Maintaining Consistency across Engineering Artifacts

Alexander Egyed, Peter Hehenberger, Andreas Demuth, and Klaus Zeman

Abstract—Software and systems engineering combines artifacts from diverse engineering domains and tools. This article explores how incremental consistency checking is able to automatically and continuously detect inconsistencies among these engineering artifacts – even if those artifacts reside in different engineering tools and hence are handled by different engineers. We argue that engineers should be aware of inconsistencies, even if they are willing to tolerate them.

1 Introduction

Engineering is a collaborative effort that involves many stakeholders. Yet its day-to-day operations cater to the needs of individuals. Complex and multi-disciplinary problems are broken down to tasks that individual engineers are able to solve with the tools and methods available to them. Yet, this tool landscape is diverse and is getting increasingly more so. It is not uncommon that companies use dozens if not hundreds of tools - for example, software engineers use tools for requirements capture, specification, design/architecture, programming, or testing; tools that are often fundamentally different from those used by mechanical, electrical, aeronautical or systems engineers.

Not only do tools differ but so does the knowledge that engineers capture and maintain within these tools. We speak of artifacts. Examples of such artifacts are requirements, methods, hardware components, computations, use cases, electrical switches, and many more. Yet, while artifacts in tools are syntactically and semantically diverse, they are not independent. Together, these artifacts describe the engineering problem, process, and solution.

This paper describes how to detect inconsistencies across diverse artifacts. Inconsistencies are not limited to artifacts within single engineering tools (intra-tool consistency [1]) but may also happen across engineering tools (inter-tool consistency) where artifacts of two seemingly independent tools contradict. One might argue that inter-tool inconsistencies are more problematic than intra-tool consistencies because of diverse tool semantics and the concurrent modification of artifacts by different engineers. Nonetheless, detecting all inconsistencies is critical for success—not only in software engineering but across all engineering disciplines. Failure to detect such inconsistencies is known to lead to project failures, cost and schedule overruns, and suboptimal designs [2]. We argue that today it is not enough to discover inconsistencies eventually but engineers should be informed about them as early as possible—ideally shortly after they have been introduced. This is desired even if engineers are willing to tolerate inconsistencies for some time [3].

The remainder of the paper is organized as follows. In Section 2, we illustrate some typical examples of inconsistencies that arise in practice. Section 3 discusses how our approach to consistency checking works. It also provides an intra-tool consistency tool example. Section 4 then extends this to detecting inconsistencies across different tools. Applicability, scalability, and use cases are discussed in Section 5.

2 Inconsistencies Illustrated

Inconsistencies come in many shapes and sizes. To better illustrate them, let us discuss a few examples. We chose a mechatronical domain, in which a robot is to be developed. Figure 1 shows a 3D CAD drawing of the robot with its most significant hardware components and the UML use case diagram describes its basic uses. Even though the CAD drawing and the use case diagram follow a simple structure and notation (to be understandable across engineering disciplines), these diagrams can reach high levels of complexity. Consequently, many kinds of errors may exist within and among them. Take for example the requirement that every use case (ellipsoid) ought to be connected to an actor (stick man)—directly or indirectly via other use cases. Such a requirement would make sure that every use case can be triggered by an actor. Indeed, the use case diagram does violate this requirement because the use case "Move Object" is not used by any actor (i.e., there is no connection between the actor "Controller" and the use case "Move Object", not even via other use cases). Though this requirement is straightforward to explain, it is hard to enforce if actors and use cases are numerous and scattered across different diagrams.

Inconsistencies become significantly harder to detect if they span across artifacts of different tools. For example, consider the requirement that hardware/software components ought to be mapped to use cases that require them (a common requirement in safety critical domains [4]). If we consider the hardware components from Figure 1 then this requirement would make sure that every component

• *This research was supported by the Austrian Science Fund (FWF): P25289 N15, the Austrian COMET-K2 Programme of the Linz Center of Mechatronics (LCM), and a grant of the JKU Linz Institute of Technology (LIT).*

