Controlling Software Complexity with Architectural Design

Keeping Code and Architecture consistent: Approaches and Benefits

Moritz Neukirchner and Guillaume Cordon, Elektrobit Automotive GmbH
Andreas Gaiser, Axivion GmbH

Managing software complexity is hard

Managing the complexity of a software project becomes increasingly difficult as the project grows and evolves over time. Software architectural design shall control this complexity by defining clear cut interfaces between components and permitted interactions among them.

Still, ensuring that the code adheres to the architecture specification is hard, even in the presence of reviews and integration testing. This challenge is even aggravated if multiple software variants must be taken into consideration and when portions of the code are being generated.

We show how this gap between architecture and code can be closed through automated checks. Complexity is managed at architecture level and the adherence to this architectural design enforced throughout the development process down to the code as deviations are automatically reported by a dedicated tool. As a result, the architecture can serve as a reliable documentation and as a discussion basis for design changes. Further, the automatic checks permit to evaluate the quality of design and implementation (i.e. their consistency), thus allowing to define a quality metric on designs beyond just simple review.

We illustrate the results using an automotive project and show that the approach scales from checking interface connections of components down to validating global data accesses.

Example of an Architecture Model

In order to automate consistency checks between architecture and implementation, strong UML guidelines need to be used across the tested software modules. In the following we use a dedicated example module (referred to as “design module”) to illustrate the required UML model scheme and how it relates to the information provided by the used static analysis tool.

Figure 1 shows a UML component diagram for the design module. It contains the adjacent modules and all required and provided interfaces which the design module uses.

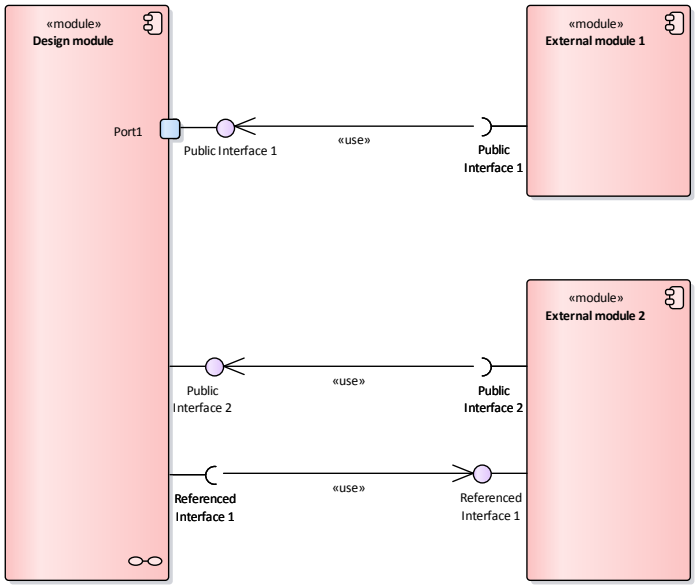

Figure 1- Component 'Design module' context view

Figure 2 and Figure 3 show how a UML component diagram is used to define UML Interfaces and the operations they implement with parameters and return values.

