*20*

## DSLs in the Implementation

|  |  |
| --- | --- |
| In this section we discuss the use of DSLs in the context of implementation. There it is crucial that the DSLs are tightly integrated with the application code that is typically written in a GPL. Language embedding and extension (Section 4.6) are obviously useful approaches. In this chapter we discuss the |  |
| mbeddr system1 which supports domain-specific extensions to |  |
| C2. |  |
| The amount of software embedded in devices is growing and the development of embedded software is challenging. In addition to functional requirements, strict operational requirements have to be fulfilled as well. These include reliability (a device may not be accessible for maintenance after deployment), safety (a system may endanger life or property if it fails), efficiency (the resources available to the system may be limited) |  |

*In this chapter we discuss the use of DSLs in the context of software implementation, based on an extensive case study in embedded software development: the mbeddr system that has been discussed in the book before. mbeddr supports extension of C with constructs useful for embedded software. In this chapter we show how language extension can address the challenges of embedded software development, and report on our experience in building these extensions.*

This section of the book is based on a paper written together with Daniel Ratiu, Bernd Kolb and Bernhard Schaetz for the SPLASH/Wavefront 2012 conference.

### 20.1 Introduction

|  |  |
| --- | --- |
| Current approaches for embedded software development can be divided roughly into programming and modeling. The *pro-* |  |
| *gramming* approach mostly relies on C, sometimes C++, and Ada in rare cases4. However, because of C’s limited support |  |
| for defining custom abstractions, this can lead to software that is hard to understand, maintain and extend. Furthermore, C’s ability to work with very low-level abstractions, such as point- |  |
| ers, makes C code very expensive to analyze statically5. The |  |
| alternative approach uses *modeling* tools with automatic code generation. The modeling tools provide predefined, higherlevel abstractions such as state machines or data flow compo- |  |
| nent diagrams6. Example tools include ASCET-SD7 or Simu- |  |

or real-time constraints (a system may have to run on a strict schedule prescribed by the system’s environment). Addressing these challenges requires any of the following: abstraction techniques should not lead to excessive runtime overhead; programs should be easily analyzable for faults before deployment; and various kinds of annotations, for example for describing and type checking physical units, must be integrated into the code. Process issues such as requirements traceability have to be addressed, and developers face a high degree of variability, since embedded systems are often developed in the context of product lines3.

link8. Using higher-level abstractions leads to more concise programs and simplified fault detection using static analysis and model checking (for example using the Simulink Design Verifier9). Increasingly, DSLs are used for embedded software, and studies show that DSLs substantially increase productivity in embedded software development. However, most real-world systems cannot be described completely and adequately with a single modeling tool or DSL, and the integration effort between manually written C code and perhaps several modeling tools and DSLs becomes significant.

A promising solution to this dilemma lies in much tighter integration between low-level C code and higher-level abstractions specific to embedded software. We achieve this with an extensible C programming language. The advantages of C can be maintained: existing *legacy* code can be easily integrated, reused, and evolved, and the need for *efficient* code is immediately addressed by relying on C’s low-level programming concepts. At the same time, domain-specific extensions such as state machines, components or data types with physical units

can be made available as C extensions. This improves *productivity* via more concise programs, it helps improve *quality* in a constructive way by avoiding low-level implementation errors up front, and leads to system implementations that are more amenable to *analysis*. By directly embedding the extensions into C, the mismatch and integration challenge between domain-specific models and general-purpose code can be removed. An industry-strength implementation of this approach must also include IDE support for C and the extensions: syntax highlighting, code completion, error checking, refactoring and debugging.

The LW-ES research project, run by itemis AG, fortiss GmbH, BMW Car IT and Sick AG explores the benefits of language engineering in the context of embedded software development with the mbeddr system10.

.

|  |  |
| --- | --- |
| *20.2 Challenges in Embedded Software*  In this section we discuss a set of challenges we address with the mbeddr approach. We label the challenges *Cn* so we can refer to them from Section 20.3.2, where we show how they are |  |
| addressed by mbeddr11. |  |
| *C*1*: Abstraction without Runtime Cost* Domain-specific concepts provide more abstract descriptions of the system under development. Examples include data flow blocks, state machines, or data types with physical units. On one hand, adequate abstractions have a higher expressive power that leads to programs that are shorter and easier to understand and maintain. On the other hand, by restricting the freedom of programmers, domain-specific abstractions also enable constructive quality assurance. For embedded systems, where runtime efficiency is a prime concern, abstraction mechanisms are needed that can be resolved before or during compilation, and not at runtime.  *C*2*: C considered Unsafe* While C is efficient and flexible, |  |
| several of C’s features are often considered unsafe12. Conse- |  |

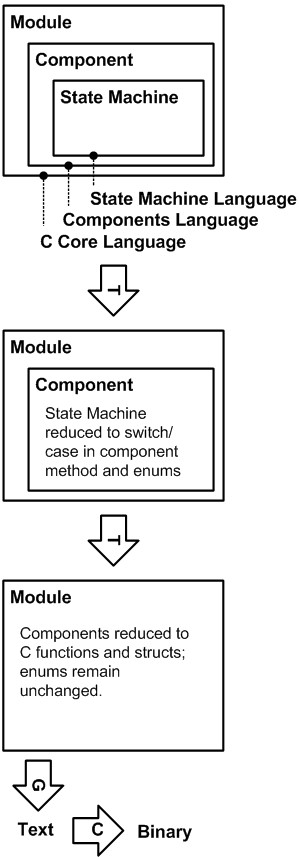
quently, the unsafe features of C are prohibited in many organizations. Standards for automotive software development such as MISRA limit C to a safer language subset. However, most C IDEs are not aware of these and other, organization-specific restrictions, so they are enforced with separate checkers that

are often not well integrated with the IDE. This makes it hard for developers to comply with these restrictions efficiently.

|  |  |
| --- | --- |
| *C*3*: Program Annotations* For reasons such as safety or efficiency, embedded systems often require additional data to be associated with program elements. Examples include physical units, coordinate systems, data encodings or value ranges for variables. These annotations are typically used by specific, often custom-built analysis or generation tools. Since C programs can only capture such data informally as comments or **pragma**s, the C type system and IDE cannot check their correct use in C programs. They may also be stored separately (for example, in XML files) and linked back to the program using |  |
| names or other weak links13. |  |
| *C*4*: Static Checks and Verification* Embedded systems often have to fulfill strict safety requirements. Industry standards for safety such as ISO-26262, DO-178B or IEC-61508 demand that for high safety certification levels various forms of static analyses are performed on the software. These range from simple type checks to sophisticated property checks, for example by model checking. Since C is a very flexible and relatively weakly-typed language, the more sophisticated analyses are very expensive. Using suitable domain-specific abstractions (for example, state machines) leads to programs that can be analyzed much more easily. |  |

*C*5*: Process Support* There are at least two cross-cutting and process-related concerns relevant to embedded software development. First, many certification standards (such as those mentioned above) require that code be explicitly linked to requirements such that full traceability is available. Today, requirements are often managed in external tools, and maintaining traceability to the code is a burden to the developers and often done in an ad hoc way, for example via comments. Second, many embedded systems are developed as part of product lines with many distinct product variants, where each variant consists of a subset of the (parts of) artifacts that comprise the product line. This variability is usually captured in constraints expressed over program parts such as statements, functions or states. Most existing tools come with their own variation mechanism, if variability is supported at all. Integration between program parts, the constraints and the variant configuration (for example via feature models) is often done through weak links, and with little awareness of the semantics of the underlying language14. As a consequence, variant management is a

huge source of accidental complexity.