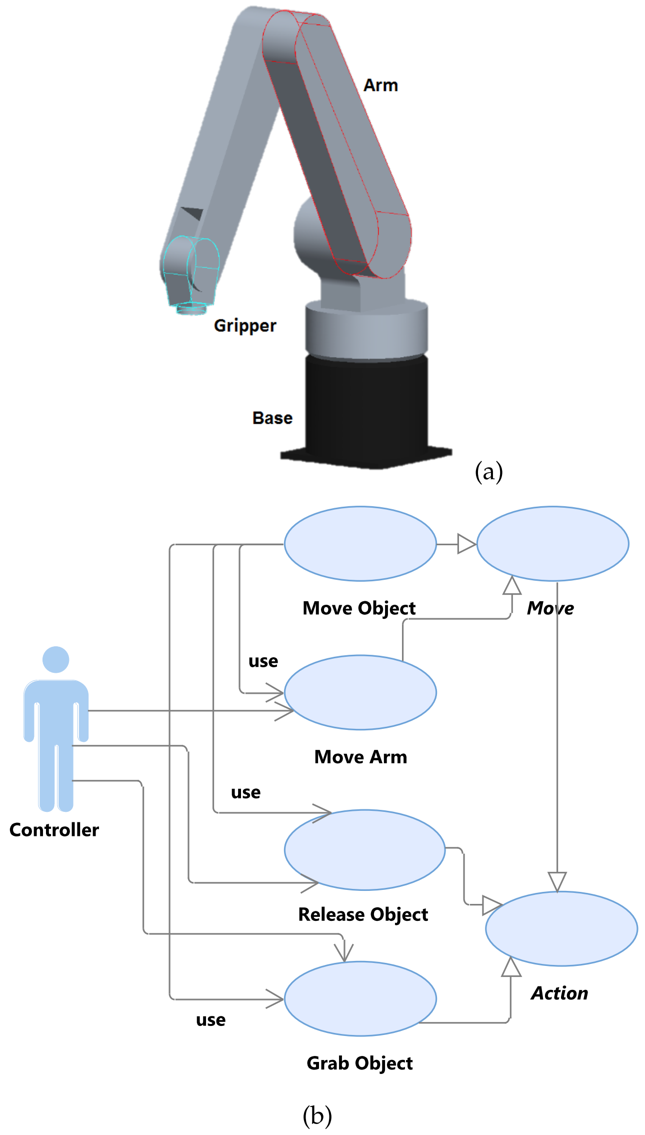


Fig. 1: Different Kinds of Engineering Artifacts of a Robot System: (a) CAD Drawing (b) Use Cases and Actor. How does one detect Inconsistencies among them if Engineers capture these Artifacts in different Tools?

identified there (e.g., `Gripper` and `Arm`) is used by at least one use case.

Conceptually, the ”every use case must be trigged by an actor” requirement is similar to the ”every component must be used by a use case” requirement. However, with the first requirement, all artifacts involved can be found within a single tool (the use case diagram). This is not the case with the second requirement. While the “Move Arm” use case likely relates to the `Arm` component in the drawing, it is much less obvious that the “Grab Object” relates to the `Gripper`. For detecting the second kind of inconsistencies, relations among use cases and CAD components need to be defined explicitly – for example, in form of traceability links [5] as seen in Figure 2. Without such explicit links [6], it is significantly harder if not impossible to detect inconsistencies among artifacts of different tools. Yet, these

		CAD Components		
		Gripper	Arm	Base
Use Case	*Move*		X	
	Action			
	Move Object	X	X	X
	Move Arm		X	X
	Release Object	X		
	Grab Object	X		

Fig. 2: Explicit Links between Use Cases and CAD Components are required for detecting Inconsistencies. How does one capture those Links if the Artifacts reside in different Tools?

inconsistencies are just as important—or perhaps even more so.

