mbeddr - Extensible languages for embedded software development

Tamás Szabó[1], Markus Voelter[2], Bernd Kolb[1], Daniel Ratiu[4], and Bernhard Schaetz[3]

[1]itemis AG, {tamas.szabo | bernd.kolb}@itemis.de

[2]independent/itemis AG, voelter@acm.org

[3]Fortiss, schaetz@fortiss.de

[4]Siemens AG, daniel.ratiu@siemens.com

ABSTRACT

In this industrial presentation we will demonstrate mbeddr, an extensible set of integrated languages for embedded software development. After discussing the context of the talk, we will give details about the mbeddr architecture, which relies on the MPS language workbench. Then we will elaborate on the extension modules and show how they fit with safety-critical development processes. Finally we will point out how the existing languages can be extended by the user by giving some real-world examples, including a language construct that could have prevented the Apple "goto fail" bug as well as mathematical notations.

Keywords Domain Specific Languages and Tooling, Embedded Systems, Language Workbenches, Synthesis of Tailored Tools

1. CONTEXT

Todays' embedded systems are highly diverse, often very complex and many domains are safety-critical, where hardware or software failures may cost lives or a lot of money. An adequate language and tool can ease the development of such systems in many ways; it can ensure the well-formedness of the content and increase the productivity of the developer through the automation of many tedious and repetitive (thus error-prone) tasks, while it can also help in the verification of the critical properties of the system.

The C programming language is often used for the development of low-level control algorithms. On the plus side, the developer has precise control over memory management and it can be used to generate efficient binaries out of the source code. On the other hand, safety-critical domains often require the introduction of custom abstractions (e.g. for verification or requirements tracing), which can be problematic in C. Managing and extending large codebases without proper modularization and separation of specification from implementation can also increase the accidental complexity of the development processes.

HILT 2014, October 18–21, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3217-0/14/10 ...$15.00.
http://dx.doi.org/10.1145/2663171.2663186 .

2. mbeddr OVERVIEW

The mbeddr project [8] is an industry-strength IDE for the C programming language, which aims to ease the design and development of embedded software. Its core idea is to use language engineering [7] to introduce additional language constructs into C. Extensions are modular, and additional language constructs can be added at any time, and also by the end user. This is a fundamental shift in the design of the tools because custom extensions can be created with minimal effort and without the need to invasively change the already existing languages. mbeddr relies on the MPS language workbench [3] to enable language and IDE extensibility and it exists as an open-source project [4]. Figure 1 shows the architecture of mbeddr. The figure also identifies the three concerns addressed by mbeddr: the implementation concern addresses the actual implementation of software, relying on existing C compilers for compilation. The analysis concern integrates existing verification tools to statically verify properties of C code. Finally, the process concern addresses integration into the development process.

The tight integration with external verification tools [6] also allows non-experts to benefit from the capabilities of the tools because (i) the input for the verification tool is automatically generated out of the edited contents and (ii) the results of the analysis is automatically lifted back to the abstraction level of the IDE (tracing information is also generated for the input of the analysis tools).This is a great benefit, because many people shy away from these tools due to the high gap between the abstraction levels of the used tools and because of the complexity of the interpretation of the results.

3. EXTENSIONS FOR HIGH-INTEGRITY LANGUAGES

mbeddr introduces many higher level abstractions to provide a more robust version of the C language, which makes it a good fit for the development of safety-critical systems:

Separation of specification from implementation Interfaces represent the specification and are essentially a set

