testMultiply {
    assert(0) times2(21) == 42;
    if ( 1 > 2 ) { fail(1); }
  }
}
```

Fig. 5. Test cases are new top level constructs. The *unittest* language also introduces the `assert` and `fail` statements which can only be used inside of a test case. The arguments to `assert` and `fail` denote the index of the statement. In error messages this number is output as a way of finding the cause in the code. These indexes are automatically projected and cannot be changed in the editor.

State Machines (addresses C_2 and W_1) provide a new top level construct as well as new statements and expressions to interact with state machines from regular C code. Entry, exit and transition actions may only access variables defined locally in state machines and set output events. This way, state machines are semantically isolated from the remaining system and can be model checked using the integrated NuSMV model checker. State machines can be connected to the surrounding C program by mapping output events to function calls and by regular C code triggering input events in the state machines. Both these aspects are not relevant to verification.

Components (addresses C_1 and W_1) are new top level constructs used for modularization, encapsulation and the separation between specification and implementation (Fig. 7). Interfaces declare operation signatures that can be implemented by components. Provided ports specify the interfaces offered by a component, required ports specify the interfaces a component expects to use. Different components can implement the same interface operations differently. Components can be instantiated, and each instance's required ports have to be connected to compatible provided ports provided by other component instances.

```
module Statemachine from cdesignpaper.statemachine {
  statemachine Counter {
    in start()
       step(int[0..10] size)
    out started()
        resetted()
        incremented(int[0..10] newVal)
    vars int[0..10] currentVal = 0
         int[0..10] LIMIT = 10
    states (initial = start)
     state start {
       on start [] -> countState { send started(); }
     }
     state countState {
       on step [currentVal + size > LIMIT] -> start { send resetted(); }
       on step [currentVal + size <= LIMIT] -> countState {
         currentVal = currentVal + size;
         send incremented(currentVal);
       }
       on start [ ] -> start { send resetted(); }
     }
  }
}
```

Fig. 6. A state machine embedded into a C module. It declares in and out events as well as local variables, states and transitions. Transitions react to in events and out events can be created in actions. State machines can be verified with the NuSMV model checker, a topic not discussed further in this paper. Through bindings (not shown) they can interact with surrounding C code, e.g. by calling functions when an out event is created.

Physical Units (addresses C_4 and W_4) are new types that, in addition to defining their actual data type, also specify the physical unit (such as m/s or N). They also specify a resolution. New literals are defined to support specifying values for these types that include the physical unit. The type checker takes into account unit compatibility. Scaling (e.g. if working with N and kN in one expression) is taken into account automatically.

Requirements Traces (addresses C_5 and W_6) Requirements traces are meta data annotations that point to requirements, essentially elements in other models imported from tools such as DOORS. Requirements traces can be applied to any program element without the definition of that element being aware of this (see Fig. 8).

Presence Conditions (addresses C_6 and W_6) Like requirements traces, presence conditions can be attached to any program element. They are boolean expressions over features in a feature model (essentially configuration switches). Their semantics is that the element they are attached to is only part of a program variant, if the boolean expression is true for the feature combination defined by that variant's feature configuration. During code generation, the program is "cut down" by removing all those elements whose condition is false. This can also be done in the editor, supporting variant-specific editing of the program.

TODO: currently we do not have them…

```
module Components {
  c/s interface Calculator {
    int multiply(int x, int y) ;
  }

  component Computer {
    provides Calculator calc
    requires LoggingService log
    int calc_multiply(int x, int y) <- op calc.multiply {
      log.info("called with " + x + " and " + y);
      return x + y;
    }
  }

