On the Systematic Conformance Check of Software Artefacts

Hylke W. van Dijk, Bas Graaf, and Rob Boerman

Delft University of Technology, Software Technology (EEMCS),
P.O. Box 5031, 2600 GA Delft, The Netherlands
{H.W.vanDijk, B.S.Graaf, R.Boerman}@ewi.tudelft.nl

Abstract. In this paper we present a systematic check of the conformance of the implemented and the intended software architecture. Nowadays industry is confronted with rapidly evolving embedded systems. In order to effectively reuse design artefacts such as requirements, architectural views and analysis, as well as the code base, it is important to have a consistent overview in each phase of the development process. In this paper we propose a conformance check framework that combines a colloquial engineering model and a conformance check system based on commodity technology, albeit the model and the system can be used in their own right. An academic experiment exemplifies the application of our framework.

1 Introduction

The current trend in embedded systems is product families rather than single products. Today's customers appeal to products that have a sense of uniqueness, products that are compatible but slightly different than those of their friends. The answer from industry is to set-up flexible product lines, which include a range of disciplines: from product development to product manufacturing. The efficacy of these product lines for evolving systems is mainly determined by the amount and ease of reuse of existing artefacts.

The maintenance phase of a product has always been significant and will increasingly be so. The growth of the complexity of systems is one reason, the trend towards product families is another reason. From our participation in two international industrial research projects [1,2] we learned that new products are rarely developed from scratch and that reuse of existing development artifacts is typically ad-hoc [3]. These observations triggered research in the field of conformance checking as a first step in enhancing the functionality of a product or adapting it to a changed environment. The conformance check baselines a consistent set of development artifacts and as such provides a starting point for a more structured approach to reuse.

We address the problem of conformance checking by means of a conformance check framework, describing the necessary steps. In order to be practically applicable in industry, it is required that such a framework suits the development organisation, builds on proven technology, and that its application is non-intrusive.

R. Morrison and F. Oquendo (Eds.): EWSA 2005, LNCS 3527, pp. 203–221, 2005.

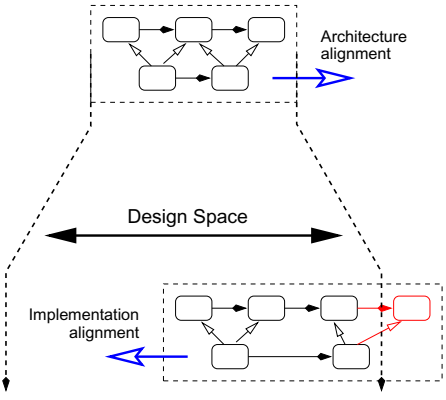


Fig. 1. Aligning architecture intention and implementation realisation

In general conformance checking could be applied on all related artefacts produced by the different domains of expertise in the software development process. In this paper we focus on the coordination between the domains of architecture and implementation.

The communication between the two domains is through views that are associated with a viewpoint [4], addressing a specific set of concerns. Views are developed in the architecture domain of expertise to specify the intentions, design restrictions, and design permissions for the eventual product implementation, i.e., the design space of Figure 1. Views can also be generated in the implementation domain of expertise [5]. These views predicate the properties of the actual implementation from an architectural perspective.

When views from the architecture and implementation domain are associated with a common viewpoint it is possible to identify discrepancies between the intended and implemented architecture. In Figure 1 there is apparently a mismatch, which can be resolved by either updating the architecture or the implementation. However, the semantic gap between the elements and relations used in architectural views and the programming language constructs available to implement them makes it difficult to reconstruct a view associated with an architectural viewpoint from an implementation.

In this paper we propose and experiment with, a conformance check framework (CCF) that combines a colloquial engineering model and a conformance check system (CCS). The CCS facilitates conformance checks through the definition of a design-space conformance viewpoint bridging the semantic gap between the implementation and architecture domain. Views associated with this viewpoint *can* be generated from the implementation and derived from the architectural views. The engineering model takes an architectural view on product development. It is based on the two principal categories of views described in literature [6,7,8]: runtime views and development views. We consider views from both categories in our experiments to attain a good coverage of the difficulties

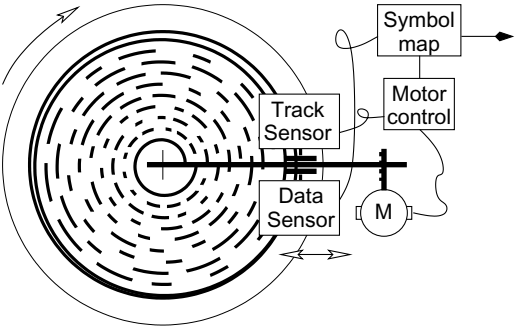


Fig. 2. Digital music box reader system

and possibilities of determining architectural conformance. We implemented the CCS using existing technology. Our experiments demonstrate the definition of conformance viewpoints and the visualisation of discrepancies between the intended (specified) architecture and the implemented (predicated) architecture. The CCF emphasises the role of conformance checks for maintainability of operational systems. Because of our academic interests, part of the result of our treatise will be an agenda for further research.

This paper is organised as follows. This introduction is concluded with the presentation of a running example. In Section 2 we present our conformance check framework, the two principal categories of views of the CCF and their relation are treated in detail in Section 3. Section 4 is devoted to a case study where we systematically conform a viewpoint from the set of runtime views and a viewpoint from the set of development views, thus implementing the CCS. In Section 5 we discuss our CCF, the applied technology, and related work. Finally, Section 6 presents conclusions on our work.