3 The Model/Analyzer Approach

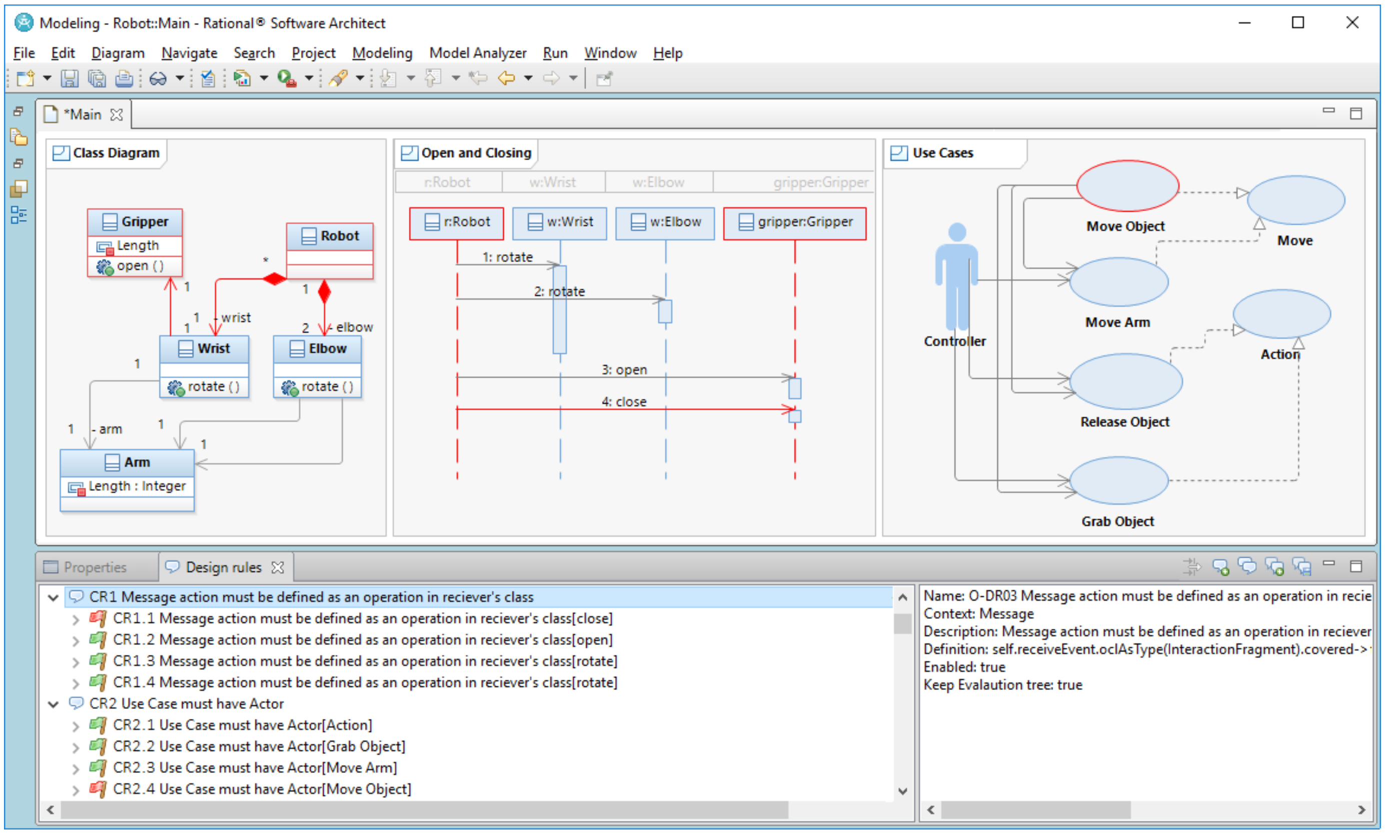
The Model/Analyzer approach is an incremental consistency checker [7] that focuses on changes (deltas) and provides engineers with immediate inconsistency feedback on the direct consequences of their actions. The engineers can then fix these inconsistencies or they can choose to ignore them [3], but do so intentionally instead of accidentally.

3.1 Intra-Consistency Example: UML Tool

Figure 3 depicts an intra-tool integration of the Model/-Analyzer approach with the IBM RSA modeling tool for the Unified Modeling Language (UML). We see a class diagram (top left), a sequence diagram (top middle), and the previously discussed use case diagram (top right). The inconsistency feedback is visualized at the bottom. All diagrams conform to the UML specification. However, they do violate other semantics that engineers typically want to enforce when using UML even if the UML tool does not. Besides the use case inconsistency discussed above, the example depicts another inconsistency with regard to message `5:close` in the sequence diagram. This message is sent from the robot controller to its gripper (see sender and receiver lifelines). However, the `Gripper` class does not actually define an operation called “close” in the class diagram. Hence, there is no operation corresponding to the message. The inconsistencies are visualized in red for easier identification.

3.2 Consistency Rule Language

Like other consistency checkers, the Model/Analyzer approach requires defined consistency rules. Such a rule embodies a condition (as a boolean expression among artifacts) with a context (a specific kind of artifact the condition applies to). We use the standard OCL language [8] for letting engineers define such rules. For example, the inconsistency where message `5:close` does not have a corresponding operation is based on the consistency rule `CR1` in Figure 4. This rule simply defines that for any given message there must be a corresponding operation in the message receiver’s class. In UML, a message receiver is a life line and its represented type is a class with operations. The rule states



CR1 Message action must be defined as an operation in reciever's class
CR1.1 Message action must be defined as an operation in reciever's class[close]
CR1.2 Message action must be defined as an operation in reciever's class[open]
CR1.3 Message action must be defined as an operation in reciever's class[rotate]
CR1.4 Message action must be defined as an operation in reciever's class[rotate]
CR2 Use Case must have Actor
CR2.1 Use Case must have Actor[Action]
CR2.2 Use Case must have Actor[Grab Object]
CR2.3 Use Case must have Actor[Move Arm]
CR2.4 Use Case must have Actor[Move Object]

Name: O-DR03 Message action must be defined as an operation in recie
Context: Message
Description: Message action must be defined as an operation in reciever
Definition: self.receiveEvent.oclAsType(InteractionFragment).covered->
Enabled: true
Keep Evalaution tree: true

Fig. 3: Intra-Tool Consistency Checking detects Inconsistencies among Artifacts within the same Tool (here UML Class, Sequence, and Use Case Diagrams). The Inconsistencies are flagged (in red) on the bottom left and the Artifacts involved are highlighted in the Diagrams.

that there must exist an operation whose name is equal to the name of the message. The context of this rule is an `UML::Message` which implies that this rule is applied to any instance thereof. One of these instances – the message `5:close` violates this rule – the other instances do not. Do note that identifiers such as ”receiveEvent” or ”type” are defined in the UML modeling language – the standardized UML meta model.