  component PrimitiveComputer {
    provides Calculator calc
    int calc_multiply(int x, int y) <- op calc.multiply {
      int res = 0;
      for (int i = 0; i < y; i++) {
        res = res + x;
      }
      return res;
    }
  }
}
```

Fig. 7. Two components providing the same interface. The `<-` notation maps operations offered through provided ports to their implementation in components.

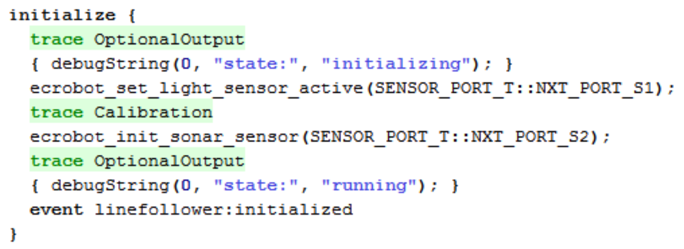
Safe Modules (addresses W_5 and C_3) Safe modules restrict the set of constructs that can be used inside them. For example, pointer arithmetics is disallowed. Errors are reported if this is attempted. In addition, runtime range checking is performed for arithmetic expressions and assignments. To enable this, arithmetic expressions are replaced by function calls that perform range checking and report errors if an overflow is detected.

3 Design and Implementation

In this section we illustrate the design of the extensible C language to explain how the challenges C_n and they ways of extending C W_n are made possible. We use the examples from Section 2.3 to illustrate the mechanisms. We start out by explaining in some detail how MPS works, to allow a more compact discussion about the design of the extensible C language.

3.1 MPS Basics

Our implementation uses the open source JetBrains MPS language workbench. MPS is a projectional editor. Projectional editors don't use parsers and grammars. Instead, editing gestures directly change an abstract representation of the program and projections render the program in a notation through which the



```
initialize {
  trace OptionalOutput
  { debugString(0, "state:", "initializing"); }
  ecrobot_set_light_sensor_active(SENSOR_PORT_T::NXT_PORT_S1);
  trace Calibration
  ecrobot_init_sonar_sensor(SENSOR_PORT_T::NXT_PORT_S2);
  trace OptionalOutput
  { debugString(0, "state:", "running"); }
  event linefollower:initialized
}
```

Fig. 8. Requirements traces can be attached to any program element of any language. An intention (pressing Alt-Enter and selecting *Add Trace*) is used to attach them.

user interacts with the program. This notation may be textual, tabular, symbolic or graphical[4].

The projection-based approach is more flexible than parser-based systems. We can use tabular or symbolic syntax (e.g. fraction bars), and syntactic composition is also not a technical problem at all. This is why, except in a few cases, we ignore concrete syntax aspects in our discussion of the design of the extensible C, we use only the abstract syntax. Defining a concrete syntax is not a challenge in the MPS environment. MPS also addresses other aspects of language definition such as type systems, transformation and refactoring via its own DSLs, optimized for those tasks. We address these aspects as well.

Projectional editing also allows preventing the user from entering language constructs in context where they are not allowed (e.g. an `assert` statement can only be used inside a test case). In addition, by adding conditions into the projection rules, program parts irrelevant in a certain situation can be hidden (e.g. we can show a program in a variant-specific way by hiding all those parts that are not included in the variant based on presence conditions).

To learn more about the mechanics of how MPS works, and details about how these mechanisms support language modularization and composition we refer to `http://voelter.de/data/pub/Voelter-GTTSE-MPS.pdf`.

3.2 The Core Languages

The equivalent of plain C and *make* is covered in the (`com.mbeddr.core.*`) languages. Fig. 9 shows the various languages and their dependencies. The `expressions` language is the most fundamental language. It depends on no other language and defines the primitive types, the corresponding literals and the basic operators. While the language serves as the core for C, expressions, types and literals are really applicable to almost any simple expression language.

[4] Traditionally, projectional editors have had a bad reputation, because, while the projection may look like text, it didn't feel this way. Editing was a pain. MPS has solved this problem to a degree where editing text feels *almost* like editing text in a parser-based environment.

Note that it does not contain anything relating to pointers and user-defined data types (`enum, struct`). Support for those is in the `pointers` and `udt` languages, respectively. `statements` defines the procedural part of C, and the `modules` language covers modularization as explained above. `unittest` contains support for unit testing, `make` is a language for makefiles and `buildconfig` provides an abstraction over `make` that works with the modules defined in the `modules` language.

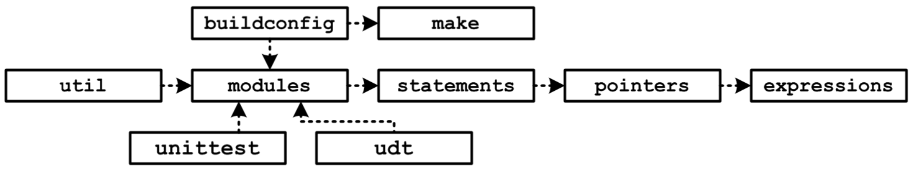


Fig. 9. Languages and their dependencies for `com.mbeddr.core` representing the C programming language plus support for unit testing and make-based build.

3.3 Addressing W_1: Top-Level Constructs

Structure The `modules` language defines the `IModuleContent` interface. This is a language concept interface that defines the properties of everything that can reside in a top-level module . It inherits from the predefined `IIdentifierNamedConcept`, which in turn provides a `name` property. `IModuleContent` also defines a boolean property `exported` that determines whether the respective module content is exported. `Module`, the container of top level constructs defines a $1..n$ child relationship `contents` of type `IModuleContent`. Language concepts such as `Function` implement `IModuleContent` allowing them to be plugged into modules. Since the `IModuleContent` interface can also be implemented by concepts in other languages, new top level constructs such as the `TestCase` can simply implement this interface, as long as the respective language has a dependency on the `modules` language.

Constraints A test case contains a `StatementList`, so any valid C statement can be used inside a test case. `StatementList`, and all other C statements become available to the unit test language since it (transitively) depends on the `statements` language. The `unittest` language also defines new statements: `assert` and `fail`. To make them usable in `StatementLists`, they simply inherit from the abstract `Statement` concept defined in the `statements` language. However, this approach makes them valid in *any* statement list, for example also in a regular function. This is undesirable, since the transformation of `asserts` into plain C depends on them being used inside a `TestCase`. To enforce this, a *can be child* constraint is defined (Fig. 10).

Transformation