An additional concern is tool integration. The diverse requirements and limitations of C discussed so far often lead to the use of a wide variety of tools in a single development project. Most commercial off-the-shelf (COTS) tools are not open enough to facilitate seamless and semantically meaningful integration with other tools, leading to significant accidental tool integration complexity. COTS tools often also do not support mean-

ingful language extension, severely limiting the ability to define and use custom domain-specific abstractions.

### 20.3 The mbeddr Approach

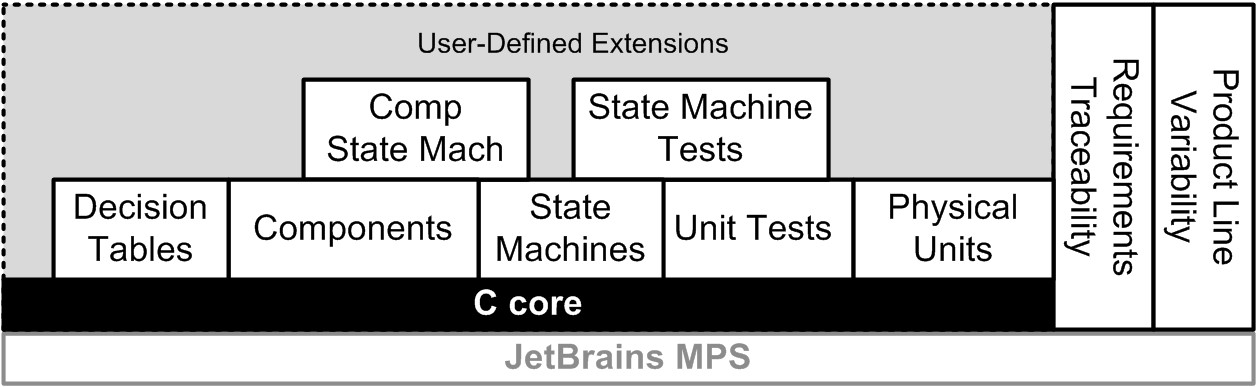
Language engineering provides a holistic approach to solving these challenges. In this section we illustrate how mbeddr addresses the challenges with an extensible version of the C programming language, growing a stack of languages extensions

(see Fig. 20.2, and Section 4.6.2 for a discussion of language extension). The following section explores which ways *Wm* of extending C are necessary to address the challenges *Cn*. Section 20.3.2 then shows examples that address each of the challenges and ways of extending C.

The semantics of an extension are typically defined by a transformation back to the base language. For example, in an extension that provides state machines, these may be transformed to a **switch/case**-based implementation in C. Extensions can be stacked (Fig. 20.2), where a higher-level extension extends (and transforms back to) a lower-level extension instead of C. At the bottom of this stack resides plain C in text form, and a suitable compiler. Fig. 20.1 shows an example in which a module containing a component that contains a state

|  |  |
| --- | --- |
| machine is transformed to C, and then compiled.  As we have seen in Section 16.2, MPS supports *modular* language extension, as well as the use of independently developed language extensions in the same system. For example, in mbeddr a user can include an extension that provides state machines and an extension that provides physical units *in the same program* without first defining a combined language statemachine-with-units. This is very useful, because it addresses real- |  |

world constraints: a set of organizations, such as the departments in a large company, will probably not agree on a *single* set of extensions to C, since they typically work in slightly different areas. Also, a language that contains *all* relevant abstractions would become big and unwieldy. Modular language extension solves these problems.



|  |  |
| --- | --- |
|  |  |
| *20.3.1 Ways of Extending C*  In this section we discuss in which particular ways C needs to |  |
| be extensible to address the challenges discussed above15.  *W*1*: Top-Level Constructs* Top level constructs (on the level of functions or **struct** declarations) are necessary. This enables the integration of test cases or new programming paradigms relevant in particular domains, such as state machines, or interfaces and components.  *W*2*: Statements* New statements, such as **assert** or **fail** |  |
| statements in test cases, must be supported16. Statements may |  |

have to be restricted to a specific context; for example, **assert** or **fail** statements must *only* be used in test cases and not in any other statement list.

*W*3*: Expressions* New kinds of expressions must be supported. An example is a decision table expression that represents a two-level decision tree as a two-dimensional table

(Fig. 20.4).

*W*4*: Types and Literals* New types, e.g., for matrices, complex numbers or quantities with physical units, must be supported. This also requires the definition of new operators, and overriding the typing rules for existing ones. New literals may also be required: for example, physical units could be attached to number literals (as in **10kg**)..

*W*5*: Transformation* Alternative transformations for existing language concepts must be possible. For example, in a module marked as **safe**, the expression **x + y** may have to be translated into an invocation of **addWithBoundsCheck(x, y)**, an **inline** function that performs bounds-checking, besides the addition.

*W*6*: Meta Data Decoration* It should be possible to add meta data, such as trace links to requirements or product line variability constraints, to arbitrary program nodes, without changing the concept of the node.

|  |  |
| --- | --- |
| the use of pointer arithmetic should be prohibited in modules marked as *safe*, or the use of real numbers should be prohibited in state machines that are intended to be model checked (model checkers do not support real numbers).  *20.3.2 Extensions Addressing the Challenges*  In this section we present example extensions that illustrate how we address the challenges discussed in Section 20.2. We |  |
| show at least one example for each challenge18. The table be- |  |
| low shows an overview of the challenges, the examples in this |  |

*W*7*: Restriction* It should be possible to define contexts that restrict the use of specific language concepts17. For example,

section, and the ways of extension each example makes use of19.

|  |  |
| --- | --- |
| **Challenge** | **Example Extensions** |
| *C*1 | State machines (*W*1, *W*2) |
| (Low-Overhead | Components (*W*1) |
| Abstraction) | Decision Tables (*W*3) |
| *C*2 | Cleaned up C (*W*7) |
| (Safer C) | Safe Modules (*W*5, *W*7) |
| *C*3  (Annotations) | Physical Units (*W*4) |
| *C*4 | Unit Tests (*W*1, *W*2) |
| (Static Checks, | State Machines (*W*1, *W*2) |
| Verification) | Safe Modules (*W*2, *W*5, *W*7) |
| *C*5 | Requirements Traceability (*W*6) |
| (Process Support) | Product Line Variability (*W*6) |

*A Cleaned-Up C* (addresses *C*2, uses *W*7) To make C extensible, we first had to implement C in MPS. This entails the

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | definition of the language structure, syntax and type system20. | | |  | | In the process we changed some aspects of C. Some of these changes are the first step in providing a safer C (challenge *C*2). Others changes were implemented, because it is more convenient to the user, or because it simplified the implementation of the language in MPS. Out of eight changes in total, four are for reasons of improved robustness and analyzability, two are for end-user convenience and three are to simplify the implementation in MPS. We discuss some of them below, and the table below shows a summary. | | |  | | **Difference** | **Reason** |  |  |  | | --- | --- | | No Preprocessor | Robustness | | Native Booleans (and a cast operator for legacy interop) | Robustness | | **enum**s are not **int**s (special operators for **next**/**previous** | Robustness | | C99 Integral Types Required | Robustness | | Modules instead of Headers | End-User Convenience | | **hex<..>**, **oct<..>**, **bin<..>** | Simplified | | instead of **0x..** and **0..** | Implementation | | Type annotation on type | Simplified | | (**int[] a** instead of **int a[]**) | Implementation | | Cleaned up syntax for function | End-User Convenience, | | types and function pointers | Simplified  Implementation |   mbeddr C provides *modules* (Fig. 20.3). A module contains the top-level C constructs (such as **struct**s, functions or global variables). These module contents can be exported. Modules can *import* other modules, in which case they can access the exported contents of the imported modules. While header files   |  |  | | --- | --- | | are generated, we do not expose them to the user: modules provide a more convenient means of controlling modularizing programs and limiting which elements are visible globally. mbeddr C does not support the *preprocessor*. Empirical stud-  21 |  |   ies show that it is often used to emulate missing features of C |

