Incremental Consistency Checking of Dynamic Constraints

Iris Groher, Alexander Reder, and Alexander Egyed

Johannes Kepler University
Altenbergerstr. 69, 4040 Linz, Austria
{iris.groher,alexander.reder,alexander.egyed}@jku.at

Abstract. Software design models are routinely adapted to domains, companies, and applications. This requires customizable consistency checkers that allow engineers to dynamically adapt model constraints. To benefit from quick design feedback, such consistency checkers should evaluate the consistency of such changeable constraints incrementally with design changes. This paper presents such a freely customizable, incremental consistency checker. We demonstrate that constraints can be defined and re-defined at will. And we demonstrate that its performance is instant for many kinds of constraints without manual annotations or restrictions on the constraint language used. Our approach supports both model and meta-model constraints and was evaluated on over 20 software models and 24 types of constraints. It is fully automated and integrated into the IBM Rational Software Modeler tool.

Keywords: consistency checking, dynamic constraints, incremental checking.

1 Introduction

Design constraints are an important means of evaluating the correctness (consistency) of a model. While it is acceptable to tolerate design errors [1], engineers should be aware of them to avoid follow-on errors – or risk having to revisit and fix the follow-on errors at a later time. Violations of design constraints should thus be detected quickly, preferably instantly, and continuously tracked throughout the software life cycle – ideally in a non-intrusive manner that does not obstruct the natural workflow of the engineer.

This stands in stark contrast to the often individualistic nature in which modeling languages are used. Today, it is common practice to adapt modeling languages to specific domains, companies, and even applications under development. The benefits range from increased utility to better automation. Design constraints are not immune to this push to individualism. It implies that *engineers must be allowed to define new or adapt existing design constraints at will* – ideally without having to know the internals of the consistency checker. Even more importantly, feedback on design correctness should be provided incrementally with model changes without any manual overhead (since the learning curve would hinder its adoption) or observable computation delay (since noticeable delays obstruct the engineers' work flow).

D.S. Rosenblum and G. Taentzer (Eds.): FASE 2010, LNCS 6013, pp. 203–217, 2010.

Unfortunately, few existing approaches to consistency checking are readily extendable to allow the incremental, on-the-fly definition of new constraints or the customization of existing constraints without having to restart the consistency checker. The few approaches that support the addition of new constraints are either not incremental or require the engineers to manually annotate constraints in ways that are beyond their ability – a manual, error-prone process that provides some performance benefits but fails to scale for large design models [2], severely restricts the expressiveness of constraints via a limited constraint language [3] or requires the engineer to manually re-write a constraint for as many times as there are model changes affecting it [4]. Also, many existing approaches are typically tied to a specific modeling language and/or constraint language.

This paper presents an approach to the incremental consistency checking of dynamically definable and modifiable design constraints. Engineers can define constraints in a language of their choice (e.g. we have done so for Java, C#, and OCL [5]) and for any modeling language of their choice (e.g. we have used UML 1.3, UML 2.1, Matlab/Stateflow, and a domain specific language [6]). Our approach works for both model and meta-model constraints, neither of which must be manually annotated or rewritten. In which can be defined, redefined, or deleted at will throughout the development life cycle. Meta model constraints refer to constraints that hold for all instances of a certain type of model element. For example, in a home automation system, we could define a meta constraint that every light has to be connected to at least one light switch. Model constraints, on the other hand, refer to specific model elements. As such, we could add a constraint for a specific light to have at least two such switches.

We observed that engineers are willing and capable of defining both meta model and model constraints but we also observed that it is not reasonable to assume that (1) engineers are capable of defining how incremental changes affect such constraints and (2) all such constraints are known ahead time. Rather, they are discovered incrementally during modeling and the engineer should be able to add or change them as necessary. For example, in the middle of the design, an engineer could introduce the model-view-controller pattern [7] into our home automation system and desire to automatically enforce the constraints associated with that pattern. We could also change the constraint that every light has to be connected to a light switch in a way that a light either has to be connected to a light switch or to a motion sensor.

Today constraint changes typically require the complete re-evaluation of a design model which is equivalent to restarting of the constraint checker. We will see that our approach is capable of keeping up with the engineer in real time *for both constraint and model changes* for *scalable constraints*. Scalable constrains refer to constraints that are local, i.e. their evaluation does not require traversing large parts of the model. For scalable constraints our approach performs much better than existing approaches and for non-scalable constraints our approach performs no worse. Our approach is fully tool supported and integrated with the modeling tool IBM Rational Software Modeler [8]. The computational efficiency and scalability of our approach and tool were evaluated through the empirical analysis or 20 industrial software models and 24 different constraints.