```
concepts constraints AssertStatement {
  can be child
    (operationContext, scope, parentNode, link, childConcept)->boolean {
      parentNode.ancestor<concept = TestCase, +>.isNotNull;
    }
}
```

Fig. 10. This constraint restricts an `AssertStatement` to be used only inside a `TestCase` by checking that, from the assert's perspective, any of the ancestors is a `TestCase`.

Although it is part of the core, the `unittest` language conceptually sits on top of plain C. This means that the new language concepts are reduced to plain C concepts upon generation, a process also called *assimilation*. The `TestCase` is transformed to a `void` function without arguments. The `assert` statement is transformed into a `report` statement from the logging language, which is in turn transformed into a platform-specific way of actually reporting an error. Fig. 11 shows an example.

Note that some top level constructs may not be transformed at all. In the example in Fig. 11, the `report` statement references a message definition `testing.FAILED`. Log messages are defined in another top level construct, a so-called message list. It defines log messages with IDs, a severity (error, warn, info), an informative error message as well as optional data arguments. While the contents of these message definitions are used in the generated code (e.g. the informative message in the `printf` statement), the actual definition is removed as the model is transformed progressively.

```
test case exampleTest {
  int x = add(21, 21);
  assert(0) x == 42;
}
```

```
void test_exampleTest {
  int x = add(21, 21);
  report testing.FAILED(0)
      on !( x == 42 );
}
```

```
void test_exampleTest {
  int x = add(21, 21);
  if ( !( x == 42 ) ) {
    printf( "failed: 0" );
  }
}
```

Fig. 11. Two-stage transformation of `TestCase`s. The `TestCase` is transformed into a C function using the logging framework to output error messages. The `report` statement is then transformed in turn into a `printf` statement *if* we generate for the Windows/Mac environment. It would be transformed to something else if we generated for the actual target device.

3.4 Addressing W_2: Statements

We have already seen the basics of integrating statements in the previous section. In this section we focus on the handling of local variable scopes and visibilities. As a simple example, let us introduce simple support for automatically freeing memory (see Fig. 12). The newly introduced variables have to be visible only inside the body and the variables have to shadow variables of the same name that may be visible inside the body as well (in Fig. 12, this could be a variable `a` declared outside the `safeheap` statement).

Structure The `safeheap` statement extends `Statement` so it can be used in function bodies and other statement lists. It contains another `StatementList` as its body, as well as a list of `SafeHeapVar`s. These extend `LocalVarDecl`, so they can be plugged into the mechanism for handling variable shadowing.