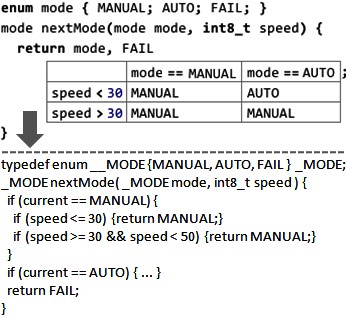
in ad hoc way, leading to problems regarding maintenance and analyzability. Instead, mbeddr C provides first-class support for the most important use cases of the preprocessor. Examples include the modules mentioned above (replacing **#include**), as well as the support for variability discussed below (replacing **#ifdef**s). Instead of defining macros, users can create first-

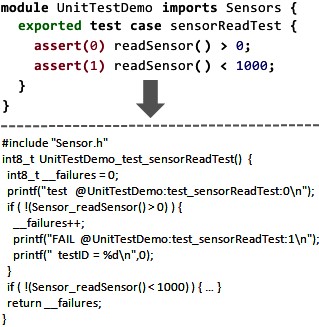
21 M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of c preprocessor use. *IEEE Trans. Softw. Eng.*, 28, December 2002

class language extensions, including type checks and IDE support. Removing the preprocessor and providing specific support for its important use cases goes a long way in creating more maintainable and more analyzable programs. The same is true for introducing a separate **boolean** type and not interpreting integers as Booleans by default (an explicit cast operator is available).

Type decorations, such as array brackets or the pointer asterisk, must be specified on the type, not on the identifier (**int[] a;** instead of **int a[];**). This has been done for reasons of consistency and to simplify the implementation in MPS: it is the property of a type to be an array type or a pointer type, not the property of an identifier. Identifiers are just names.

*Decision Tables* (addressing *C*1, uses *W*3) Decision tables are a new kind of expression, i.e. they can be evaluated. An example is shown in Fig. 20.4. A decision table represents nested **if** statements. It is evaluated to the value of the first cell whose column and row headers are **true** (the evaluation order is left to right, top to bottom). A default value (**FAIL**) is specified to handle the case in which none of the column/row header combinations is **true**. Since the compiler and IDE have to compute a type for expressions, the decision table specifies the type of its result values explicitly (**int8**).

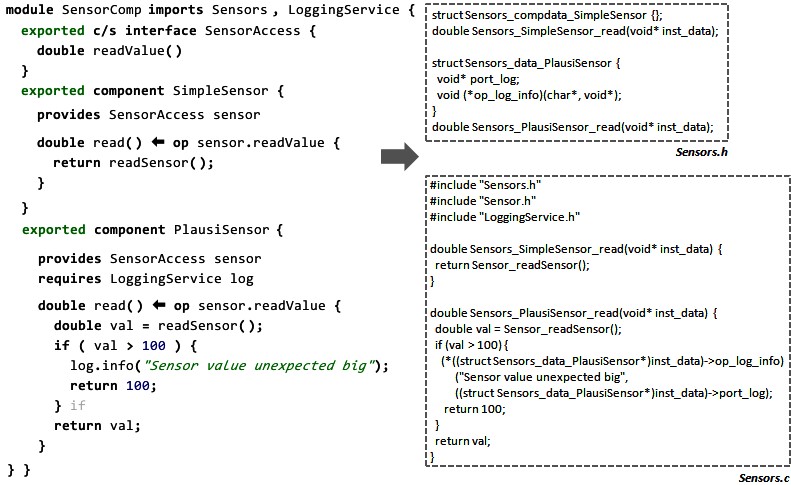




*Unit Tests* (addresses *C*4, uses *W*1, *W*2) Unit tests are new top-level constructs (Fig. 20.5) introduced in a separate *unittest* language that extends the C core. They are like **void** functions without arguments. The *unittest* language also introduces **assert** and **fail** statements, which can only be used inside test cases. Testing embedded software can be a challenge, and the *unittest* extension is an initial step towards providing comprehensive support for testing.

|  |  |
| --- | --- |
|  |  |
| *Components* (addresses *C*1, uses *W*1) are new top-level constructs that support modularization, encapsulation and the separation between specification and implementation (Fig. 20.6). In contrast to modules, a component uses interfaces and ports to declare the contract it obeys. Interfaces define operation signatures and optional pre- and post-conditions (not shown in the example). Provided ports declare the interfaces offered by a component; required ports specify the interfaces a component expects to use. Different components can implement the same |  |

interface differently. Components can be instantiated (also in contrast to modules), and each instance’s required ports have to be connected to compatible provided ports provided by other component instances. Polymorphic invocations (different components "behind" the same interface) are supported.

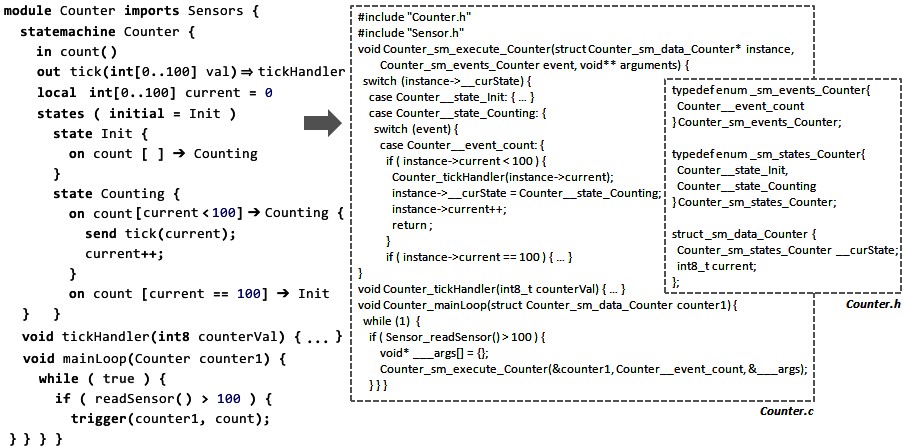


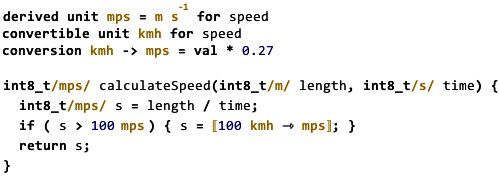
|  |  |
| --- | --- |
|  |  |
| *State Machines* (addresses *C*1, *C*4, uses *W*1, *W*2) State machines provide a new top-level construct (the state machine itself), as well as a **trigger** statement to send events into state machines (see Fig. 20.7). State machines are transformed into a **switch/case**-based implementation in the C program. Entry, exit and transition actions may only access variables defined locally in state machines and fire out events. Out events may optionally be mapped to functions in the surrounding C program, where arbitrary behavior can be implemented. In this way state machines are semantically isolated from the rest of the code, enabling them to be model checked: if a state machine is marked as **verifiable**, we also generate a representation of the state machine in the input language of the NuSMV | . |
| model checker22, including a set of property specifications that | 22 **nusmv.fbk.eu** |