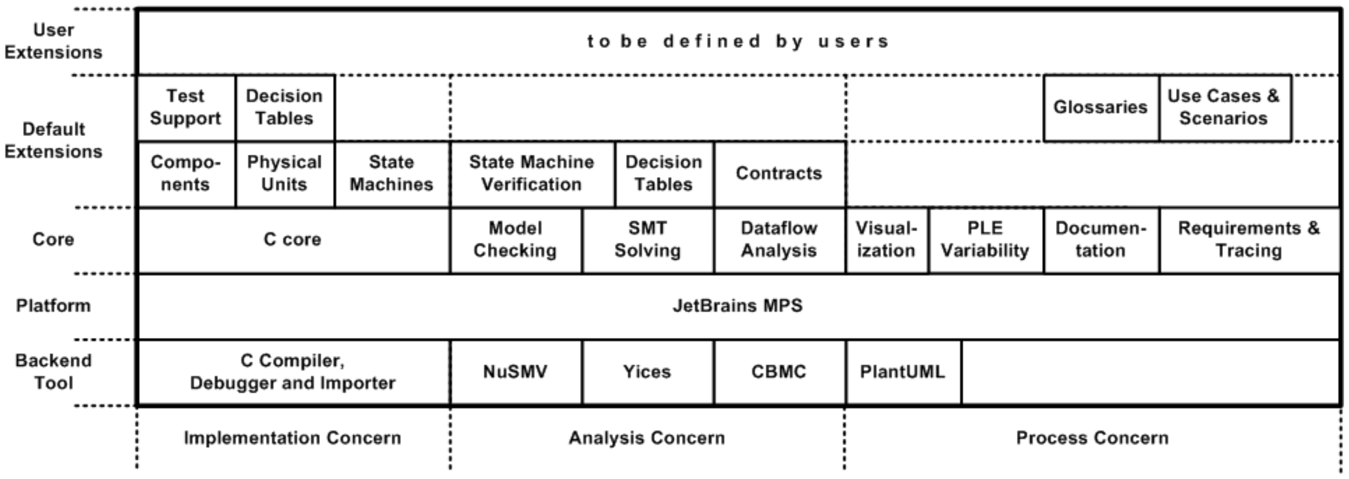


Figure 1: Overview of the mbeddr architecture

of operation signatures. The implementation of an interface is defined in a component, where the component can require other interfaces through ports. It then provides the implementation of the specification by making use of the required ports. Polymorphic behavior is also introduced with this approach, because a required port only specifies the interface, not the implementing component. This construct increases reusability and the developer can create more loosely-coupled software than in the standard version of C.

Testing and Verification The components mentioned above also improve testability because of the improved modularization. An extension supports the definition of mock components, further increasing the fidelity of tests. Interfaces also specify contracts (pre- and postconditions, protocol state machines) which can be verified statically or at runtime.

Physical Units mbeddr makes it possible to use physical units on types or literals (e.g. time `[s]`, current `[A]` or voltage `[V]`). The type system has been extended with unit checks and computations (for example, adding `V` and `A` results in a type error and multiplying `V` and `A` results in `W`). It is also possible to define meta units for functions, which will be bound based on the units on the input parameters of a function call. The meta units that are used inside the function body in a return statement will then be checked with the specific binding. One can also create conversion rules between various unit definitions and invoke these conversions in the C code (conversion rules will be converted into macros in the generated C code). Beyond the benefits for type checking, this extension makes the code more readable and comprehensible.

Decision tables MPS supports notational flexibility within the IDE. This allows, for example, to insert a decision table inside the edited source code. Compared to a textual notation using nested `if` statements, decision tables are easier to edit and read. Decision tables can also be automatically verified for completeness and determinism directly from the IDE.

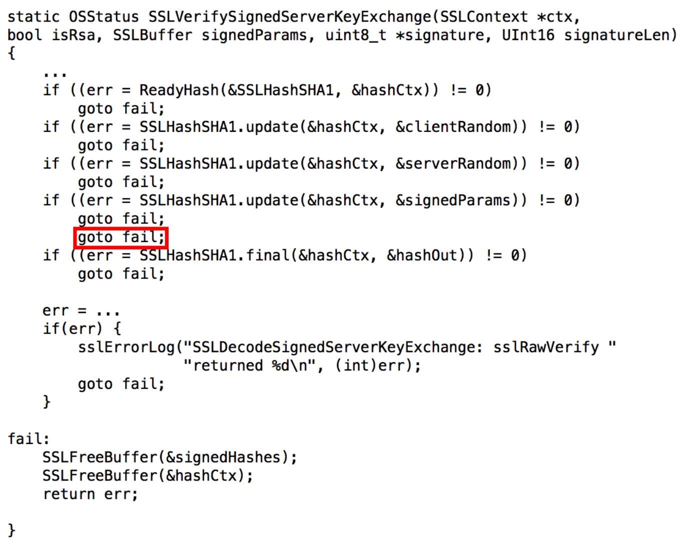
State machines Designing and implementing complex protocols can be cubersome in plain source code. Apart from the readability issues the verifiability of the algorithm is also an important concern. mbeddr helps here with the introduction of state machines. Interesting properties of the designed state machine can then be verified through model checking; counterexamples are lifted back to the abstraction level of the state machine if the verification finds an error.

Requirements tracing Development processes and certification standards emphasize the traceability between the implementation and the corresponding requirements. Many interesting impact analyses can be carried out if proper tooling is provided for the maintenance and update of traces. mbeddr makes it possible to easily specify requirements in the IDE and then link these requirements to corresponding implementation artifacts.

All of the extensions mentioned above are shipped with mbeddr and can be extended or enhanced even further to fully meet the requirements of the specific domain. In the end all of these constructs will be transformed to C99 source code, possibly in several steps. For example state machines embedded in components are first transformed to regular component implementations, those are then transformed to C, which is finally transformed to text for compilation.