```
void aFunction() {
  int* a;        // this one should not be visible from the safeheap's body
  safeheap(int* a, int* b) {
    *a = 10;     // this is the a from the safeheap statement
    b = a;
  }
  *a = 42;        // this is the a declared outside the safeheap statement
}
```

Fig. 12. A `safeheap` statement declares a number of heap variables. These are then automatically allocated and can be used inside the body of the statement. When the body is left, the memory is automatically freed.

Behaviour `LocalVarRef`s are expressions that reference `LocalVarDecl`. A scope constraint determines the set of visible variables. This constraint ascends the containment tree until it finds a node which implements `ILocalVarScopeProvider` and calls its `getLocalVarScope` method which returns a `LocalVarScope` object which in turn can be asked for its visible local variables. A `LocalVarScope` has reference to an outer scope, which is created by finding *its* ancestor that implements `ILocalVarScopeProvider`. This way, a hierarchy of `LocalVarScope` objects is created. To get at the list of actually visible variables, the `LocalVarRef` scope constraint calls the `getVisibleLocalVars` method on the `LocalVarScope` object. This method returns a flat list of `LocalVarDecl`s, taking into account that variables of a `LocalVarScope` that is lower in the hierarchy shadow variables of the same name from a higher level in the hierarchy.

So, to plug the `SafeHeapStatement` into this mechanism, it has to implement `ILocalVarScopeProvider` and implement its two methods to return correctly filled `LocalVarScope` objects. The code is shown in Fig. 13.

3.5 Addressing W_3: Expressions

Expressions are different from statements in that they have *values* as the program executes. During editing and compilation, the *type* of an expression is relevant for the static correctness of the program. So extending a language regarding expressions requires extending the type system rules as well.

```
public LocalVarScope getLocalVarScope(node<> context, int statementIndex) {
  LocalVarScope scope = new LocalVarScope(getContainedLocalVariables());
  node<ILocalVarScopeProvider> outercScopeProvider =
      this.ancestor<concept = ILocalVarScopeProvider>;
  if (outercScopeProvider != null) {
    scope.setOuterScope(outercScopeProvider.getLocalVarScope(this, this.index));
  }
  return scope;
}

public sequence<node<LocalVariableDeclaration>> getContainedLocalVariables() {
  this.vars;
}
```

Fig. 13. A `safeheap` statement implements the two methods declared by the `ILocalVarScopeProvider` interface. `getContainedLocalVariables` returns the `LocalVarDecls` that it declares between the parentheses (see Fig. 12). `getLocalVarScope` constructs a scope that contains these variables and then builds the hierarchy of outer scopes by relying on its ancestors that implement `ILocalVarScopeProvider`.

Fig. 2 shows the decision table expression. It's value is the expression in cell c if the expressions in the column header of c and the row header of c are true. So, in Fig. 2, the value of the table is 4 if $x > 1$ and $y > 1$. If none of the condition pairs is true, then the default value, -1 in the example, is used. A decision table also specifies the type of the value it will evaluate to, `int` in the example, so all the expression in content cells (those that are not column or row headers) have to be compatible with that type. The type of the headers has to be boolean, of course.

Structure To be able to use the decision table as an expression, it has to extend the `Expression` concept defined in the core language. It contains a list of expressions for the column headers, one for the row headers and one for the result values. It also contains a child of type `Type` to declare the type of the result expressions, as well as an expression as the default value.