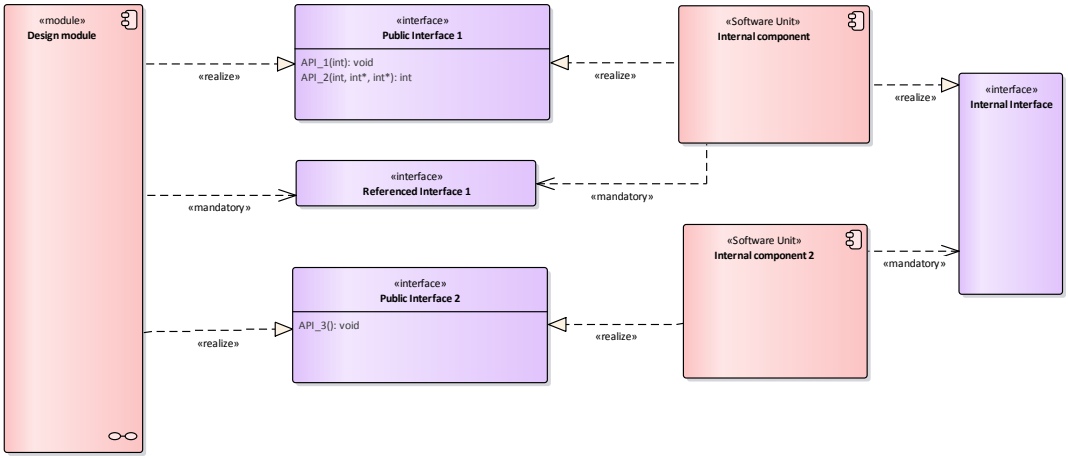

Figure 2 - 'Design module' Interface view

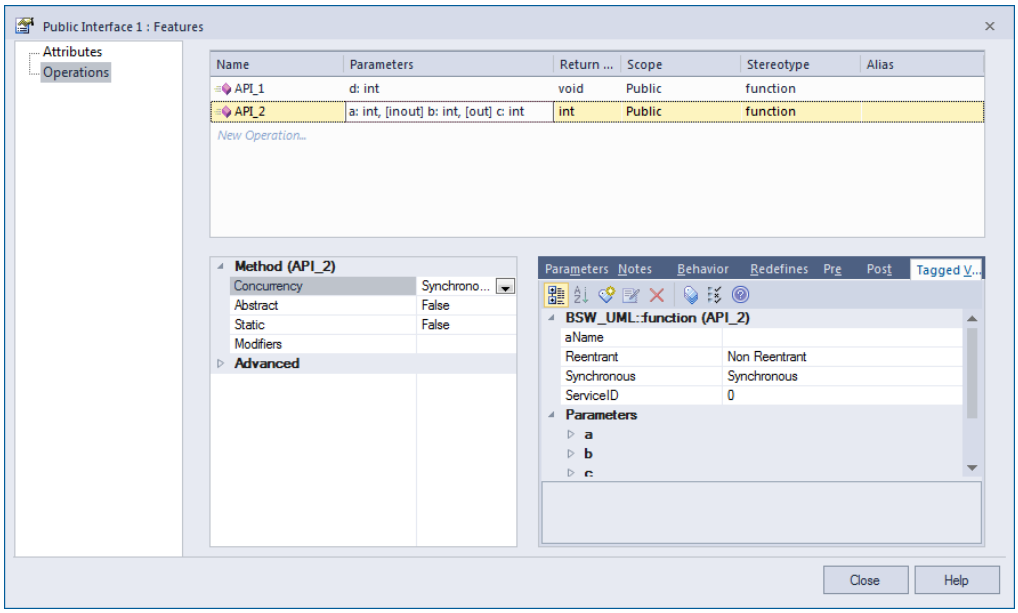

Figure 3 - Example of modelled interface and contained operations with parameters

UML representation of interfaces, operations and relationships between modules allows to detect whether the design module performs unspecified/unwanted static function calls to a different module using a public interface. Based on the same UML methodology, it is possible to define internal interfaces and sub-components for composed modules (see Figure 4).