The inconsistency where the use case `Move Object` does not have a connected actor is based on another rule. Rule `CR2` in Figure 4 defines that for a given `UML::UseCase` there must exist an association whose end type is an Actor.

3.3 Incremental Validation and Scopes

A key scalability aspect of the Model/Analyzer is that a given consistency rule is validated separately for every instance of a rule's context. In our example in Figure 3, the rule `CR1` is validated four separate times – once for every message as depicted on the bottom left (notice the four flags underneath the consistency rule) Likewise, the rule `CR2` is validated six separate times – once for every use case (the flags of four of those evaluations are depicted in the figure). We refer to a single validation of a consistency rule on a specific artifact as a *consistency rule instance (CRI)*. A CRI is uniquely identified by the consistency rule it validates and the specific artifact (*context element*) it applies to. For example, the CRI `<CR1,close[UML::Message]>` validates whether the message `close` in Figure 3 has a corresponding operation defined in the receiver class `Gripper`. This is one of the four evaluations visible in the figure and the only one flagged inconsistent (red flag) because of its inconsistent state. Note that we use a simplified notation in this article to identify artifacts by their name followed by their type in brackets for readability reasons. In reality, artifacts are identified by unique ids.

CRIs do provide two major benefits. First, rules are written from the viewpoint of individual artifacts (e.g., for a specific `UML::Message`). Second, CRIs provide fine-grained feedback about the consistency of specific artifacts. Each CRI is validated separately and the consistency of each CRI is determined on an individual basis. Therefore, the impact of a change onto consistency is often very cheap to compute and engineers will be informed about the impact of the changes they are currently working on only.

Traditional approaches require all design rules to be validated every time a change occurs because they do not understand the impact of a change onto consistency rules. In the Model/Analyzer, each CRI automatically maintains its own change impact *scope* which is computed during a CRI's validation. A typical validation consists of navigating through artifacts and computing a boolean result. For example, during the validation of the CRI `<CR1,close[UML::Message]>`, the `name` attribute of the object `close[UML::Message]` is read, the path to the message receiver's class is walked through, the names of

```
Consistency Rule CR1:
context UML::Message:
self.receiveEvent.covered->forAll(l:Lifeline |
l.represents.type.ownedOperation->
     exists(o:Operation | o.name = self.name))
Consistency Rule CR2
context UML::UseCase:
self.getAssociations()->exists(a | a.endType->
     exists(t | t.oclIsTypeOf(Actor)))
```

Fig. 4: Two Consistency Rules written in OCL. For example, CR2 ensures that each use case is connected to an actor who needs it.

this class' operations are collected, and finally the collection is searched for operations with a matching message name – an equality computation which returns "false" because no operation exists that matches the name "close". The CRI's scope is simply the union of all artifacts visited during this validation. This scope is observable automatically and all that is needed to enable correct and complete incremental consistency checking. The basic observation is that a CRI's consistency state can only be affected if an artifact in the CRI's scope changes. Hence only those CRIs whose scopes contain a changed artifact need to be re-validated. In essence, Figure 3 depicts the scope elements of all inconsistent CRIs (e.g., `<CR1,close[UML::Message]>`) in red which is not only a pre-requisite for scalability but also a useful visualization.

4 Consistency Checking across Tools

There is no technical reason why every engineering tool could not offer its own built-in consistency checking mechanism (analogous to Figure 3). However, doing so would not provide a comprehensive solution because any given tool typically contains a subset of artifacts only and hence cannot support comprehensive consistency checking. For example, a programming tool does not contain design modeling artifacts or requirements and, consequently, a programming tool could not detect inconsistencies with those design artifacts or requirements. A cross-tool consistency checking mechanism provides a single, uniform solution for all tools and it would be usable and understandable by all engineers because it would be based on a common consistency rule language and an uniform user interface. This is especially important in situations where consistency rules span across engineering tools and engineers need to work together to write and to interpret them.