*Physical Units* (addresses *C*3, uses *W*4) Physical units are new types that specify a physical unit in addition to the data type (see Fig. 20.8). New literals support the specification of values for those types that include the physical unit. The typing rules for the existing operators (**+**, **\*** or **>**) are overridden to perform the correct type checks for types with units. The type system also performs unit computations to deal correctly with unit computations (as in **speed = length/time**).

|  |
| --- |
| *Requirements Traces* (addresses *C*5, uses *W*6) Requirements traces are meta data annotations that link a program element to requirements, essentially elements in other models imported from requirements management tools[[1]](#footnote-1). Requirements traces can be attached to any program element without that element’s definition having to be aware of this (see green (gray in print) highlights in Fig. 20.9 and in Fig. 20.22). |

|  |  |
| --- | --- |
| are verified by default. Examples include dead state detection, dead transition detection, non-determinism and variable bounds checks. In addition, users can specify additional highlevel properties based on the well-established catalog of tem- |  |
| poral logic properties patterns23. The state machines extension |  |
| also supports hierarchical states as a further means of decomposing complex behavior. |  |

Figure 20.7: A state machine is embedded in a C module as a top-level construct. It declares **in** and **out events**, as well as local variables, states and transitions. Transitions react to **in event**s, and **out event**s can be fired in actions. Through bindings (e.g., **tickHandler**), state machines interact with C code. State machines can be instantiated. They are transformed to **enum**s for states and events, and a function that executes the state machine using **switch** statements. The **trigger** statement injects events into a state machine instance by calling the state machine function.



*Presence Conditions* (addresses *C*5 and *W*6) A presence condition determines whether the program element to which it is attached is part of a product in the product line25. A prod-

|  |  |
| --- | --- |
| uct is configured by specifying a set of configuration flags (expressed via feature models), and the presence condition specifies a Boolean expression over these configuration switches. Like requirements traces, presence conditions can be attached |  |
| to any program element26. Upon transformation, program el- |  |

.

ements whose presence condition evaluates to **false** for the selected product configuration are simply removed from the program (and hence will not end up in the generated binary). This program customization can also be performed by the editor, effectively supporting variant-specific editing.

*Safe Modules* (addresses *C*2, uses *W*5, *W*7) Safe modules help prevent writing risky code. For example, 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. As another example, safe modules also provide the **safeheap** statement, which automatically frees dynamic variables allocated inside its body (see Fig. 20.13).

#### 20.3.3 Addressing the Tool Integration Challenge

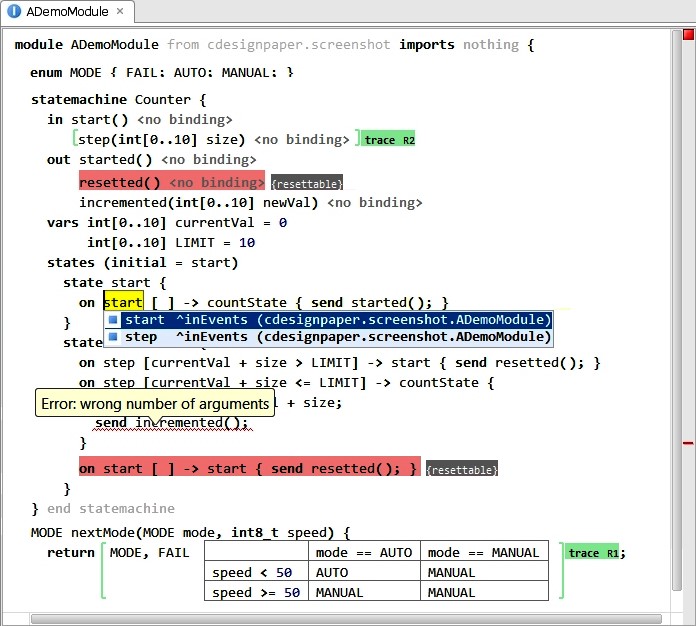
By building all languages (C, its extensions or any other DSLs) on top of MPS, the tool integration challenge is completely solved. All languages get an MPS-style IDE, including syntax highlighting, code completion, static error checking and annotation, quick fixes and refactorings, as well as a debugger

(details see Section 15.2.5). Fig. 20.9 shows a screenshot of the tool, as we edit a module with a decision table, a state machine, requirements traces and presence conditions.

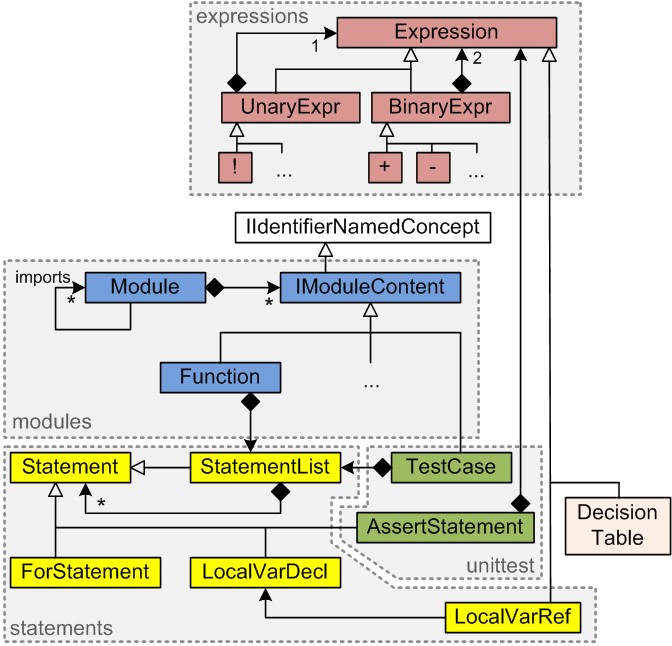
### 20.4 Design and Implementation

|  |  |
| --- | --- |
| previous section27.  *20.4.1 The mbeddr Core Languages*  C can be partitioned into expressions, statements, functions, etc. We have factored these parts into separate language modules to make each of them reusable without pulling in all of | . |
| C. The **expressions** language28 is the most fundamental lan- |  |
| guage. It depends on no other language and defines the primitive types, the corresponding literals and the basic operators. Support for pointers and user defined data types (**enum, struct, union**) is factored into the **pointers** and **udt** languages respec- |  |

This section discusses the implementation of mbeddr language extensions. We briefly discuss the structure of the C core language. The main part of this section discusses each of the ways *Wm* of extending C based on the extensions discussed in the

 attached to an out-event and a transition.

tively. **statements** contains the procedural part of C, and the **modules** language covers modularization. Fig. 20.10 shows an overview of some of the languages and constructs.