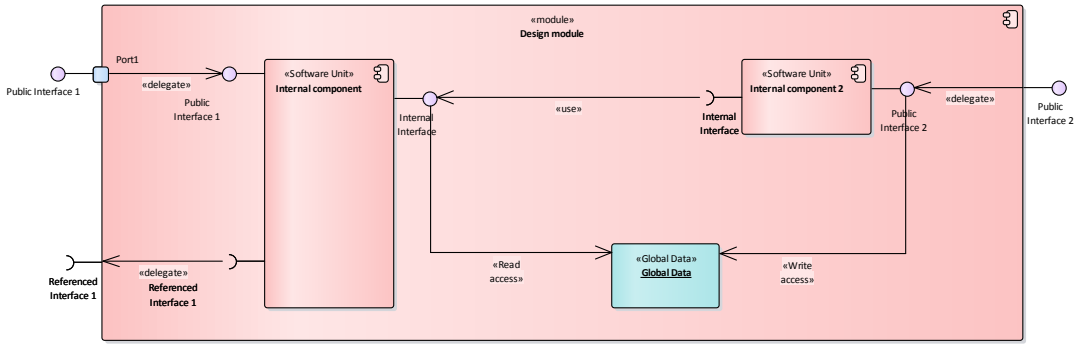

Figure 4 - 'Design module' Composition View and Data Accesses

The previous diagram allows to model global volatile run-time data and their data accesses. Derived from the interfaces view of the model, it is possible to extract how the design module shall access global data and therefore to apply checks on the module source code.

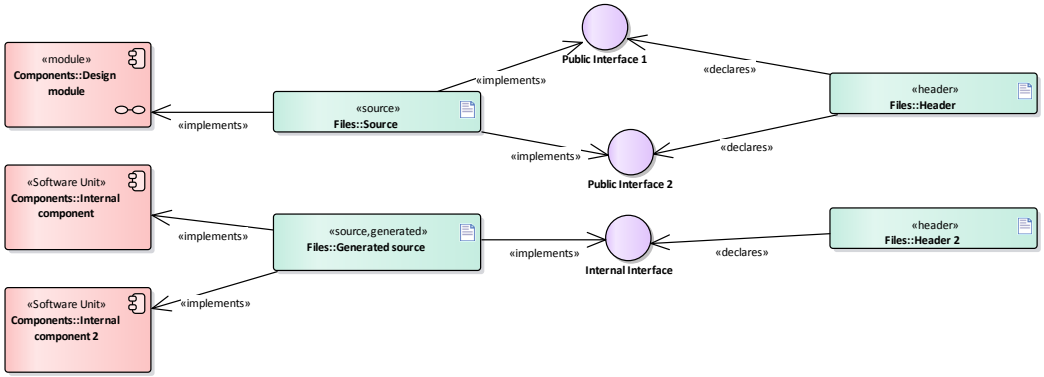

Figure 5 - File Mapping View

The file mapping view (Figure 6) allows to model the declaration/implementation relationship between the source code and interfaces.

The architectural model based on these strong UML guidelines form the basis for two aspects:

1. The automatic generation of the architectural documentation.
2. The automatic checking of consistency between architectural model and implementation.

As a result we can ensure that architecture documentation and implementation stay in-sync. In the remainder we focus on the automatic consistency check rather than the automatic generation of documentation.

Checking Architecture Consistency using Static Analysis

Our approach to automated architectural consistency checking via static analysis is split into the following three steps.

1. Extraction of code dependencies and architecture dependencies. Static analysis tools generate a *dependency graph* from a code base. In **Fehler! Verweisquelle konnte nicht gefunden werden.**, a project consisting of four code files is shown. A matching dependency graph is depicted in **Fehler! Verweisquelle konnte nicht gefunden werden.**, upper part. It uses node and edge types to express different kinds of code entities and relations. Nodes can be e.g. files, routines, or global variables. Further, they can be nested. Edge types include:

- "Static Call": e.g. the call between API_1 and API_2,
- "Declare": e.g. the declaration of API_1 refers to the definition in Source.c,
- "Variable Use": e.g. API_1, API_3 use the variable GlobalData.

If its format is flexible enough, it is also possible to enrich the dependency graph with architecture entities and dependencies. In this project we have used the Axivion Bauhaus Suite© for analysis. In Figure 9 lower part, an import of the UML component diagram for 'Design module' (Figure 1) is shown.

```c
/* File Source.c */

#include "Header.h"

int GlobalData = 0;

void API_1(int a)
{
    /* ... */
    if (GlobalData == 0)
    {
      GlobalData = 1;
    }
    /* ... */
}

void API_2 (int a, int b, int c)
{
/* ... */
}

void API_3 (void)
{
    GlobalData = 0;
}
```

```c
/* File Header.h */

#include "ExternalHeader.h"
/* Public Interface 1 */
extern void API_1 (int a);
extern void API_2 (int a, int b, int c);
/* Public Interface 2 */
extern void API_3 (void);
extern GlobalData;
```

```c
/* File Header2.h */
#if (CONDITION == TRUE)
#include "ExternalHeader.h"
#endif
#include "Header.h"
/* Internal Interface 2 */
extern void API_4 (int a);
```

```c
/* File GeneratedSource.c */
#include CONDITION TRUE
#include "Header2.h"
void API_4 (int a);
{
  /* ... */
  API_2 (a,0,0);
}
```