1.1 Running Example

The running example in this paper is the development of an academic system: a digital music box (DMB) that reads data from a paper disc (record). The record is a plotted spiral track of pulse-width modulated data bits. The disc rotates with a constant speed. The system tracks the spiral, reads the data bits, and then maps those bits to symbols. A string of symbols will be fed to an output device that transforms the string into audible music. Here we focus on the process of reading the record and the generation of the symbol stream. The physical system is composed of a traditional turn table and a set of simple light sensors that can be moved axially by a motor; the control is implemented on a simple micro controller. The controller is programmed in Java. Figure 2 gives an overview of the reader system.

2 Conformance Check Framework

Architecture is typically described using different views each addressing a different set of concerns. We therefore need to be explicit about the architectural view involved in the conformance check. For this purpose our framework includes an engineering model that involves the two principle categories of views for conformance checking: development views and runtime views. The model is complemented with a strategy for conformance checking and a generic conformance check system. The latter can be implemented with readily available technology.

2.1 Engineering Model

Our engineering model shows the architectural views involved in a product's life cycle. To position our engineering model, we consider a generic product model for embedded system development: the Vorgehensmodell (V-model [9]).

The V-model binds the analysis or design activities of product development with the synthesis or integration activities. Given a context with changing requirements and environment, and where new products are not developed from scratch, it is essential that the process model facilitates flexible interactions between different domains of expertise in the software development process. Thus generally, specifications flow forward from analysis to synthesis, whereas the return flow from synthesis to analysis carries predicated properties of the system.

We take an architecture centric position to product development. The architecture of a system provides a reasoning framework of that system; it is a common understanding of the involved stakeholders. Ideally the architecture

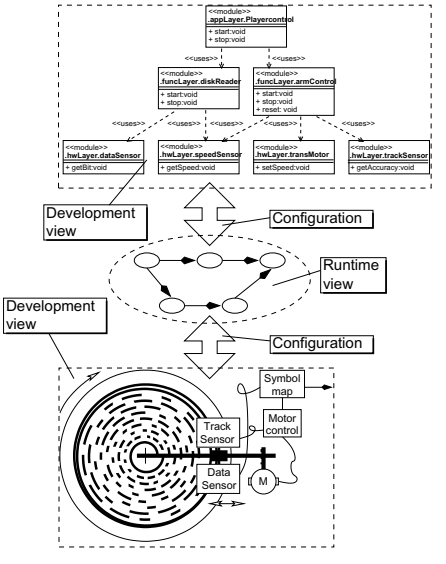


Fig. 3. Architectural views in the engineering domain

explicitly rationalises all important design decisions, in practice however architecture documentation mainly concerns the structural effects, manifested in the graphical presentation of the different architectural views [6, 4, 8].

The understanding of the *working* system takes a central position in the communication among stakeholders. This is indicated by the central role of the runtime view in Figure 3. The process view in Kruchten's "4+1" view model [6]; the different component-and-connector (C&C) views described in [8]; and also data and control flow diagrams known from structured analysis and design methods are all known examples of views that capture the structural organisation of a *working* system. Runtime views address the question: how does the system work? In order to arrive at a system functioning as presented in the runtime views, the periphery of the runtime views consists of views driving the actual *implementation* of software and hardware. Those views that capture the structural organisation of the implementation units are known as the set of *development views* and address the question (concern): how is the system developed? Examples of development views on the organisation of the software implementation units (modules) are decomposition views and uses views [8].

The compilation *configuration*, indicated by the double arrows in Figure 3, describes the integration of the constituent elements of a system, as described in the development views, into the working system, as described by the runtime views. It specifies the allocation of software modules of a development view to components of a runtime view and it additionally describes the allocation of that component to a hardware unit of the appropriate development view.

Figure 4 situates our engineering model in the V-model. It shows the two principal sets of views (the runtime view and the development view) and their configuration. Obviously development views are mainly used during coding and

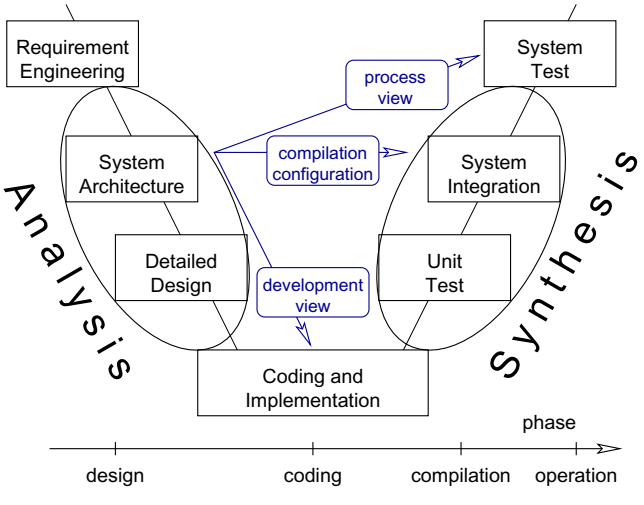


Fig. 4. V-Model engineering development model

implementation, runtime views are used to operate the system, compile configurations are used during the system integration (compilation) phase.

2.2 Conformance

Our engineering model suggests to check views from the set of runtime and development views, in order to arrive at a sufficient coverage of discrepancy detection between the specified and implemented system. A distinct constraint for a practical implementation of a conformance check system is the prevention of interference with ordinary system development. Therefore we regard the domains of architecture and implementation as autonomous activities, circumventing the use of an integrated development model.

practice, the implementation and architecture domains differ considerably with respect to the level of detail at which the involved concepts are defined. Implementation level constructs can be defined formally. At least there is a compiler that deterministically attaches meaning to implementation concepts. In the architecture domain, however, concepts do not typically have a universal, unambiguous meaning and their semantics is only specified, if at all explicitly, informally. Taking into account the fact that architectural decisions are typically made in the early phase in a product's life-cycle, we consider this a virtue of the architecture domain.