.

|  |  |
| --- | --- |
| *20.4.2 Addressing W*1 *(Top-Level Constructs): Test Cases*  In this section we illustrate the implementation of the **test case** construct, as well as of the **assert** and **fail** statements available inside test cases.  *Structure* **Module**s own a collection of **IModuleContent**s, an interface that defines the properties of everything that can reside directly in a module. All top-level constructs such as  **Function**s implement **IModuleContent**. **IModuleContent** extends MPS’ **IIdentifierNamedConcept** interface, which provides a **name** property. **IModuleContent** also defines a Boolean property **exported** that determines whether the respective mod- |  |
| ule content is visible to modules that import this module29. |  |
| Since the **IModuleContent** interface can also be implemented by concepts in other languages, new top-level constructs such as the **TestCase** in the **unittest** language can implement this interface, as long as the respective language has a dependency on the **modules** language, which defines **IModuleContent**. The class diagram in Fig. 20.10 shows some of the relevant concepts and languages. |  |

*Constraints* A test case contains a **StatementList**, so any C statement can be used in a test case. **StatementList** becomes available to the unit test language through its dependency on the **statements** language. **unittest** also defines new statements: **assert** and **fail**. They extend the abstract **Statement** concept defined in the **statements** language. This makes them valid in *any* statement list, for example in a function body. This is undesirable, since the transformation of **assert**s into C depends on them being used in a **TestCase**. To enforce this, a *can be child* constraint is defined (Fig. 20.11).

|  |
| --- |
| **concept constraints** AssertStatement {  **can be child**  (context, scope, parentNode, link, childConcept)->**boolean** { parentNode.ancestor<TestCase>.isNotNull;  }  } |

.

*Transformation* The new language concepts in **unittest** are reduced to C concepts: the **TestCase** is transformed to a **void** function without arguments, and the **assert** statement is transformed into a **report** statement defined in the logging language. The **report** statement, in turn, it is transformed into a platform-specific way of reporting an error (console, serial line or error memory). Fig. 20.12 shows an example of this two-step process.

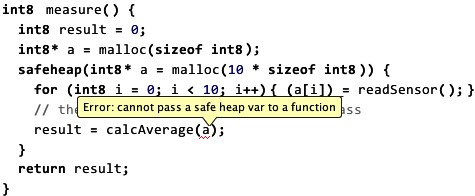
|  |  |  |
| --- | --- | --- |
| **test case** exTest { | **void** test\_exTest { | **void** test\_exTest { |
| **int** x = add(2, 2); | **int** x = add(2, 2); | **int** x = add(2, 2); |
| **assert**(0) x == 4; | **report** | **if** (!(x == 4)) { |
| } | **test**.FAIL(0) | printf("fail:0"); |
|  | **on** !(x == 4); | } |
|  | } | } |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| *20.4.3* | *Addressing W*2 *(Statements): Safeheap Statement* |  |

We have seen the basics of integrating new statements in the previous section where **assert** and **fail** extended the **Statement** concept inherited from the C core languages. In this section we focus on statements that need to handle local variable scopes and visibilities. We implement the **safeheap** statement mentioned earlier (see Fig. 20.13), which automatically frees dynamically allocated memory. The variables introduced by the **safeheap** statement must only be visible inside its body, and have to shadow variables of the same name declared in outer scopes (such as the **a** declared in the second line of the **measure** function in Fig. 20.13).

*Structure* The **safeheap** statement extends **Statement**. It contains a **StatementList** as its body, as well as a list of **SafeHeapVar**s. These extend **LocalVarDecl**, so they fit with the existing mechanism for handling variable shadowing (explained below).

.



*Behavior* **LocalVarRef**s are expressions that reference a **LocalVarDecl**. A scope constraint determines the set of visible variables for a given **LocalVarRef**. We implement this constraint by plugging into mbeddr’s generic local variable scoping mechanism using the following approach. The constraint ascends the containment tree until it finds a node which implements **ILocalVarScopeProvider**, and calls its **getLocalVarScope** method. A **LocalVarScope** has a reference to an outer scope, which is set by finding *its* **ILocalVarScopeProvider** ancestor, effectively building a hierarchy of **LocalVarScope**s. To get at the list of the visible variables, the **LocalVarRef** scope constraint calls the **getVisibleLocalVars** method on the innermost **LocalVarScope** object. This method returns a flat list of **LocalVarDecl**s, taking into account that variables owned by 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 the two methods shown in Fig. 20.14.

*Type System* To make the **safeheap** statement work correctly, we have to ensure that the variables declared and allocated in a **safeheap** statement do not escape from its scope. To

|  |
| --- |
| **public** LocalVarScope getLocalVarScope(node<> ctx, **int** stmtIdx) {  LocalVarScope scope = **new** LocalVarScope(getContainedLocalVariables()); node<ILocalVarScopeProvider> outer = **this**.ancestor<ILocalVarScopeProvider>;  **if** (outer != **null**) {  scope.setOuterScope(outer.getLocalVarScope(**this**, **this**.index));  } **return** scope;  } **public** sequence<node<LocalVariableDecl>> getContainedLocalVars() { **this**.vars;  } |

prevent this, an error is reported if a reference to a **safeheap** variable is passed to a function. Fig. 20.15 shows the code.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **checking rule** check\_safeVarRef **for concept** = LocalVarRef **as** lvr { **boolean** isInSafeHeap = lvr.ancestor<SafeHeapStatement>.isNotNull;  **boolean** isInFunctionCall = lvr.ancestor<FunctionCall>.isNotNull;  **boolean** referencesSafeHeapVar = lvr.var.parent.isInstanceOf(SafeHeapStatement);  **if** (isInSafeHeap && isInFunctionCall && referencesSafeHeapVar) **error** "cannot pass a safe heap var **to** a function" -> lvr; } |  |  |  | | --- | --- | | *20.4.4 Addressing W*3 *(Expressions): Decision Tables*  Fig. 20.4 showed the decision table expression. It is evaluated to the expression in a cell *c* if the column header of *c* and the |  | | row header of *c* are true30. If none of the condition pairs is |  | | true, then the default value, **FAIL** in the example, is used as the resulting value. A decision table also specifies the type of the value it will evaluate to, and all the expressions in content cells have to be compatible with that type. The type of the header cells has to be Boolean.  *Structure* The decision table extends the **Expression** concept defined in the **expressions** language. Decision tables contain a list of expressions for the column headers, one for the row headers and another for the result values. It also contains a child of type **Type**, to declare the type of the result expressions, as well as a default value expression. The concept defines an alias **dectab** to allow users to instantiate a decision table in |  | | the editor31. |  | |

.

*Editor* Defining a tabular editor is straightforward: the editor definition contains a **table** cell, which delegates to a Java class that implements **ITableModel**. This is similar to

the approach used by Java Swing. It provides methods such as **getValueAt( int row, int col)** or **deleteRow(int row)**, which have to be implemented for any specific table-based editor. To embed another node in a table cell (such as the expression in the decision table), the implementation of **getValueAt** simply returns the node (whose editor is then embedded in the table’s editor).

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

*Type System* MPS uses unification in the type system. Language concepts specify type equations that contain type literals (such as **boolean**) as well as type variables (such as **typeof (dectab)**). The unification engine then tries to assign values to the type variables such that all applicable type equations become true. New language concepts contribute additional type equations. Fig. 20.16 shows those for decision tables32.

|  |  |
| --- | --- |
| *20.4.5 Addressing W*4 *(Types and Literals): Physical Units*  We use physical units to illustrate the addition of new types and literals. We have already shown example code earlier in  Fig. 20.8.  *Structure* Derived and convertible **UnitDeclaration**s are |  |
| **IModuleContents**. Derived unit declarations specify a name (**mps**, **kmh**) and the corresponding SI base units (**m**, **s**), plus an exponent; a convertible unit declaration specifies a name and a conversion formula33. The backbone of the extension | 33 The unit extension does not automatically support prefixes like **k**, **M** or **m**. If you need **km** or **mm** you have to define this as a convertible unit with the respective conversion formulae. This is a conscious decision driven by |

is the **UnitType**, which is a composite type that has another type (**int**, **float**) in its **valueType** slot, plus a unit (either

.

an SI base unit or a reference to a **UnitDeclaration**). It is represented in programs as **baseType/unit/**. We also provide **LiteralWithUnit**s, which are expressions that contain a **valueLiteral** and, like the **UnitType**, a unit (so we can write, for example, **100 kmh**).