Figure 6 - Example of source code for 'Design module'

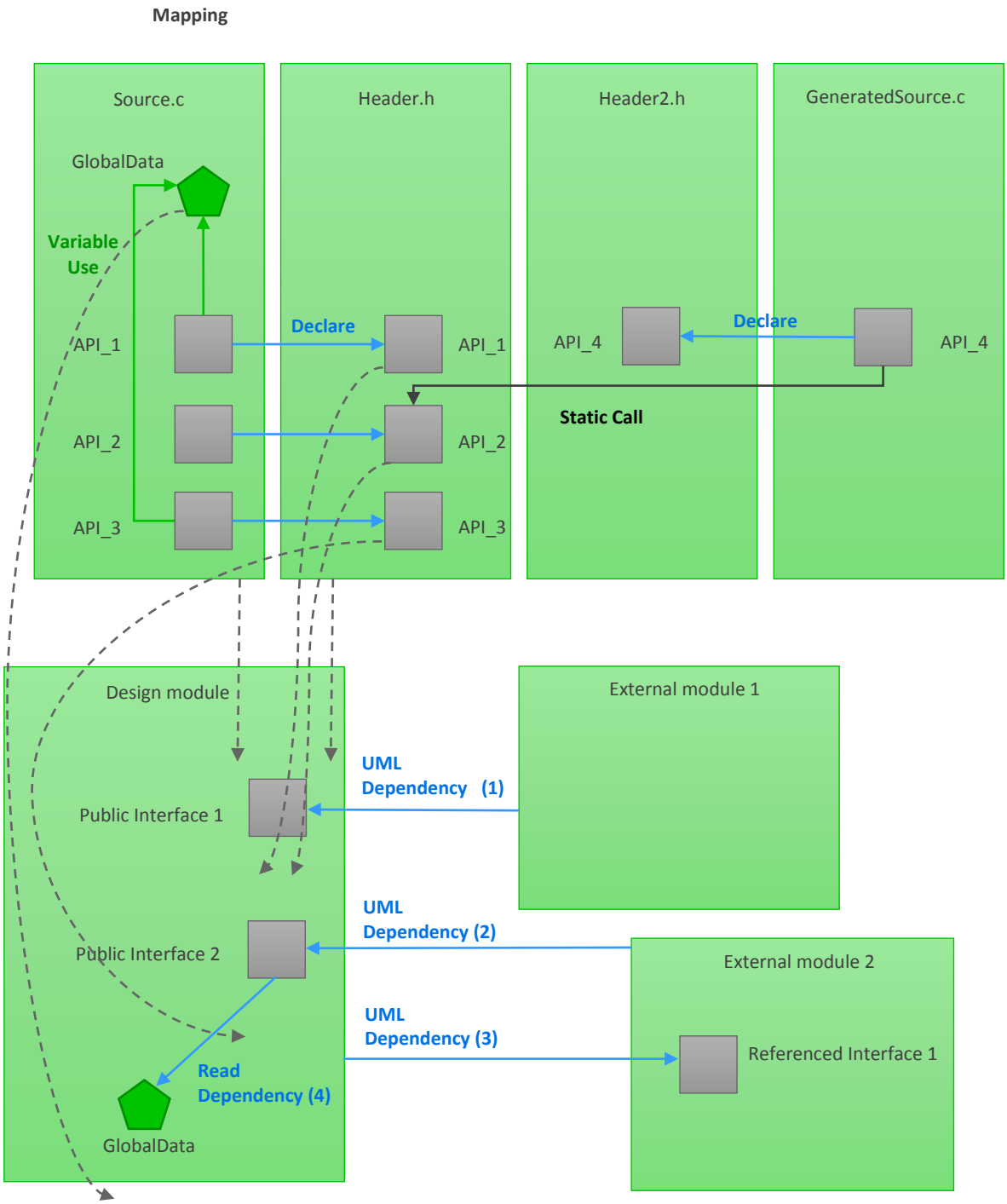

Figure 9 – Dependency Graph: the upper part are dependencies obtained from source code, the lower part is the represented UML model with semantic interpretation. Mappings are indicated by dotted edges.

2. Create a semantic mapping between dependency graph and architecture model.

For an automated check we have to define how code components *map* to architectural entities. The names of operations of an interface are explicitly specified in the Interface View (see Figure 2), which is used for mapping them to API functions. Similarly, global objects are mapped to global variables through naming patterns. Mapping between components and the files that implement them is extracted from explicit modelling as shown in Figure 6.

Further the *semantic interpretation* of dependencies has to be fixed: which code dependencies does a specific architecture dependency allow? In our project, we e.g. interpret the interface dependency 'Public Interface 1' between 'Design Module' and 'External module 1' as follows: each code entity mapped to 'External module 1' is allowed to call routines mapped to 'Public Interface 1'.

Both mapping and semantic interpretation are constructed automatically by Python scripts, which emit warnings if modelling errors or problems with the mapping occur. These provide valuable hints for the software architect.

3. Architecture conformance check.

The outcomes of steps 1) and 2) are used to perform the actual comparison between code and architectural model. We use the hierarchical reflexion model approach for the check (see [1,2]). The analysis detects the following three scenarios:

1. *Convergence*: a code dependency has a counterpart in the architecture model.