To enable inter-tool consistency checking, two problems need to be solved for the Model/Analyzer approach to function: 1) the artifacts need to be transformed to a common representation and 2) the artifacts need to be linked to allow consistency rules to navigate them (i.e., to identify the CAD components that are meant to implement the use cases). We have explored two possible strategies which are discussed next:

4.1 Unifying Tool

A simple strategy is to use an unifying tool that represents artifacts from various tools. Transformation methods are then used to translate the various tool artifacts to the

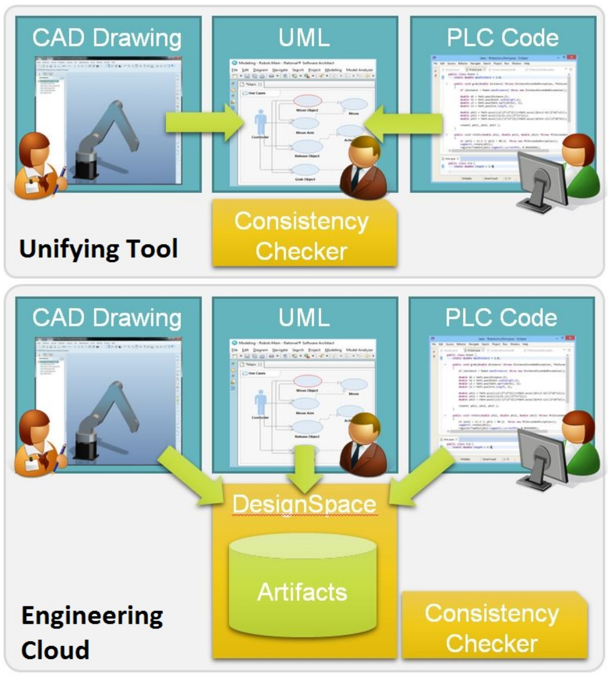


Fig. 5: Inter-Tool Consistency Checking is possible through the use of an Unifying Tool (e.g., an UML Tool at the top) or an Engineering Cloud (e.g., DesignSpace at the bottom) which provide uniform, tool-independent representations for Artifacts.

representation of the unifying tool (Figure 5 top). Within the unifying tool it is possible to add links among these artifacts, if needed. Once artifacts are represented and linked in unifying tools then inter-tool consistency checking is very similar to intra-tool consistency checking since all artifacts reside within that same tool.

We commonly (mis)use UML tools for this purpose, taking advantage of the UML extensibility mechanism through stereotyping. Figure 6 (a) again depicts the IBM RSA modeling tool but now it also represents the main CAD components and the links between these components and the use cases. To realize this, we merely need to transform the CAD drawing from Figure 1 to UML (this transformation is discussed later). While UML does not have built-in constructs for arbitrary mechatronical artifacts, existing UML artifacts can be (mis)used to represent them. In our example, we merely require the existence of the CAD components and represent them as UML components (note the UML stereotype `component`). Engineers may then add links between the CAD components (aka UML components) and use cases through UML trace associations – an existing linking mechanism that is now (mis)used to represent CAD to use case links. These links essentially implement the relations defined in Figure*2. For example, we previously learned that the `Gripper` component is necessary to perform the use cases "Release Object", "Grab Object", and "Move Object". In Figure 6 (a) we thus find three corresponding UML trace links. The consistency rule then ensures that such links do in fact exist.

A drawback of this strategy is that only those engineers benefit from consistency feedback who are using the unifying tools. Moreover, the unifying tool must be in-use continuously for consistency checking to function. Hence, this strategy is mostly only useful if a single engineer uses multiple tools and likes to maintain the consistency among them. Moreover, one also needs to pay attention to the writing of consistency rules since they reflect on UML elements as placeholders for other tools' artifacts.

4.2 Engineering Cloud

For multi-user consistency checking, a better strategy is the use of an engineering cloud such as the DesignSpace [9]. The cloud provides a common representation for all tools – typically on a server. The various tools sync their artifacts to this common representation using transformations that are similar to the ones discussed above (and described later). The consistency is validated in the cloud since all artifacts are represented there. Figure 6 (b) depicts the DesignSpace Engineering Cloud [9]. The left side shows artifacts that were synced to the cloud (e.g., the "Use Cases" and "CAD Components"). Engineers can further augment these artifacts. In this case, a "Link" type was added to allow links between CAD Components and Use Cases. One such link is visualized at the bottom left. This kind of links are thus another implementation of the relations defined in Figure*2.

Inter and Intra-Consistency checking is done in the cloud. For example, we are now able to handle the aforementioned consistency rule that ensures that any given CAD component must be used by at least one use case. This consistency rule is depicted in the top right of Figure 6 (b). It starts its validation at the CAD component (the context element) and through the "links" field (the one we added to the cloud) it navigates to the target elements, of which one is expected to be of type "Use Case".

It is more elaborate to realize this strategy than compared to the unifying tool because it requires a dedicated cloud/server. Moreover, all tool artifacts required for consistency checking need to be represented there - complete with meta model definitions and transformations. Yet, it provides comprehensive, multi-tool consistency checking with consistency rules that are quite straightforward to define.