A conformance check between a specification and predication among two domains of expertise without affecting them requires the definition of a *common viewpoint*. The semantics of such a design-space conformance viewpoint must be clearly interpretable by both domains of expertise. Thus there must be a bidirectional *mapping* between the design-space conformance view and the domain-specific views. Mismatches between the design-space conformance views derived from the two domains-specific views identify potential discrepancies, or architecture violations. Whether a mismatch indeed implies a discrepancy involves more detailed knowledge of the relevant design decisions.

In Section 3 we develop design-space conformance viewpoints for a runtime view and a development view, plus their respective mappings from a typical set of architectural views and implementation views. Although implementation will mainly use the development views, it must obey the runtime views so as to facilitate their proper implementation in later stages of the product's life time. Predicated properties of the realised system contain the evidence that the development view has been properly implemented and that the runtime views can be realised in later stages of the product's life time. Our conformance check system (Section 2.3) gathers and extracts the specifications and the attributes in terms of the predefined conformance viewpoints, checks their conformance, and visualises the result.

In this paper we consider the architectural view as leading. From the perspective of an implementation there are three important situations for any entity or relation: *covered*, *excess*, and *deficit* [10]. A covered relation (or entity) has a corresponding relation in the architecture domain. An excess relation only exists in the implementation domain and a deficit relation only exists in the architec-

ture domain. The result of a conformance check is a set of entities and relations that are attributed according to the three types above; the significance of mismatches found depends in general on the involved design decisions. Therefore discovery of mismatches should serve as a trigger to investigate further if they are allowed and possibly documented elsewhere. If not, they are considered to be discrepancies that reduce the conceptual integrity of a system and may result in unexpected dependencies, reducing the system's maintainability.

A conformance check can be used to evaluate specific concerns. As an example, we can locate the set of design-decision related to a selected quality attribute. The set of design decision determines a set affected views. This set of views can be used to define a design-space conformance view, the set of design decisions determines the impact of possible discrepancies found during a conformance check. An alternative use is to incorporate a conformance check in the system integration test set of a product, e.g., as part of the nightly build process.

2.3 Conformance Check System

The conceptual conformance check system (CCS) of Figure 5 outlines the process to identify discrepancies. A fact extractor derives views, associated with the design-space conformance viewpoint, from architecture and implementation domain artefacts. The subsequent comparison of the derived design-space conformance views is done based on a set of comparison rules. These rules are used for simple graph matching to identify mismatches, and more involved for selecting those mismatches that are actually related to discrepancies. Finally, a presentation filter visualises the comparison results.

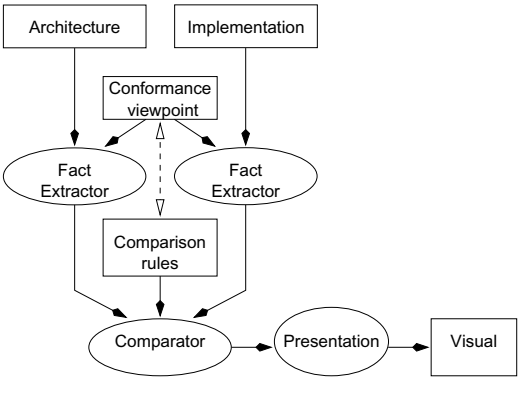


Fig. 5. Conceptual CCS

3 Deriving Conformance Viewpoints

In this section we describe how design-space conformance viewpoints can be derived for the two principle categories of architectural viewpoints. In particular

we consider suitable (informal) semantics of the conformance viewpoints, their checkable mismatches, and their respective mappings from both the implementation and the architecture domain viewpoints.

3.1 Development Views

A development view describes a decomposition of the system in terms of implementation (e.g. source code) units, often called modules, and their dependencies. These modules, supposedly coherent units of functionality, are eventually assigned to development teams. Dependency relations between the modules of a development view are important, several types of them exists such as uses, allowed-to-use, and shares-data-with relations. Here we will focus on *use* dependencies.

Figure 6 depicts a development view of the digital music box architecture of Section 1.1 that reveals the use dependencies between the different modules. This view is part of the specification of the system, resulting from the architecting phase. The chosen viewpoint contains a module element and a use dependency relation, both indicated by UML stereotypes.

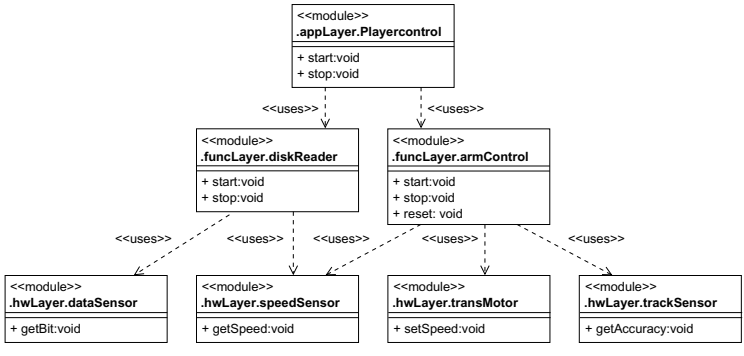


Fig. 6. Uses (module) view

Typically the implementation-level modularisation constructs do not match one-to-one with the architecture-level modules. Implementation engineers typically have reasons to further refine the provided decomposition of the development views. This is safe, provided the decomposition is registered, e.g., annotated. A simple, yet sufficient, method is to augment the implementation with *belongs-to* clauses that associate decomposed subsystems with a module of the architectural uses view. With the advent of integrated development environments dealing with these annotations is simple. Eclipse could, for instance, be easily changed such that this information is requested from the programmer in the wizard for defining a new class. Subsequently this information could be included in the header of the skeleton code generated by the wizard. The belongs-to clauses can be automatically retrieved during the fact extraction phase. Gathering these facts yields a design-space conformance view consisting of aggregates