2. *Divergence*: a code dependency has no matching counterpart in the architecture model.
3. *Absence*: An architecture dependency is specified, but no convergences uses it.

Technical setup

In the analysed project, each module to be tested exists in multiple versions depending on the configuration that is used for code generation. Each such variant typically has multiple separate test cases to verify correct functionality. Therefore, it is necessary to perform the analysis on an extensive set of test cases, which may use differing code bases depending on the generated code.

For measuring the quality of both test case coverage and the architectural design, it is important to know whether

- Each interface dependency given in the architecture model is actually used in at least one of the test cases (*mandatory*). If an interface dependency is not used in any of the test cases, we call it a *global absence*.
- Code mapped to a module calls a routine belonging to an interface, but no corresponding interface dependency has been specified in the architectural model.
- Each access of a global object in the code is backed up by a corresponding access permission defined in the model.

Figure 1 shows the automated workflow for checking the questions from above:

1. For each test case T1, T2, T3, a dependency graph is generated.
2. The architecture model is imported into each dependency graph, mappings and semantic interpretation are created.
3. The actual architecture check is performed. Each divergence encountered in one of these checks hints to a flaw in either the code or the architecture model and is reported, as are accesses to global variables which are not allowed by the architecture model.
4. The architecture check results of the test cases are combined to detect *global absences*.