5 Application

5.1 Transformation of Artifacts

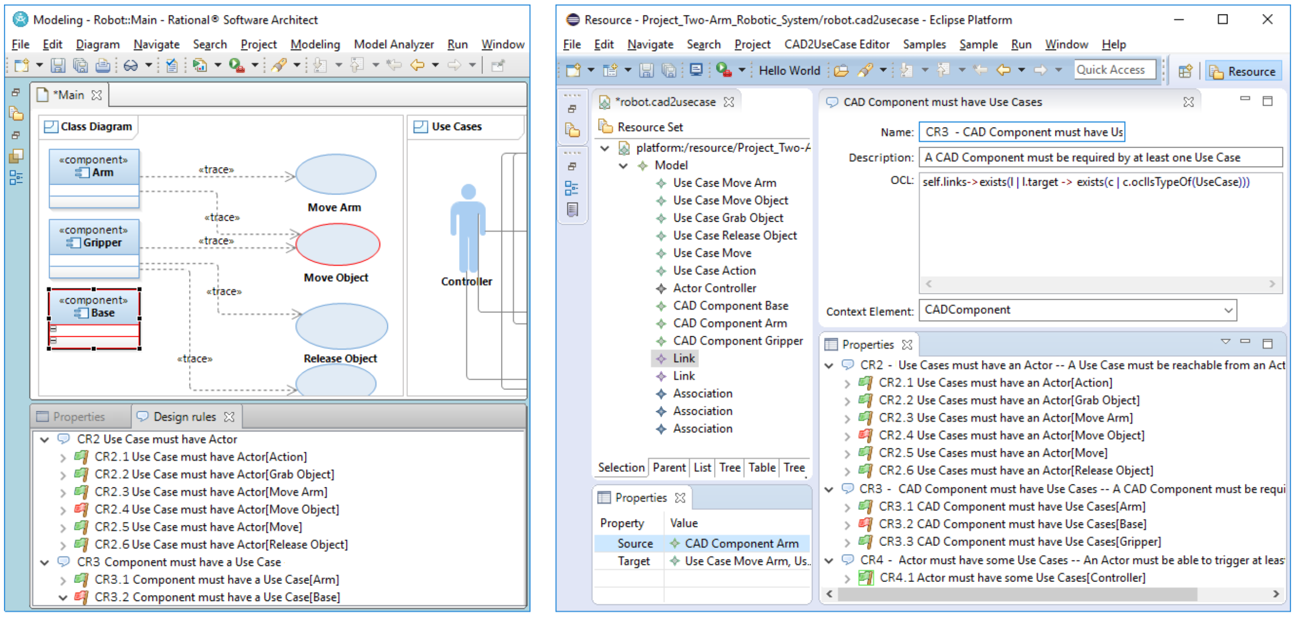
The unifying tool and the engineering cloud shared a common goal: the uniform representation of all artifacts needed for consistency checking. With such an uniform representation, the Model/Analyzer functions much like it would for a standalone tool (e.g., like with the IBM RSA modeling tool in Figure 3). However, a key problem is how to transform artifacts to this uniform representation. For this purpose, we build tool adapters that are plugged into their respective engineering tools. It is important to note that the actual adapter implementation is tool specific and may vary significantly from tool to tool. Nonetheless, there are common principles [10]. The effort of building such adapters depends on the openness of tools. Fortunately, most engineering tools do provide programmatic APIs for accessing its artifacts and in the past we typically had to invest a few person months of programming effort for each such adapter. Once implemented, the adapters functioned automatically by propagating artifacts and their changes to the unifying tool or engineering cloud. Examples of adapters we have built are Eclipse/Eclipse Modeling Framework (Java and Domain Specific Languages), EPlan (Electrical Drawings), IBM Rational Software Architect (UML), LibrePlan (Requirements and Projekt Management), Microsoft Excel (Calculations), or PTC Pro/Engineer (CAD Drawings).

The key question was knowing which artifacts the adapters had to propagate. Not every detail was relevant for consistency checking and we generally found it useful to simplify artifact representation with transformation. For example, Figure 6 merely depicted the existence of CAD components but not their many graphical properties because these properties were not relevant for consistency checking. Such simplifications greatly limited the complexity and cost of building tool adapters.

We generally relied on domain experts to define the consistency rules and to decide what artifacts were needed for evaluating them. These domain experts also helped us understand how to visualize inconsistencies once detected. for the most part, we enumerated inconsistencies and highlighted the artifacts involved (i.e., the scope). This was sufficient if engineers understood well the meaning of the consistency rules. Future work will investigate other visualizations.