```
typeof(dectab) :==: typeof(dectabc.type);  // the type of the whole decision
                                           // table is the type specified in the
                                           // type field
foreach expr in dectab.colHeaders {        // for each of the expressions in
  typeof(expr) :==: <boolean>;             // the column headers, the type
}                                          // must be boolean
foreach expr in dectabc.rowHeaders {       // ... same for the row headers
  typeof(expr) :==: <boolean>;
}
foreach expr in dectab.resultValues {      // the type of each of the result
  infer typeof(expr) :<=: typeof(dcectab); // values must be the same or a
}                                          // subtype of the table itself
typeof(dc.def) :<=: typeof(dectab);        // ... same for the default
```

Fig. 14. The type calculation equations for the decision table (see the comments for details).

Type System MPS uses a unification for the type system. A language simply specifies a set of type equations (Fig. 14) that contain type literals (such as `boolean` as well as type variables (such as `typeof(dectab)`). The unification engine then tries to assign type values to the type variables so that all type equations become true. Extending the type system of an existing language is simply a matter of specifying additional typing rules that are solved along with the tyoing rules of the existing language. For example, the typing rules for a `ReturnStatement` make sure that the type of the returned expression are the same or a subtype of the type of the surrounding function. If a `ReturnStatement` uses a decision table as the returned expression, the typing rules for decision tables have to be solved along with those for the `ReturnStatement`.

3.6 Addressing W_4: Types and Literals

To illustrate the addition of new types and literals we use physical units (see Fig. 15).

```
module Units  {
  unit kg for int;
  unit lb for int;

  exported test case testUnits {
    kg/int m1 = 10kg;
    lb/int m2 = 10lb;
    assert(0) m1 + 10kg == 20kg;
    // the following reports an error in the IDE (adding kg and lb)
    assert(1) m2 + 10kg == 20kg;
  }
}
```

Fig. 15. The *units* extension supports data types and literals with physical units. A `UnitDeclaration` specifies the unit and the base type. Type checks ensure that the values associated with unit literals are the correct type (`10.2kg` would not be allowed in the example). The typing rules for the existing `+` and `==` operators must be overridden to support types with units.

Structure `UnitDeclaration` are top-level concepts, so they simply implement `IModuleContent`. They have a name and the `Type` to which they apply. Introducing new types (so one can write `kg/int m = ...`) is done by introducing a `UnitType` which extends the `Type` concept. The `UnitType` refers to the `UnitDeclaration` whose type it represents. A scoping rule must be defined control which `UnitDeclaration`s are visible. We discuss this in the next paragraph. We also have to introduce `LiteralWithUnits` (as in `10kg`). These are `Expressions` that have a child of type `Literal` as their `value` and also a reference to a `UnitDeclaration` (scoping also explained in the next paragraph).

Scoping The `LiteralWithUnit`s as well as the `UnitType`s have to refer to an existing `UnitDeclaration`. `UnitDeclaration`s are defined at the top level within modules. According to the visibility rules in our C language, valid targets for the reference are those in the same module, and the *exported* ones in all imported modules. This general rule holds for references to all module contents, and hence is implemented in a reusable fashion. Fig. 16 shows code that has to be specified as the scope of the reference to the `UnitDeclaration`. We use an interface `IVisibleElementProvider`, implemented among others by `Module`s, to find all instances of a given type. The implementation of this method simply searches through the contents of the current and imported modules to fins instances of the passed in concept. The result collection is then used as the scope for references for the respective element.