*Scoping* **LiteralWithUnit**s and **UnitType**s refer to a **UnitDeclaration**, which is a module content. According to the visibility rules, valid targets for the reference are the **UnitDeclaration**s in the same module, and the *exported* ones in all imported modules. This rule applies to *any* reference to *any* module content, and is implemented generically. Fig. 20.17 shows the code for the scope of the reference to the **UnitDeclaration**. We use an interface **IVisibleNodeProvider**, (implemented by **Module**s) to find all instances of a given type. The implementation of **visibleContentsOfType** searches through the contents of the current and imported modules and collects instances of the specified concept. The result is used as the scope for the reference.

|  |
| --- |
| **link** {unit} **search scope**:  (model, refNode, enclosingNode, operationContext)  ->sequence<node<UnitDeclaration>> {  enclosingNode.ancestor<IVisibleNodeProvider>. visibleContentsOfType(**concept**/UnitDeclaration/);  } |

.

*Type System* We have seen how MPS uses equations and unification to specify type system rules. However, there is special support for binary operators that makes overloading for new types easy: overloaded operations containers essentially specify 3-tuples of *(leftArgType, rightArgType, resultType)*, plus applicability conditions to match type patterns and decide on the resulting type. Typing rules for new (combinations of) types can be added by specifying additional 3-tuples.

Fig. 20.18 shows the overloaded rules for C’s **MultiExpression** (the language concept that implements the multiplication operator **\***) when applied to two **UnitType**s: the result type will be a **UnitType** as well, where the exponents of the SI units are added.

While any two units can legally be used with **\*** and **/** (as long as we compute the resulting unit exponents correctly), this is not true for **+** and -. There, the two operand types must

|  |
| --- |
| **operation concepts**: MultiExpression **left operand type**: **new** node<UnitType>() **right operand type**: **new** node<UnitType>()  **is applicable**:  (op, leftOpType, rightOpType)->**boolean** { node<> resultingValueType = **operation type**(op, leftOpType.valueType , rightOpType.valueType );  **return** resultingValueType != **null**;  } **operation type**:  (op, leftOpType, rightOpType)->node<> { node<> resultingValueType = **operation type**(op, leftOpType.valueType, rightOpType.valueType );  UnitType.create(resultingValueType, leftOpType.unit.toSIBase().add( rightOpType.unit.toSIBase(), 1 )  );  } |

.

|  |  |
| --- | --- |
| be the same in terms of their representation in SI base units. We express this by using the following expression in the **is** |  |
| **applicable** section34: |  |

leftOpType.unit.isSameAs(rightOpType.unit)

|  |  |
| --- | --- |
| In the **operation type** section we then compute the resulting unit type by adding the exponents of the components of the two unit types.  The typing rule for the **LocalVariableDeclaration** requires that the type of the **init** expression must be the same or a subtype of the **type** of the variable. To make this work correctly, we have to define a type hierarchy for **UnitType**s. We achieve this by defining the supertypes for each **UnitType**: the supertypes are those **UnitType**s whose unit is the same, and whose **valueType** is a supertype of the current **UnitType**’s value type.  Fig. 20.19 shows the rule. | . |

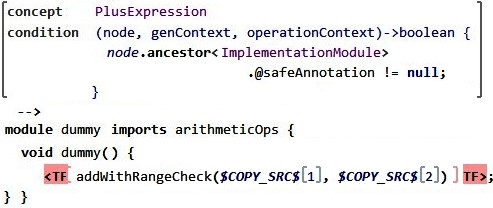
|  |
| --- |
| **subtyping rule** supertypeOf\_UnitType **for concept** = UnitType **as** ut {  nlist<> res = **new** nlist<>;  **foreach** st **in** immediateSupertypes(ut.valueType) { res.add(UnitType.create(st, ut.unit.copy));  }  **return** res;  } |

#### 20.4.6 Addressing W5 (Alternative Transformations): Range Checking

The **safemodules** language defines an *annotation* to mark **Modules** as safe (we will discuss annotations in the next subsection). If a module is safe, the binary operators such as **+** or **\*** are replaced with calls to functions that, in addition to performing the addition or multiplication, perform a range check.

*Transformation* The transformation that replaces the binary operators with function calls is triggered by the presence of this annotation on the **Module** which contains the operator.

Fig. 20.20 shows the code. The **@safeAnnotation != null** checks for the presence of the annotation.



MPS uses priorities to specify relative orderings of transfor-

mations, and MPS then calculates a global transformation order for any given model. We use a priority to express the fact that this transformation runs *before* the final transformation that maps the C tree to C text for compilation.

#### 20.4.7 Addressing W6 (Meta Data): Requirements Traces

Annotations are concepts whose instances can be added as children to a node *N* without this being specified in the definition of *N*’s concept35. While structurally the annotations are children of the annotated node, the editor is defined the other way round: the annotation editor delegates to the editor of the annotated element. This allows the annotation editor to add additional syntax *around* the annotated element36.

We illustrate the annotation mechanism based on the requirements traces. As we discussed at the end of Section 20.3.2, a requirements trace establishes a link from a program element to a requirement. It is important that this annotation can be annotated to **any** node, independent of the concept of which it is an instance. As a consequence of the projectional approach, the program can be shown with or without the annotations, controlled by a global switch. Fig. 17.5 had shown an example.

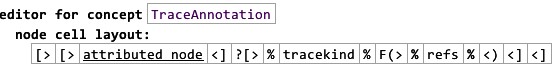
*Structure* Fig. 20.21 shows the structure. Notice how it extends the MPS-predefined concept **NodeAnnotation**. It also specifies a **role**, which is the name of the property that is used to store **TraceAnnotation**s under the annotated node.

.

|  |
| --- |
| **concept** TraceAnnotation **extends** NodeAnnotation **implements** <none> **children**:  TraceKind tracekind 1  TraceTargetRef refs 0..n **concept properties**:  role = trace **concept** links:  annotated = BaseConcept |

*Editor* In the editor annotations look as if they *surrounded*

their parent node (although they are in fact children). Fig. 20.22 shows the definition of the editor of the requirements trace annotation (an example is shown in Fig. 20.9): it puts the trace to the right of the annotated node. Since MPS is a projectional editor, there is base-language grammar that needs to be made aware of the additional syntax in the program. This is key to enabling arbitrary annotations on arbitrary program nodes.



|  |  |
| --- | --- |
| Annotations are typically attached to a program node via an intention. Intentions are an MPS editor mechanism: a user selects the target element, presses **Alt-Enter** and selects **Add Trace** from the popup menu. Fig. 20.23 shows the code for the intention that attaches a requirements trace. | . |

|  |
| --- |
| **intention** addTrace **for** BaseConcept { description(node)->**string** { "Add Trace";  }  isApplicable(node)->**boolean** { node.@trace == **null**;  }  execute(editorContext, node)->**void** { node.@trace = **new** node<TraceAnnotation>(); }  } |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| *20.4.8* | *Addressing W*7 *(Restriction): Preventing Use of Real*  *Numbers* | . |

We have already seen in Section 20.4.2 how constraints can prevent the use of specific concepts in certain contexts. We use the same approach for preventing the use of real number types inside model-checkable state machines: a **can be ancestor** constraint in the state machine prevents instances of **float** in the state machine if the **verifiable** flag is set[[2]](#footnote-2).