5.2 Linking of Artifacts

Consistency checking investigates relationships among artifacts. For this purpose, it is necessary to navigate among artifacts as was discussed above. This is challenging when artifacts reside in different tools. Tool vendors generally do not integrate their tools. Even the publishing of artifacts into shared repositories (e.g., GIT, SVN, Engineering Clouds) does not remedy this problem because there these artifacts merely co-exist but remain unlinked. For example, we encountered this problem when we needed to describe the relationships among the robot's use cases and CAD components. Neither the CAD tool nor the Use Case modeling tool provided means of capturing such relationships. Only when we provided an uniform representation were we able to add links among them. If two tools use semantically similar artifacts with a consistent naming scheme then such links might be implicit and could be derived automatically. However, the CAD/Use Case example also showed the more common situation where this is not possible. There is little naming similarity among the artifacts in Figure 1. To remedy this,



CR2 Use Case must have Actor
- CR2.1 Use Case must have Actor[Action]
- CR2.2 Use Case must have Actor[Grab Object]
- CR2.3 Use Case must have Actor[Move Arm]
- CR2.4 Use Case must have Actor[Move Object]
- CR2.5 Use Case must have Actor[Move]
- CR2.6 Use Case must have Actor[Release Object]

CR3 Component must have a Use Case
- CR3.1 Component must have a Use Case[Arm]
- CR3.2 Component must have a Use Case[Base]

Name: CR3 - CAD Component must have Us

Description: A CAD Component must be required by at least one Use Case

OCL: `self.links->exists(l | l.target -> exists(c | c.oclIsTypeOf(UseCase)))`

Context Element: CADComponent

Property	Value
Source	CAD Component Arm
Target	Use Case Move Arm, Us..

CR2 - Use Cases must have an Actor -- A Use Case must be reachable from an Act
- CR2.1 Use Cases must have an Actor[Action]
- CR2.2 Use Cases must have an Actor[Grab Object]
- CR2.3 Use Cases must have an Actor[Move Arm]
- CR2.4 Use Cases must have an Actor[Move Object]
- CR2.5 Use Cases must have an Actor[Move]
- CR2.6 Use Cases must have an Actor[Release Object]

CR3 - CAD Component must have Use Cases -- A CAD Component must be requi
- CR3.1 CAD Component must have Use Cases[Arm]
- CR3.2 CAD Component must have Use Cases[Base]
- CR3.3 CAD Component must have Use Cases[Gripper]

CR4 - Actor must have some Use Cases -- An Actor must be able to trigger at leas
- CR4.1 Actor must have some Use Cases[Controller]

a) UML Unifying Tool **b) DesignSpace Engineering Cloud**

Fig. 6: Two Inter-Tool Consistency Checking Examples: a) Unifying UML Modeling Tool Example which represents CAD Components as UML Components with CAD-UseCase Links as UML trace associations or b) DesignSpace Engineering Cloud Example which represents relevant Artifacts from both CAD Tool and UML Modeling Tool with CAD-UseCase Links as explicit Link Types.

we provided special purpose user interfaces to let engineers define links among artifacts manually. Examples of such interfaces were shown in Figure 6.

5.3 Scalability and Case Studies

The Model/Analyzer provides consistency checking at a much reduced computational cost because the approach focuses on the changes made by engineers. We found that the approach is very scalable—in previous studies we applied it to models with more than 100,000 artifacts and observed average evaluation times around 1ms per change [7].

In terms of applications, we implemented incremental consistency checking for standalone tools, unifying tools, and engineering clouds. Moreover, we have conducted case studies that cover both inter- and intra- tool consistency checking scenarios on a variety of artifacts [6], [9], [11], [12]. Specifically, the approach was applied to consistency checking within UML, model-to-code, electrical-drawing-to-code, metamodel-to-model, and others.

The approach has even been augmented with repair suggestions [13] as it might be complicated to precisely determine the cause of an inconsistency – especially if artifacts from multiple tools are involved. Combined with repairs, the approach even supports basic change impact analysis because inconsistencies caused by a change reflect the failure to have propagated that change to other artifacts, tools, or engineering domains [6].

While the Model/Analyzer uses OCL for writing consistency rule, few restrictions exist on the language and complexity of such rules. Existing work has demonstrated that incremental consistency rules can scale to rather complex situations and such rules can be structural (e.g., the existence of a link between the use case and the actor) or behavioral (e.g., to ensure that artifact changes happen in a certain order [14]).

6 Conclusion

The Model/Analyzer approach to incremental consistency checking provides up to date assessment on the consistency, correctness, and even completeness of an engineering project. The time frame of the feedback is freely definable—from instantaneous inconsistency feedback with every single change to periodic, user-definable intervals (when saved, when committed to a repository, once a day, etc.). Its passive feedback ensures that inconsistency feedback is available on demand but it does not interrupt the engineers' workflow. The approach detects inconsistencies both within and across tools. Its scalability and applicablity were evaluated through a wide range of case studies.