of implementation units. As an example consider Listing 1 where the following belongs-to relations are defined: BelongsTo($X; A$), BelongsTo($Y; B, C$).

Next to modules a uses view defines use relations [8]. The mere wording of use has conflicting interpretations [11]. As we strive for a clear definition of viewpoints, the meaning of "use" has to be clearly defined in order to determine the existence or possibly inexistence of a particular use relation. We start with the definition given by Clements et al [8]: "Unit A is said to use unit B if A's correctness depends on a correct implementation of B being present."

We take a pragmatic position by mapping the architectural use relation to a checkable tuple: a link plus an action that effectuates the link. The link is a reference to the used module and the action can be anything from a function call to an attribute access. This design-space conformance viewpoint only covers part of the architectural concept of using, as it does not take into account that the used module needs to be implemented *correctly*, it merely requires it to be present. Furthermore the architectural uses dependency does not necessarily require a direct reference in the implementation; more complicated indirect dependencies can also correspond to a use relation. In fact the design-space viewpoint captures calls and shares-date-with dependency relations, which are different specialisations of the depends-on relation.

A link from module X to module Y typically emerges as a reference, e.g., a declaration of an attribute of type B in class A, where A and B belong to X and Y respectively. The necessary action is determined by a method invocation or attribute access of that reference. Combining the links, actions and belongs-to relations, the example of Listing 1 contains the following use relations, as defined

```
// @belongsTo module X
Class A {
    private B objB;
    void A() {
      B = new B();
    }
    void doA(C objC ) {
        B.doB( C );
    }
}

// @belongsTo module Y
Class B {
    void doB( C objC ) {
        C.doC();
    }
}

// @belongsTo module Y
Class C {
    private A objA;
    void doC() {
        // stub
    }
}
```

Listing 1. Sample source code

by the design-space conformance viewpoint: Uses$(Y;X)$ and Uses$(Y;Y)$. The absence of a relation is not a property (fact), e.g. Not Uses$(X;Y)$.

3.2 Runtime Views

Runtime views on software architectures are frequently specified using component-and-connector (C&C) views [8]. The box-and-line diagrams created early during software design, usually are C&C type of views. C&C views are detailed runtime views addressing concerns such as concurrency and flow of data. Architectural components are loci of computation and state. Architectural connectors are loci of interaction. Both are architectural abstractions of elements that consume resources, either processing time or memory. A complete C&C view is an abstraction of a system during runtime.

To describe C&C views we adhere to the terminology of architecture description languages (ADL), e.g., [12]. Typically in C&C views components are associated with the connectors by means of attachments. Components and connectors habitat *processes* that interact with their environment through associated *interfaces*. In case of a component the interface is called a *port*, whereas in case of connector we call it a *role*. In order to establish interaction between two components over a connector we can attach component ports to connector roles, with the limitation that an attachment is only allowed if the component interacts using the port as interface and according to the expectations described by the connector role, i.e. port and role need to be compatible.

Figure 7 depicts a runtime view of the digital music box architecture of Section 1.1. It shows concurrently executing components as communicating-processes. The components interact through different types of connectors. Although UML is not the preferred modelling language, mapping of ADL constructs to UML is sometimes awkward, it can be done [13, 14]. Following an approach proposed by Garlan et. al. in [14], we represent component types and components by classes and objects, connectors by links (labelled with connector type names), and ports by link-roles (labelled with port type names). As we are only using relatively simple connectors we do not consider connector roles.

Components, ports, connectors, and roles are architectural concepts that may or may not have explicit counterparts in the development views or implementation. Source code is not merely a refinement of these architectural elements as in the case of development views, making the mapping between the architecture runtime views and implementation domain constructs indirect and more difficult.

The main concern of the C&C view in Figure 7 is concurrency. For such a view the components, processes, correspond to implementation mechanisms for concurrency and parallelism, such as processes, threads and tasks. For example in the case of a system implemented in Java, a component corresponds to a subclass of the thread class and all other classes it instantiates.

Connectors correspond to the mechanisms that allow these threads and tasks to interact, for instance inter-process-communication mechanisms, remote-procedure calls, or shared-data. As opposed to the architectural connectors these im-