### 20.5 Experiences

In Section 20.5.1 we provide a brief overview of our experiences in implementing mbeddr, including the size of the project and the efforts spent. Section 20.5.2 discusses to what degree this approach leads to improvements in embedded software development.

|  |  |  |
| --- | --- | --- |
| **Element** | **Count** | **LOC-Factor** |

|  |  |  |
| --- | --- | --- |
| Language Concepts | 260 | 3 |
| Property Declarations | 47 | 1 |
| Link Declarations | 156 | 1 |
| Editor Cells | 841 | 0.25 |
| Reference Constraints | 21 | 2 |
| Property Constraints | 26 | 2 |
| Behavior Methods | 299 | 1 |
| Type System Rules | 148 | 1 |
| Generation Rules | 57 | 10 |
| Statements | 4,919 | 1.2 |
| Intentions | 47 | 3 |
| Text Generators | 103 | 2 |
| **Total LOC** |  | **8,640** |

.

#### 20.5.1 Language Extension

*Size* Typically, lines of code are used to describe the size of a software system. In MPS, a "line" is not necessarily meaningful. Instead we count important elements of the implementation and then estimate a corresponding number of lines of code. Fig. 20.24 shows the respective numbers for the core, i.e. C itself plus unit test support, decision tables and build/make integration (the table also shows how many LOC equivalents we assume for each language definition element, and the caption explains to some extent the rationale for these factors). According to our metric the C core is implemented with less than 10,000 lines of code.

Let us look at an incremental extension of C. The components extension (interfaces, components, pre- and post-conditions, support for mock components in testing and a generator back to plain C) is circa 3,000 LOC equivalents. The state machines extension is circa 1,000. Considering the fact that these LOC equivalents represent the language definition (including type systems and generators) and the IDE (including code completion, syntax coloring, some quick fixes and refactorings), this clearly demonstrates the efficiency of MPS for language development and extension.

|  |  |
| --- | --- |
| *Effort* In terms of effort, the core C implementation has been circa 4 person months divided between three people. This results in roughly 2,500 lines of code per person month. Extrapolated to a year, this would be 7,500 lines of code per de- |  |
| veloper. According to McConnell38, in a project up to 10,000 |  |
| LOC, a developer can typically do between 2,000 and 25,000 LOC. The fact that we are at the low end of this range can be explained by the fact that MPS provides very expressive languages for DSL development: you don’t have to write a lot of code to express a lot about a DSL. Instead, MPS code is relatively dense and requires quite a bit of thought. Pair programming is very valuable in language development.  Once a developer has mastered the learning curve, language extension can be very productive. The state machines and components extension have both been developed in about a month. The unit testing extension or the support for decision tables can be implemented in a few days. |  |

*Language Modularity, Reuse and Growth* Modularity and composition are central to mbeddr. Building a language extension should not require changes to the base languages. This requires that the extended languages are built with extension in mind. Just as in object-oriented programming, where only complete methods can be overridden, only specific parts of a language definition can be extended or overwritten. The implementation of the default extensions served as a test case to confirm that the C core language is in fact extensible. We found a few problems, especially in the type system, and fixed them. None of these fixes were "hacks" to enable a specific extension – they were all genuine mistakes in the design of the C core. Due to the broad spectrum covered by our extensions, we are confident that the current core language provides a high degree of extensibility.

|  |  |
| --- | --- |
| Modularity should also support reuse in contexts not anticipated during the design of a language module. Just as in the case of language extension (discussed above), the languages to be reused have to be written in a suitable way so that the right parts can be reused separately. We have shown this with the state machines language. State machines can be used as toplevel concepts in modules (binding out-events to C functions), and also inside components (binding out-events to component methods). Parts of the transformation of a state machine have to be different in these two cases, and these differences were successfully isolated to make them exchangeable. Also, we reuse the C expression language inside the guard conditions in a state machine’s transitions. We use constraints to prevent the use of those C expression that are not allowed inside transitions (for example, references to global variables). Finally, we have successfully used physical units in components and interfaces.  Summing up, these facilities allow different user groups to |  |
| develop independent extensions, growing40 the mbeddr stack |  |
| even closer towards their particular domain.  *Who can create Extensions?* mbeddr is built to be extended. The question is by whom. This question can be addressed in two ways: who is *able* to extend it from a skills perspective, and who *should* extend it?  Let us address the *skills* question first. We find that it takes about a month for a developer with solid object-oriented pro- |  |
| gramming experience to become proficient with MPS and the structures of the mbeddr core languages41. Also, *designing* |  |
| good languages, independent of their implementation, is a skill |  |
| that requires practice and experience42. So, from this perspec- |  |
| tive we assume that in any given organization there should be a select group of language developers who build the extensions for the end users. Notice that such an organizational structure is common today for frameworks and other reusable artifacts.  There is also the question of who *should* create extensions. |  |

Independently developed extensions should not interact with each other in unexpected ways. While MPS provides no automated way of ensuring this, we have not seen such interactions so far. The following steps can be taken to minimize the risk of unexpected interactions. Generated names should be qualified to make sure that no symbol name clashes occur in the generated C code. An extension should never consume "scarce resources": for example, it is a bad idea for a new **Statement** to require a particular return type of the containing function, or change that return type during transformation. Two such badly designed statements cannot be used together, because they are likely to require *different* return types39.

|  |  |
| --- | --- |
|  |  |
| *20.5.2 Improvements in Embedded Development*  In this section we discuss preliminary results of a real-world development project. The project develops the software for a smart meter system44. A smart meter is an electrical meter that continuously records the consumption of electric power in a home and sends the data back to the utility for monitoring and billing. The particular software we develop will run on a 2-chip board (TI MSP-430 for metrology, another TI processor (tbd.) for the remaining application logic). Instead of the **gcc** compiler used in mbeddr by default, this project uses an IAR compiler.  The software comprises circa 30,000 lines of mbeddr code, has several time-sensitive parts that require a low-overhead implementation, and will have to be certified by an independent body. The software is derived from an existing example smart meter meter system written in traditional C, and reuses exist- |  |
| ing artifacts such as header files and libraries45. |  |
| *Why mbeddr?* mbeddr was chosen to implement the smart meter for the following reasons. The project has to work with an existing code base which had no production-level quality. The code quality needed to be improved *incrementally*. So starting with the existing C code and then refactoring towards bet- |  |
| ter abstractions seemed like a good idea. Also, as we will see below, the existing C extensions provided by mbeddr are a good match for what is needed in the smart meter (see below)46. Finally, as the goal is to have the meter certified, test- |  |

One could argue that, as language development becomes simpler, an uncontrolled growth in languages could occur, ultimately resulting in chaos. This concern should be addressed with governance structures that guide the development of languages. The bigger the organization is, the more important such governance becomes43.

ing the software is very important. By using the abstraction mechanisms provided by mbeddr, and by exploiting the ability to build custom extensions, testability can be improved significantly. In particular, hardware-specifics can be isolated, which enables testing without the actual target hardware. Also, mbeddr’s support for requirements traceability comes in handy for the upcoming certification.

*Using the Existing Extensions* The smart meter uses the following default extensions:

*Components* The smart meter uses components to improve the structure of the application, and to support different implementations of the same interface. This improves modularity and testability. Mocks components are used excessively for testing.