References

[1] F. J. Lucas, F. Molina, and J. A. T. Álvarez, "A systematic review of UML model consistency management," *Information & Software Technology*, vol. 51, no. 12, pp. 1631–1645, 2009.

[2] B. W. Boehm, B. Clark, E. Horowitz, J. C. Westland, R. J. Madachy, and R. W. Selby, "Cost models for future software life cycle processes: COCOMO 2.0," *Annals of Software Engineering*, vol. 1, pp. 57–94, 1995.

[3] R. Balzer, "Tolerating inconsistency," in *Proceedings of the International Conference on Software Engineering (ICSE), Austin, Texas, USA*, pp. 158–165, 1991.

[4] B. S. Andersen and G. Romanski, "Verification of safety-critical software," *Communications of the ACM*, vol. 54, no. 10, pp. 52–57, 2011.

[5] A. Ghabi and A. Egyed, "Exploiting traceability uncertainty among artifacts and code," *Journal of Systems and Software*, vol. 108, pp. 178–192, 2015.

[6] A. Demuth, R. Kretschmer, A. Egyed, and D. Maes, "Introducing traceability and consistency checking for change impact analysis across engineering tools in an automation solution company: An experience report," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME), Raleigh, North Carolina, USA*, 2016.

[7] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 188–204, 2011.

[8] Object Management Group, "Object Constraint Language (OCL)." http://www.omg.org/spec/OCL/, 2014.

[9] A. Demuth, M. Riedl-Ehrenleitner, A. Nöhrer, P. Hehenberger, K. Zeman, and A. Egyed, "DesignSpace – An Infrastructure for Multi-User/Multi-Tool Engineering," in *Proceedings of the Annual ACM Symposium on Applied Computing, Salamanca, Spain*, pp. 1486–1491, 2015.

[10] D. S. Wile, R. Balzer, N. M. Goldman, M. Tallis, A. Egyed, and T. Hollebeek, "Adapting cots products," in *Proceedings of the International Conference on Software Maintenance (ICSM), Timisoara, Romania*, pp. 1–9, 2010.

[11] M. Riedl-Ehrenleitner, A. Demuth, and A. Egyed, "Towards model-and-code consistency checking," in *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC), Vasteras, Sweden*, pp. 85–90, 2014.

[12] A. Demuth, M. Riedl-Ehrenleitner, R. Kretschmer, and A. Egyed, "Towards efficient risk-identification in risk-driven development processes," in *Proceedings of the International Conference on Software and Systems Process (ICSSP), Austin, Texas, USA*, pp. 36–40, 2016.

[13] A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *International Conference on Automated Software Engineering (ASE), Essen, Germany*, pp. 220–229, 2012.

[14] M. Vierhauser, R. Rabiser, P. Grünbacher, K. Seyerlehner, S. Wallner, and H. Zeisel, "Reminds : A flexible runtime monitoring framework for systems of systems," *Journal of Systems and Software*, vol. 112, pp. 123–136, 2016.

Alexander Egyed is Professor for Software-Intensive Systems at the Johannes Kepler University (JKU), Linz, Austria. He received his Doctorate degree from the University of Southern California, USA and previously worked for industry for many years. Dr. Egyed was named an IBM Research Faculty Fellow and was recognized as a leading software engineering scholar in the Communications of the ACM and the Springer Scientometrics. He is most recognized for this work on model-driven software and systems engineering – particularly on variability, engineering methods, consistency and traceability. He is a senior member of ACM and IEEE.

Peter Hehenberger is Professor for Integrated Product Development at the School of Engineering, University of Applied Sciences Upper Austria, Austria. He received his Doctorate Degree from the Johannes Kepler University (JKU) and previously was also an Assistant Professor there. His core competencies cover model-based mechatronic system design. He is member of IFAC TC4.1 Mechatronics, IFIP WG5.1 and Design Society, where he co-chairs a Special Interest Group (SIG) on Methodologies for Design, Integration, Modelling and Simulation of Cyber Physical Systems.

Andreas Demuth was Post Doc at the Johannes Kepler University (JKU) Linz, Austria. He received his Doctorate degree from the Johannes Kepler University. His research work is centered on model transformation and constraint-driven software engineering.

Klaus Zeman is Professor for Mecharonic Design and Production at the Johannes Kepler University (JKU) Linz, Austria. He received his Doctorate Degree from the Vienna Technical University, Austria. Afterwards he gained 12 years of industrial experience as senior manager responsible for research and development in rolling mills and strip processing lines. His current research focuses on modeling and simulation of metal forming processes, machine dynamics, model based design of mechatronic systems, and design science. Klaus Zeman is member of WiGeP and the Scientific Forum for Product Development.