4. CUSTOM EXTENSIONS

Apple has recently become "famous" for the *goto fail* bug, which was about a security flaw during the authentication of SSL certificates. A snippet of the corresponding C source code is shown on Figure 2 and the whole file can be found at [5]. The code tries to validate an SSL certificate using the multi-step error checking idiom where different kinds of validations are performed in a sequential order and the first failure will result in a jump to the error-handling routine (marked with the `fail` label here). As a consequence of the additional `goto` statement (marked in red) the program will immediately jump to the error-handling code instead of following the mentioned idiom. The `err` variable will not be set and the function will always return 0, indicating that the certificate is always valid.
This scenario could have been easily avoided in mbeddr with a small language extension for a `try-catch`-like construct (shown in Figure 3). The idea is to execute the separate



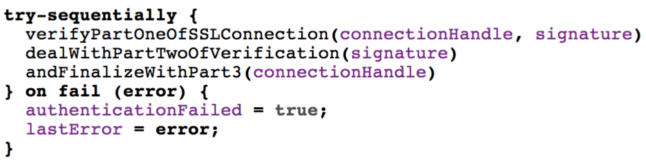
```
static OSStatus SSLVerifySignedServerKeyExchange(SSLContext *ctx,
bool isRsa, SSLBuffer signedParams, uint8_t *signature, UInt16 signatureLen)
{
    ...
    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    err = ...
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
        goto fail;
    }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

}
```

Figure 2: The original goto fail bug



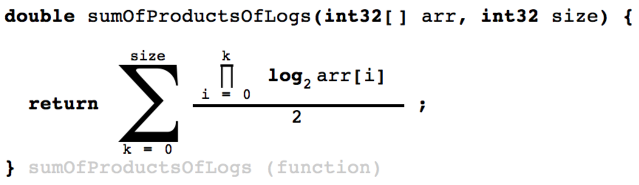
```
try-sequentially {
  verifyPartOneOfSSLConnection(connectionHandle, signature)
  dealWithPartTwoOfVerification(signature)
  andFinalizeWithPart3(connectionHandle)
} on fail (error) {
  authenticationFailed = true;
  lastError = error;
}
```

Figure 3: try-sequentially construct as a C language extension

branches of the multi-step error checking logic sequentially in the try block and free the developer from the need to write the individual return value checks. The construct will make sure that upon the presence of a non-zero return value, the logic in the catch block will be executed to handle the erroneous behavior. The generated C code will look exactly like the multi-step error checking in Apple's source code (with the if statements), but this low-level `goto` logic is automatically generated. This makes sure that errors such as the *goto fail* can be avoided. The extension code is more readable and less error prone. A modular extension such as this one can be built in 15 minutes; this is a nice example of the benefits of language extensions.
Another extension introduces mathematical notations to C (this functionality is part of mbeddr itself). Figure 4 shows an expression embedded in C code, which uses logarithm, fractions, summation and also muliplication. This seamless integration increases the readability of mathematical expressions and algorithms, making it easier to validate them (by simply looking a lot alike as they would do in a book or paper). A real-world project is using this language extension to implement complex calculation rules in the insurance domain.

5. CURRENT STATE AND OUTLOOK

mbeddr has been developed as part of a government-funded research project and it is now open-source software hosted at eclipse.org. Its continued development is ensured by itemis and fortiss; several other companies are interested and are investigating the use of mbeddr.



double sumOfProductsOfLogs(**int32**[] arr, **int32** size) {

return $\sum_{k=0}^{size} \frac{\prod_{i=0}^{k} \log_2 arr[i]}{2}$;

} sumOfProductsOfLogs (function)

Figure 4: Mathematical notation in the C code

mbeddr is currently being used by itemis' French subsidiary to develop a commercial smart meter. It is currently ca. 100,000 lines of C code. Components, units and state machines have been used successfully, significantly improving the testability (and hence, reliability) of the overall system. In addition, a set of commercial mbeddr add-ons are currently being developed by Siemens PL. The extension languages include data flow diagrams as well as systematic management of controlled names.

In the future we will be working on adding specific support for functional safety, including languages for code-integrated fault-tree analyses [2] and failure mode and effects analyses [1].

6. REFERENCES

[1] *Failure mode and effects analysis - http://goo.gl/3CoKV (as on 21.07.2014).*

[2] *Fault tree analysis - http://goo.gl/XQBevA (as on 21.07.2014).*

[3] *Jetbrains MPS (Meta Programming System) - http://www.jetbrains.com/mps/.*

[4] *The mbeddr project - mbeddr.com.*

[5] *The original source code for the Apple goto fail: http://goo.gl/ZiAFhk.*

[6] D. Ratiu, B. Schaetz, M. Voelter, and B. Kolb. Language engineering as an enabler for incrementally defined formal analyses. In *Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in*, pages 9–15, June 2012.

[7] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[8] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):339–390, 2013.