*State Machines* The smart meter communicates with its environment via several different protocols. So far, one of these protocols has been refactored to use a state machine. This has proven to be much more readable than the original C code. Components and state machines are combined, which allows decoupling message assembly and parsing from the application logic in the server component.

*Units* A major part of the smart meter application logic performs computations on physical quantities (time [**s**], current [**A**] or voltage [**V**]). So mbeddr’s support for physical units comes in handy. The benefits of these extensions are mostly in type checking, using types with units also improves the readability and comprehensibility of the code.

*Requirements Tracing* The smart meter also makes use of requirements traces. During the upcoming certification process, these will be extremely useful for tracking if and how the customer requirements have been implemented. Because of their orthogonal nature, the traces can be attached to the new language concepts specifically developed for the smart meter.

*Custom Extensions* As part of the smart meter, so far mbeddr has been extended in the following ways:

*Registers* The smart meter software makes extensive use of registers (metrology: access the sensor values, UART: send and receive data). This cannot be abstracted away easily due to performance/overhead constraints. In addition, some registers are special-purpose registers: when a value is written to such a register, a hardware-implemented computation is automatically triggered based on the value supplied by the programmer. The result of this computation is then stored in the register. To run code that works with these registers on the PC for testing, developers face two problems:

first, the header files that define the addresses of the registers are not valid for the PC’s processor. Second, there are no special-purpose registers on the PC, so no automatic computations or other hardware-triggered actions would be triggered. This problem was solved with a language extension that supports registers as first-class citizens and supports accessing them from mbeddr-C code (see code below).

|  |
| --- |
| **exported** register8 ADC10CTL0 **compute as val** \* 1000  **void** calculateAndStore( int8 value ) {  int8 result = // some calculation with value  ADC10CTL0 = result; // actually stores result \* 1000 } |

The extension also supports specifying an expression that performs the computation. When the code is translated for the real device, the real headers are included, and access to the registers is replaced with access to the constants defined in the header. In testing, **struct**s are generated to hold the register data. Each write access to a register is replaced with a write access to the struct, and the expression that simulates the special purpose register is included in that assignment.

*Interrupts* Many aspects of the smart meter system are driven by interrupts. To integrate the component-based architecture used in the smart meter with interrupts, it is necessary to trigger component runnables (methods) via an interrupt. To this end, we have implemented a language extension that allows us to declare interrupts. In addition, the extension provides runnable triggers that express the fact that a runnable is triggered by an interrupt47. The extension

|  |  |
| --- | --- |
| during component instantiation. A check makes sure that each interrupt-triggered runnable has at least one interrupt assigned[[3]](#footnote-3).  *Data Encoding* As part of the communication protocol, data has |  |
| to be encoded into messages49. A language extension sup- |  |
| ports the definition of data structures, and generated code deals with constructing messages. In particular, the generated code deals with packed data (where data has sizes that are not multiples of 8 bits). Also, code that processes messages can be statically checked against the message structure definition, making this code much more robust (in particular if the message definitions are changed). | ASN.1. |

also provides a concept to assign an interrupt to a runnable

*Conclusions* The mbeddr default extensions have proven extremely useful in the development of the smart meter. The fact that the extensions are directly integrated into C (as opposed to the classical approach of using external DSLs or separate modeling tools) reduces the hurdle of using higher-level extensions and removes any potential mismatch between DSL code and C code.

|  |  |
| --- | --- |
| Additional effort is required to integrate with existing legacy code. As a consequence of the projectional editor, we have to parse the C text (with an existing parser) and construct the MPS AST. mbeddr provides an importer for header files as a means of connecting to existing libraries. However, mostly as a consequence of C’s preprocessor, which allows all kinds of mischief to be done to otherwise well-structured C code, this importer is not trivial. For example, we currently cannot import all alternatives expressed by **#ifdef**s. Users have to specify a specific configuration to be imported (in the future, we will support importing of all options by mapping the **#ifdef**s to mbeddr’s product line variability mechanism). Also, header files often contain platform-specific keywords or macros. Since they are not supported by the mbeddr C implementation, these have to be removed before they can be imported. The header importer provides a regular expression-based facility to remove these platform specifics before the import. The smart meter project, which is heavily based on an existing code base, also drove the |  |
| need for a complete source code importer (including **.c** files, not just header files), which we are currently in the process of developing51. |  |
| We have performed scalability tests and found that mbeddr scales to at least the equivalent of 100,000 lines of C code in |  |

Generating code from higher-level abstractions may introduce performance and resource consumption overhead. While we have not yet performed a systematic analysis of the overhead incurred by the mbeddr extensions, it is low enough to run the smart meter system on the hardware intended for it50.

the developed system. These tests were based on automatically generated sample code and measured editor responsiveness and transformation times. While there are certainly systems that are substantially larger, a significant share of embedded software is below this limit and can be addressed with mbeddr52.

### 20.6 Discussion

*Why MPS?* Our choice of MPS is due to its support for all aspects of language development (structure, syntax, type systems, IDE, transformations), its support for flexible syntax as a consequence of projectional editing, and its support for advanced modularization and composition of languages. The ability to attach annotations to arbitrary program elements without a change to that element’s definition is another strong advantage of MPS (we we use this for presence conditions and trace links, for example).

|  |  |
| --- | --- |
|  |  |
| *Projectional Editing* Projectional editing is often considered a drawback, because the editors feel somewhat different and the programs are not stored as text, but as a tree (XML). We have already highlighted the fact that MPS does a good job regarding the editor experience, and we feel that the advantages of projectional editors regarding syntactic freedom far outweigh the drawback of requiring some initial familiarization. Our experience so far with about ten users (pilot users from industry, students) shows that after a short guided introduction of about 30 minutes, and an initial accommodation period (circa 1-2 days), users can work productively with the projectional editor. Regarding storage, the situation is not any worse than with current modeling tools that store models in a non-textual format, and MPS does provide good support for diff and merge using the projected syntax. |  |
| *Feasibility of Language Extension* Based on the experience with the smart meter, the effort for building extensions is reasonable. For example, the implementation of the language extensions for registers (and the simulation for testing) was done in half a day. The addition of interrupts, interrupt-trig-gered runnables and the way to "wire them" up was circa one day54.  Building a language extension should not require changes to the base language. The extensions for the smart meter demonstrate this point. The registers extension discussed above has been built without changing the underlying C language55. Sim- |  |
| ilarly, the interrupt-based runnable triggers have been hooked | ten). |

While the learning curve for MPS is significant (a developer who wants to become proficient in MPS language development has to invest at least a month), we found that it scales extremely well for larger and more sophisticated languages53.

into the generic trigger facility that is part of the components language. Once a language is designed in a reasonable way, the language (or parts of it) should be reusable in contexts that have not been specifically anticipated in advance. The smart meter system contains such examples: expressions have been embedded in the register definition concept for emulating the hardware behavior, and types with units have been used in decision tables. Again, no change to the existing languages has been necessary.

One criticism that has been used against language extension is that the language will grow large and that it is hard for users to learn all its constructs. In our experience, this is not a problem in mbeddr for the following three reasons: first, the extensions provide linguistic abstractions for concepts that are well known to the users: state-based behavior, interfaces and components or test cases. Second, the additional language features are easily discoverable because of the IDE support. Third, and most important, these extensions are modularized, and any particular end user will only use those extensions that are relevant to whatever their current program addresses. This avoids overwhelming the user with too much "stuff" at a time.

1. [↑](#footnote-ref-1)
2. [↑](#footnote-ref-2)
3. [↑](#footnote-ref-3)