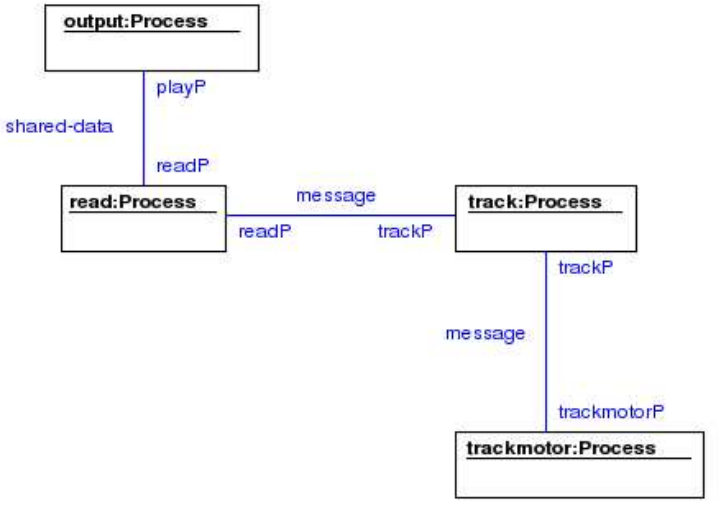


Fig. 7. Communicating-processes view

plementation-level communication constructs have an obvious direction. Therefore in the design-space conformance view we add a direction to the connectors defined in the architectural view. We need to consult the architect or the architecture documentation to discover the intentions. For message-based connectors the direction corresponds to the direction of the first message, i.e. from the component initiating the interaction to the other component. The direction of shared-data connectors is from the component writing in the shared-data to the component reading from the shared-data, assuming that components do not read *and* write to the shared-data, which in our case was a valid assumption.

4 An XML Implementation of the ccs

Our sample implementation of the CCS, depicted in Figure 8, uses readily available XML technology; the XLINKIT toolkit is the heart of our CCS.

Fact extraction involves two steps: a transformation of the sources (architecture and source code documents) onto XML format followed with a filtering and interpretation operation to populate the design-space viewpoint. In our experiment, the architecture has been described in UML, using its accompanying XML schema: XMI. The implementation has been coded in Java. The (static) facts about the implementation reside in the Abstract Syntax Tree (AST), which can be retrieved by replacing the code generation back end of a compiler. In this case we used an XML specification JAVAML [15] that is generated by a patched version of the Jikes compiler. Similar technology is available for many other programming languages [16].

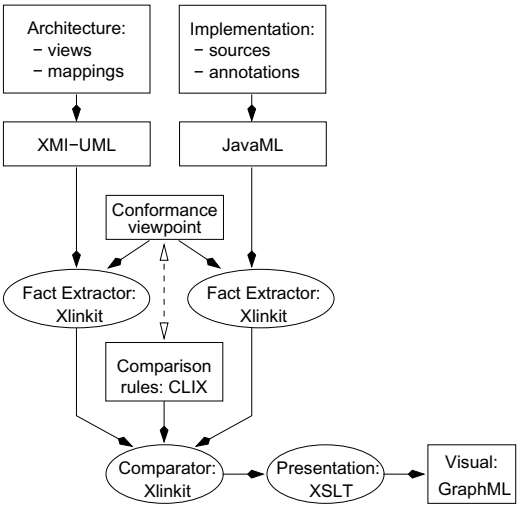


Fig. 8. Implementation of CCS

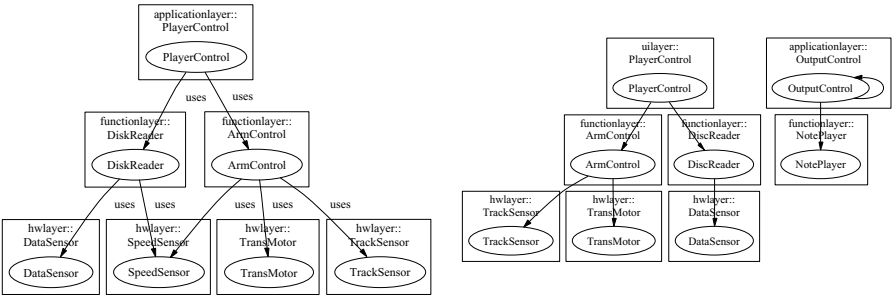
The filtering operation uses the XLINKIT technology [17] as a lightweight rule-based link generation tool. It combines current XML technology such as XPATH, XLINK, and XSLT. The XLINKIT rules that map architecture and implementation domain facts to the design-space conformance viewpoint (see Sec. 3) Figure 8, are specified in so-called CLIX [18] constructs.

The XLINKIT tool is also used as a comparator in the CCS. It takes the XML representations of the design-space conformance views and a set of CLIX comparison rules, which identify possible mismatches between the extracted conformance views and subsequently identify those mismatches that actually involve discrepancies. The result is a set of XLINK hyperlinks between the two conformance views. The comparison rules basically check the semantic consistency of the two design-space conformance views. The hyperlinks either represent evidence of a correct (covered relation) or an incorrect (deficit or excess relation) interpretation of the architectural views by the implementation domain.

Finally the information presentation phase takes the hyperlinks and produces a visual representation. This is done with an XSLT transformation engine. The result is in our case a graph, which is specified in GRAPHML, a flexible XML schema. Graph visualisation and layout tools are indispensable for the interpretation of the results, here we used GRAPHVIZ and YED.

4.1 Development Views

The uses view of Figure 6 was one input of the CCS. Derivation of the design-space conformance view from this UML model is straightforward because of the use of stereotypes to denote modules and use relations. Simple XPATH expressions suffice to generate a canonical XML model for the following phases in the CCS. A visual representation, through XSLT, is given in Figure 9(a).



(a) Architecture

(b) Implementation

Fig. 9. Uses views

Recovering the design-space use-view from the implementation involves the interpretation of Java language constructs and the belongs−to annotations. Locating a module is simply done by retrieving the belongs−to attribute of identified classes in the sources. Locating a use-relation is more involved however. We demand *access* from a source class to a target class. It is insufficient for a source class to only maintain a reference to a target class or invoke a constructor for that target. Access involves the explicit invocation of a method in the target class or a field update of the target class. The CLIX expression to identify access is not very elegant, merely enumerating all possibilities. The resulting visualisation is given in Figure 9(b).