```
link {unit}
  search scope:
    (model, scope, referenceNode, linkTarget, enclosingNode, operationContext)
                      ->join(ISearchScope | sequence<node<UnitDeclaration>>) {
      enclosingNode.ancestor<concept = IVisibleElementProvider>.
             visibleContentsOfType(concept/UnitDeclaration/).
             select({~it => it : UnitDeclaration; });
    }
```

Fig. 16. The `visibleContentsOfType` operation returns all instances of the passed in concept in the current module, as well as all exported instances in modules imported by the current module. The `select` clause casts down each content to a `UnitDeclaration`, which is safe, because we pass in `concept/UnitDeclaration/` to `visibleContentsOfType`.

Type System As we have seen, MPS uses type system equations and unification for specifying type system rules. However, there is special support for binary operators that makes overloading them for new types especially easy: overloaded operations containers. These basically use 3-tuples of *(leftArgType, rightArgType, resultType)* plus applicability conditions. The overloaded operations containers are additive: typing rules for new (combinations of) types can simply be added. Fig. 17 shows the typing rules for the `PlusExpression`, an operator defined in the core C language that should be overloaded for unit types. The type of a `PlusExpression` that adds two `UnitType`s will be a `UnitType` as well.

Editing Experience A drawback of projectional editing is that special care has to be taken to make sure the editing experience feels as much like normal text editing as possible. For example if the have an existing number literal such as `20` we want to be able to just type `kg` on its right side to transform it into a `LiteralWithUnit` with the value `20` and the unit `kg`. This can be achieved with a so-called *right transformation* for literals. The one needed that handles the case described above is shown in Fig. 18. It basically replaces the `Literal` with a `LiteralWithUnit` if it detects the name of a unit on the right side of a `Literal`.

```
operation concepts: PlusExpression
left operand type: new node<UnitType>()
right operand type: new node<UnitType>()
is applicable:
  (operation, leftOperandType, rightOperandType)->boolean {
    return leftOperandType : UnitType.unit == rightOperandType : UnitType.unit;
  }
resulting type:
  (operation, leftOperandType, rightOperandType)->node<> {
    leftOperandType.copy;
  }
```

Fig. 17. An overloaded operations container declares the language concept and the types of the left and right arguments for which this rule applies, plus an optional applicability condition. In this case we make sure we can only add two numbers if the use *the same* unit (more sophisticated unit compatibility rules could be added here). The resulting type is a copy of the left operand's type (same as the right one, so we can choose either).

3.7 Addressing W_5: Alternative Transformations

We use the example of *safe* modules and runtime range checking. Range checking is implemented by replacing binary operators like `+` or `*` with calls to functions that, in addition to perfoming the addition, check the range. The solution approach is as follows: we use an *annotation* to mark `Modules` as safe. If a module is safe, we perform a transformation of the binary operators into function calls.

Annotations are concepts whose instances can be added as children of other nodes without this being declared in the parent node's concept definition. The `safe` annotation can be attached to `Module`s. The transformation that replaces the binary operators with function calls is triggered by the presence of this annotation on the `Module` which (transitively) contains the operator. Fig. 19 shows the transformation code. Using transformation priorities, it is arranged that this transformation runes *before* the final transformation that maps the C "model" to actual C text for subsequent compilation via GCC.

3.8 Addressing W_6: Meta Data

It is desirable to be able to attach meta data to any program element. We define meta data as data that is not required from the core transformation's point of view, but useful to specialized optional tools. Hence it is important that it is possible to attach such meta data to program nodes without the definition of these nodes have to be aware of that (see Fig. 8). We have already come across MPS' approach of this in the previous section: annotations. We use the same approach for meta data.

Structurally, annotations become children of the node they attached to. The annotation's definition declares the name of the link that contains these additional children. Visually, in the editor, annotations look as if they surround the target element. Fig. 20 shows the definition of the editor. Note that because of the projectional nature of the editor there is no grammar that needs to be

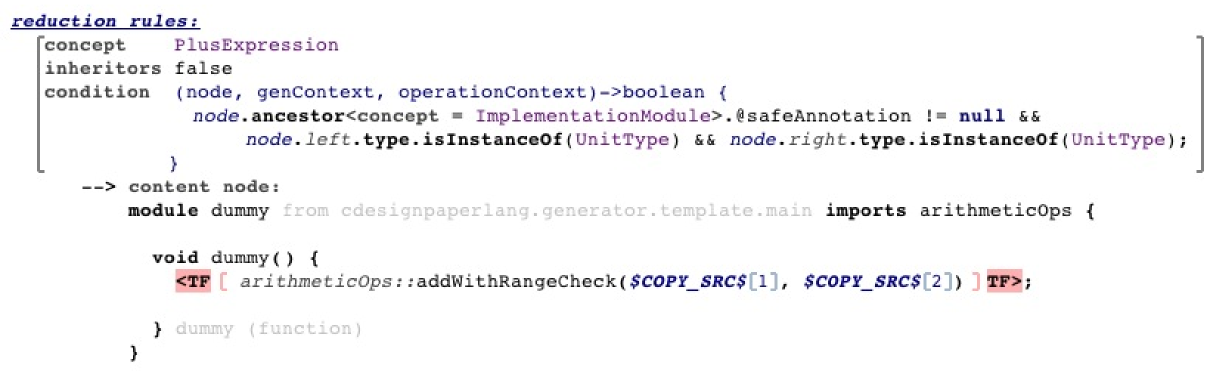
```
right transformed node: Literal
initializer:
    sequence<node<UnitDeclaration>> units = // get all visible units, like in scope above
actions :
  add custom items  (output concept: LiteralWithUnit)