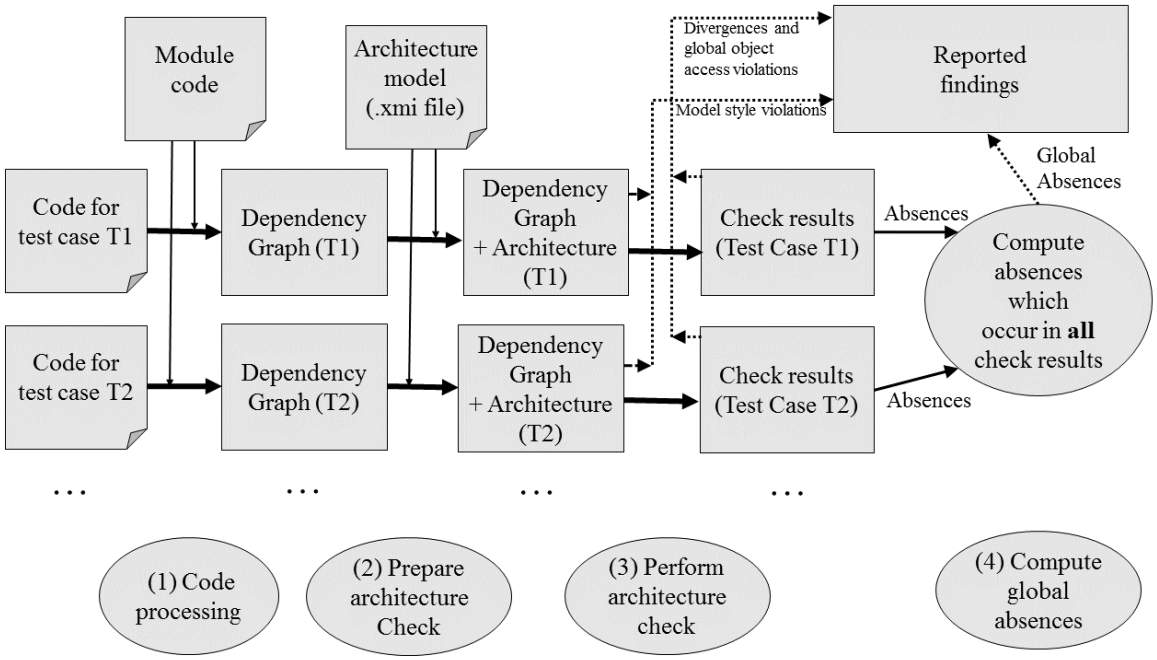

Figure 10 - Overall automatic aggregation process

The concrete case of our design module depicted in **Fehler! Verweisquelle konnte nicht gefunden werden.** is raising the following kind of findings:

Violation type	Architecture Source	Architecture Target	Dependency Type	Comment
Absence	Design Module	External Module	Static Call	Design module shall call *Referenced Interface 1* according to architecture model
Divergence	Design Module	Internal component 2	Static Call	API_2 is called from API_4 (internal interface) but not modelled

Figure 11 - Architecture violation findings

Error	Message	Comment
GlobalVariableInvalidAccess	Global object "GlobalData" is read-accessed by routine "API_1"	Unmodelled data access
GlobalVariableInvalidAccess	Global object "GlobalData" is write-accessed by routine "API_1"	Unmodelled data access
GlobalVariableInvalidAccess	Global object "GlobalData" is write-accessed by routine "API_3"	Unmodelled data access
InterfaceWithoutImplementation	Interface "API_4" has no incoming <<implements>>-edge.	
InterfaceWithoutDeclaration	Interface "API_4" has no incoming <<declares>>-edge.	

Figure 7 - Style violation findings

Conclusion

We have shown how the management of complexity in software development can be assisted through automatic checking of software designs. Based on the requirements of a productive automotive software project, which includes the challenging aspects of a multitude of variants and a large degree of code generation, we have demonstrated that deviations in implementation from the design specification can be effectively detected in an automated way.

This approach guarantees that implementation and design stay in sync such that the design can serve as a reliable documentation of software. Further, it permits to define quality metrics on design documents beyond just simple review results as deviations can be properly quantified. This allows to enhance quality control and to prevent that design and code diverge over time, i.e. that the quality of the design document deteriorates over time.

The approach was tested on architectural properties such as interface definitions and their connections, file structure and global data accesses. Nonetheless, the approach is not limited to these aspects.

The approach is in use in the qualification of productive code and is currently being used to detect undocumented dependencies between software components and undocumented global variable accesses.

References

[1] Rainer Koschke and Daniel Simon. “Hierarchical Reflexion Models”. In *Proceedings of the 10th Working Conference on Reverse Engineering* (WCRE '03). IEEE Computer Society, Washington, DC, USA.

[2] Rainer Koschke. “Incremental reflexion analysis.” In *Journal of Software: Evolution and Process* (2013), Volume 25, Issue 6.