The comparison phase of the CCS co-locates facts from the conformance views extracted from the architecture and implementation domains. It binds entities from both domains through their names and determines covered, excess, and deficit relations. Note that in the current implementation we determine manually which mismatches actually involve discrepancies.

In the presentation phase of our CCS we map the entities and relations to a graph in which covered, excess, and deficit relations are coloured and shaped. The result is in Figure 10. The trapezoid shaped vertices and the edges with open delta arrowheads represent the deficit entities and relations respectively. Partly these relations originate from name mismatches, e.g. DiskReader and DiscReader. One entity has not been implemented: the SpeedSensor. The parallelepiped shaped vertices and the closed delta arrowheads represent the excess relations, e.g, showing name mismatches. But also real excessive relations emerge: NotePlayer and OutputControl. The covered relations use boxed vertices and sharp short arrowheads on the edges.

4.2 Runtime Views

The design-space conformance views resulting from the fact extraction in the architecture domain is given in Figure 11(a). It corresponds to the component-and-connector view in Figure 7. As explained in Section 3.2 we necessarily added

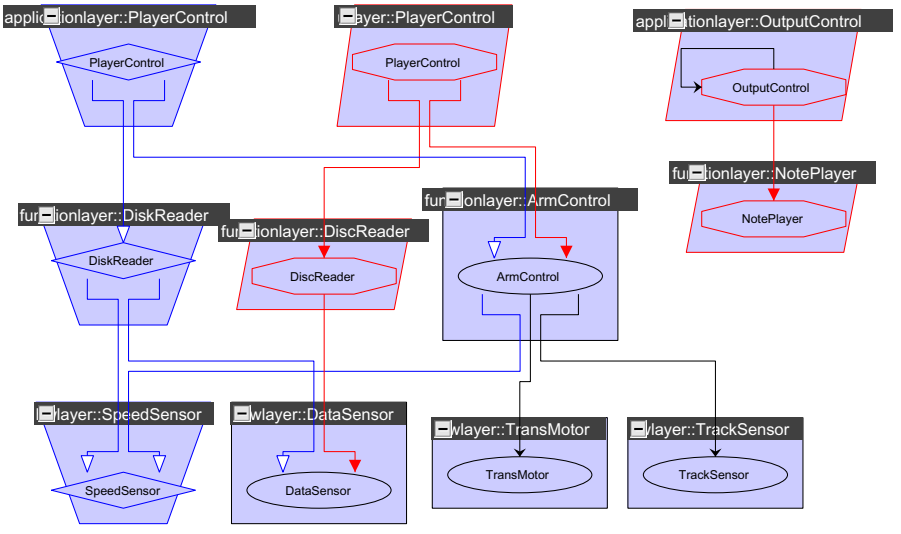


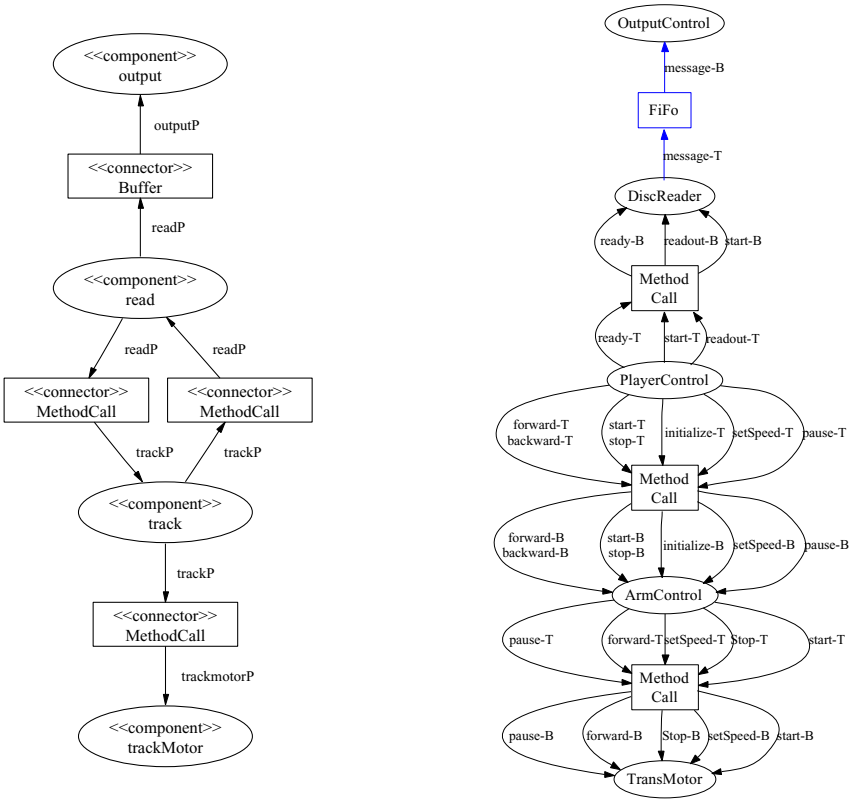
Fig. 10. Uses conformance view

direction to the connectors. In the design-space conformance view we represent components by rectangles and connectors by ellipses with arrows to indicate the direction.

Creating a C&C view from static sources is very application specific. For the transformation of sources to XML of the data gathering phase we use the same technology as for the development view. The filtering stage is now a multi-stage approach that extracts, combines, and interprets facts by cascading XLINKIT extracted reports. In this case we used two stages.