    matching text
      (operationContext, scope, model, sourceNode, pattern)->string {
         foreach u in units {
           node<UnitDeclaration> ud = ((node<UnitDeclaration>) u);
           if (ud.name.startsWith(pattern.toString())) {
             return ud.name;
           }
         }
         return null;
      }

    do transform
       (operationContext, scope, model, sourceNode, pattern)->node<> {
         foreach u in units {
           node<UnitDeclaration> ud = ((node<UnitDeclaration>) u);
           if (ud.name.equals(pattern.toString())) {
             node<LiteralWithUnit> lwu = new node<LiteralWithUnit>();
             sourceNode.replace with(lwu);
             lwu.value = sourceNode;
             lwu.unit = ud;
             return lwu;
           }
         }
         return sourceNode;
      }
```

Fig. 18. This is the *right transformation* which transforms a `Literal` into a `LiteralWithUnit` if the name of a `UnitDeclaration` is typed in on the right side of a `Literal`. The `matching test` block checks if one of the visible unit declaration starts with the text that has been entered so far. If so, the `do transform` block is executed, which, once we finished entering a complete name, replaces the `Literal` with a new instance of `LiteralWithUnit` using the original `Literal`'s value and the unit whose name was entered.

made aware of the additional text in the program. This means that arbitrary annotations can be added to arbitrary program nodes.

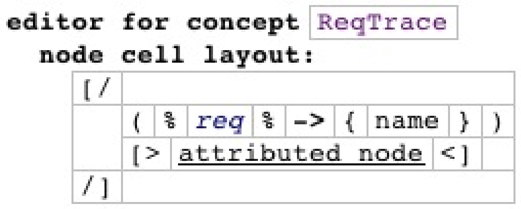


```
reduction rules:
  concept     PlusExpression
  inheritors false
  condition   (node, genContext, operationContext)->boolean {
                node.ancestor<concept = ImplementationModule>.@safeAnnotation != null &&
                    node.left.type.isInstanceOf(UnitType) && node.right.type.isInstanceOf(UnitType);
              }
     --> content node:
         module dummy from cdesignpaperlang.generator.template.main imports arithmeticOps {

           void dummy() {
             <TF [ arithmeticOps::addWithRangeCheck($COPY_SRC$[1], $COPY_SRC$[2]) ] TF>;

           } dummy (function)
         }
```

Fig. 19. This *reduction rule* transforms `PlusExpression` into a call to a library function `addWithRangeChecks`, passing in the left and right argument of the `+` using the two `COPY_SRC` macros. Using the `condition` expression, the transformation is only executed if the containing `Module` has a `safeAnnotation` attached to it and the type of both arguments of the `+` are `UnitType`s.



```
editor for concept ReqTrace
  node cell layout:
    [/
       ( % req % -> { name } )
       [> attributed node <]
    /]
```

Fig. 20. The editor definition for the `ReqTrace` annotation. It consits of a vertical list `[/ .. /]` with two lines. The first line contains the reference to the reference requirement. The second line uses the `attributed node` construct to "delegate" to the editor of the program node to which this annotation is attached. So the annotation is always rendered right on top of whatever syntax the orginal node uses.