The first stage extracts associations from the source code and identifies autonomous threads of control. The associations identification reuses the CLIX rules of the static case. Autonomous threads are, in this case, defined as classes with a main-method or classes that extend the Java thread class. The second stage gathers the actual instantiated threads as well as interaction between thread instances. We recognise two types of communication links: a method call and a buffered stream with read and write access. The resulting graph is given in Figure 11(b).

Comparing the C&C runtime views of Figure 11 involves as before merging the namespaces. We have multiple mismatches here, as the names in the source code are derived from the names used in the *development* view. Furthermore implementation-level constructs used to implement interaction are often not named, e.g. procedure calls. Subsequently the components, connectors, and ports must be identified. Since the recovered C&C views from the sources lack ports and roles altogether we transform the conformance view by inserting ports and roles when appropriate. Comparing the design-space views results in the identification of covered, excess, and deficit constructs.



(a) Architecture (b) Implementation

Fig. 11. C&C views

The resulting diagram is not shown, but considering the namespace mapping of the components (output ↦ OutputControl, read ↦ DiscReader, track ↦ ArmControl, and trackMotor ↦ TransMotor) of the two views in Figure 11 it is immediate that the connectors are more detailed in Figure 11(b). This is because it shows the different occurrences of interaction over each connector separately. Furthermore the PlayerControl component is an excess component not present in the architecture specification. It was intended as a connector between the read and track components, however in the implementation it included handling user interaction for which it required a separate thread.

5 Discussion

In this section we discuss issues that emerged in the execution of the experiment of the previous section.

Consistency and Technology Imperfections. The architectural views are expressed in UML, using stereotypes to identify modules and use dependencies. However, the current version of UML cannot enforce the consistent use of stereotypes, which potentially may yield false alerts in our CCS. Additionally UML does not offer support to conveniently specify all elements of component-and-connector views. For instance, the style we used to describe the component-and-connector views in UML does not represent connectors as first-class UML modelling elements, making it inconvenient to specify connector types and properties. The forthcoming UML 2.0 standard expectedly has better support for specifying software architectures.

Similarly, extracting the uses conformance view from the sources too depends on the applied (programming) style, e.g. usage of patterns, coding conventions, and so forth. Our systematic approach to conformance can be extended partly to the domain of consistency verification. Modelling style and programming style violations can be captured in rules that when checked provide insight in the overall consistency of the design or implementation [19] with respect to these rules. It is generally accepted that consistency is a desirable quality attribute.

There are two more unfortunates with our method of identifying use relations in the implementation domain. First it is unclear whether the implemented enumeration to locate use relations is complete and second the applied XPATH technology does not support a transitive closure function, which is necessary to handle nested access (e.g. A.B.C.foo()). The required breadth of the enumeration depends again on the programming style. For instance, the use of getter and setter methods circumvents the need to look for direct field access.

Intriguing research questions are to what extent we can include consistency checks in the mapping from the architecture and implementation domain views to the design-space conformance views and whether we can circumvent the need of enumerations in rules. The latter could be realised, for instance, by using canonical (intermediate) representations of artefacts?

Conformance Interpretation. In the conformance phase of the CCS, we merge the namespaces of the architecture and implementation domains. The current implementation uses string matching. Because of the human in the loop this is a workable situation. An alternative approach for string matching would be a graph matching and merging approach, in theory, because these algorithms execute in non-polynomial time. Graph matching thus automatically retrieves part of the mappings from the architecture and implementation domains to the design-space conformance viewpoint, e.g., compare the Figures 11 and 7.

Conformance checking identified covered, excess, and deficit constructs. This seems a sensible situation. However situations may occur that require more detail, i.e., specialisation of the identified constructs; an excess construct, for instance, may emerge due to namespace mismatches.

Research questions here involve means to specify, and possibly automatically resolve, conformance check results.

System Dynamics. Our CCF does not address the behaviour of the implemented system. Although various formalisms exist to describe the intended behaviour of a system, e.g., CSP and statecharts, these are not commonly used in practice [3]. Requiring such a view would therefore interfere with our prerequisite to develop a non-intrusive method for conformance checking. Furthermore, we use static views derived from the source whereas proper validation of the correct behaviour requires run-time information.

The runtime view conformance check has been executed based on a straightforward static code evaluation. This approach has drawbacks. Simple static evaluations consider the entire design space, including configurations that will not be reached in reality. Another option is to use run-time evaluations, however such a method is confined to the set of configurations of the executed test set. Alternatively sophisticated parsing and graph rewriting techniques could be used. In our implementation we rely on the consistent use of an architecture model with carefully chosen naming conventions. This yields static attachments that reveal the system configuration in the parse tree. Dynamic attachments cannot be retrieved this way.

The research question here is to find flexible parsing and logical reasoning techniques, maybe in combination with the use of runtime information.

More Related Work. Conformance checking is a systematic and quantitative approach that gives an indication of maintainability, whereas better known architecture evaluation methods such as scenario-based assessment methods (see [20]) and inspection methods (see [21]) are qualitative methods.

Methods for systematic architectural conformance checking have independently been compared in [22] and [23]. They categorise methods for architectural conformance checking, such as software reflexion methods [10] and their own expressive methods. In [22] design-space viewpoints are defined in a relational partition algebra, whereas in [23] a logic meta-modelling technique is used. Our CCF adopts the pragmatics of the efficient methods, while introducing a semantic interpretation of available artefacts.

An alternative would involve the use a code generation framework, such as the Ptolemy framework [24], extend the implementation language with architectural constructs as was done in ArchJava [25], or use an MDA-approach [26]. Such an approach directly connects architecture to implementation, improving consistency between the domains. However, this requires at least a change in the way of working of the implementation domain; it has to use a new language. This poses a barrier for implementing such an approach in practical settings.

6 Conclusions

In this paper we propose a conformance check framework (CCF) that systematically determines discrepancies between an intended architecture and the realised architecture. Illuminating these differences is a preparatory step for architecture-driven maintenance and evolution in which previously developed artefacts are

reused for reasons of efficiency. Our CCF is non-intrusive. It coordinates the interaction between the architecture and the implementation domain of expertise, while regarding them autonomously. It uses readily available, possibly tailored, technology for the actual implementation of the conformance check system (CCS).

The CCF combines a colloquial engineering model and the CCS. The engineering model defines two principal categories of views on a system: runtime and development views. Checking the conformance of views from both categories requires different type of approaches. The engineering model also defines the concepts of the CCS. Two domains of expertise that independently develop view-based reasoning frameworks and a common design-space conformance viewpoint. The CCS relies on a clear definition of the design-space viewpoint and the mappings from the architectural and implementation views to this common conformance viewpoint.

The design-space viewpoint captures checkable concepts, which are the consensus between verifying abstract properties of the architecture domain and emerging properties of the implementation domain. Possible discrepancies between the domains are revealed as mismatches between the derived design-space conformance views and the impact of a mismatch on either of the domains; the severity of a mismatch is identified as part of the transformation from a domain specific viewpoint to the design-space viewpoint. We gave examples of design-space viewpoints for two principal categories of views and their mappings from architecture and implementation artefacts to this design-space viewpoint. The case study we executed uses and configures XML technology. Although the results are promising we encountered intriguing research questions, such as to what extent we can include consistency checks in CCF and how to use parsing and logic reasoning technology to implement the CCS.

Acknowledgement

This work has been sponsored in part by the Moose/Merlin ITEA projects. We thank the anonymous reviewers for their valuable comments.

References

1. MOOSE: software engineering MethOdOlogieS for Embedded systems (2004)
2. MERLIN: Embedded systems engineering in collaboration (2005)
3. Graaf, B., Lormans, M., Toetenel, H.: Embedded software engineering: state of the practice. IEEE Software **20** (2003) 61–69
4. IEEE-1471: IEEE recommended practice for architectural description of software intensive systems. IEEE Std 1471–2000 (2000)
5. van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., Riva, C.: Symphony: View-driven software architecture reconstruction. In: Proc. IEEE/IFIP Working Conf. on Software Architecture (WICSA'04). (2004) 122–134
6. Kruchten, P.B.: The 4+1 view model of architecture. IEEE Software **12** (1995) 42–50

7. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison-Wesley (1999)
8. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures:Views and Beyond. Addison-Wesley (2002)
9. IABG: Das v-modell: Vorgehensmodell zur planung und durchführung von it-vorhaben (1997)
10. Murphy, G.C., Notkin, D., Sullivan, K.: Software reflexion models: bridging the gap between source and high-level models. In: SIGSOFT '95: Proc. of the Symp. on Foundations of Software Engineering. (1995) 18–28
11. Stevens, P.: On associations in the unified modelling language. In: Proc. of UML 2001. Volume 2185 of Lecture Notes in Computer Science. (2001)
12. Garlan, D., Monroe, R.T., Wile, D.: ACME: architectural description of component-based systems. In: Foundations of component-based systems. Cambridge University Press (2000) 47–67
13. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E.: Modeling software architectures in the unified modeling language. ACM Trans. Softw. Eng. Methodol. **11** (2002) 2–57
14. Garlan, D., Cheng, S.W., Kompanek, A.J.: Reconciling the needs of architectural description with object-modeling notations. Science of Computer Programming **44** (2002) 23–49
15. Greg J, B.: JavaML - an XML-based source code representation for Java programs. `http://www.cs.washington.edu/homes/gjb/JavaML/` (2000)
16. Al-Ekram, R., Kontogiannis, K.: An XML-based framework for language neutral program representation and generic analysis. In: Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR 2005). (2005) 42–51
17. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: Xlinkit: a consistency checking and smart link generation service. ACM Trans. Inter. Tech. **2** (2002) 151–185
18. Michael Marconi, C.N.: Clix language specification version 1.0. `http://www.clixml.org/clix/1.0/` (2004)
19. Nentwich, C., Emmerich, W., Finkelstein, A., Ellmer, E.: Flexible consistency checking. ACM Trans. Softw. Eng. Methodol. **12** (2003) 28–63
20. Dobrica, L., Niemelä, E.: A survey on software architecture analysis methods. IEEE Transactions on software Engineering **28** (2002) 638–653
21. Laitenberger, O., DeBaud, J.M.: An encompassing life cycle centric survey of software inspection. Journal System and Software **50** (2000) 5–31
22. Krikhaar, R.L.: Software architecture Reconstruction. PhD thesis, Universiteit van Amsterdam (1999)
23. Mens, K.: Automating Architectural Conformance Checking by means of Logic Meta Programming. PhD thesis, Vrije Universiteit Brussel (2002)
24. Davis, II, J., Hylands, C., Janneck, J., Lee, E.A., et al.: Overview of the ptolemy project. Technical Report UCB/ERL M01/11, University of California (2001)
25. ArchJava: Home. `www.archjava.org` (2005)
26. OMG: MDA. `www.omg.org/mda` (2005)