2 Illustrative Example

In the following, we illustrate our approach on a simple home automation system [9]. In homes there are a wide range of electrical and electronic devices such as lights, thermostats, electric blinds, fire and smoke detection sensors, white goods such as washing machines, as well as entertainment equipment. The home automation system connects those devices and enables inhabitants to monitor and control them from a single GUI. The home network also allows the devices to coordinate their behavior in order to fulfill complex tasks without human intervention.

Fig. 2 presents a simple structural model of a home automation system. The *MyHouse* building consists of two floors, *Cellar* and *GroundFloor*. Two rooms, *WorkRoom* and *LivingRoom*, are located on the floors, each containing a light. The lights are connected to a light switch and to a motion sensor respectively. The sequence diagram in Fig. 3 describes the process of turning on the light in the work room. The user first gets a list of available devices in the room and presses the light switch. The light switch invokes the *turnOn* method on the light object. Stereotypes denote the different devices present in the house. Stereotypes are a common way of adding domain-specific extensions to UML.

Fig. 3 describes four sample constraints (using an OCL-like syntax) for the home automation system model. Constraint C1 is a standard UML consistency rule, constraints C2 and C3 are domain-specific meta model constraints, and C4 is an application-specific model constraint (we will discuss later the difference between application and domain/meta model constraints). C1 describes how UML sequence diagrams relate to UML class diagrams. It states that the name of a message must match a method in the receiver's class. The constraint is a general-purpose meta model constraint because it holds for all messages in sequence diagrams across all UML models. If the constraint is evaluated on the 2nd message in the sequence diagram in Fig. 3 (the *press* message) then the condition first computes the set of methods defined in the base class of the receiver object. The receiver object is *lightSwitch* and its base class is *LightSwitch*. The set of methods defined in the *LightSwitch* class is {*press()*}. The condition returns true because the set of methods contains a method with the name equal the message name *press*.

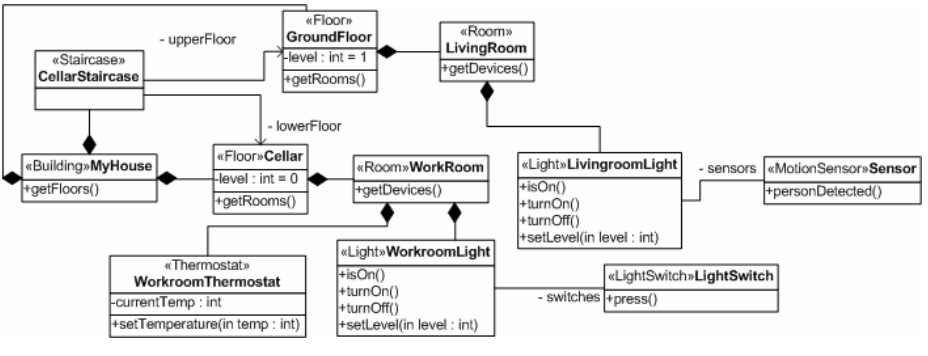

Fig. 1. Home automation system model

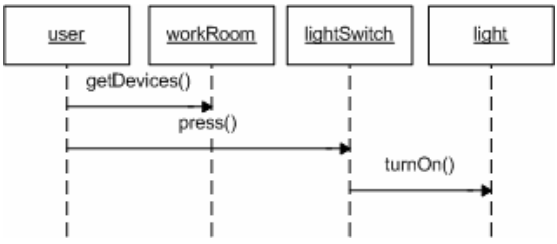

Fig. 2. Sequence diagram – User turns the work room light on

C2 and C3 describe domain-specific, meta model constraints. C2 ensures that every light is connected to at least one light switch. Clearly, this constraint no longer applies to any UML model. The condition computes the number of the switches attached to a light which must be greater than zero. C3 ensures that staircases only connect floors at neighboring levels. It compares the level number of the upper floor to the level number of the lower floor. Finally, C4 describes a domain-specific, model level constraint. It holds only for the model element *WorkroomThermostat* defined in the model in Fig.2. The constraint ensures that the current temperature is above 5 degrees *for that room only*.

C1	**Name of message must match an operation in receiver's class** methods = message.receiver.base.methods return (methods->name->contains(message.name))
C2	**Every light must at least be connected to one light switch** return light.switches.size > 0
C3	**Staircase must only connect floors at neighboring levels** return staircase.upperFloor.level = staircase.lowerFloor.level + 1
C4	**Thermostat in the work room must always be above 5 degrees** return workroomThermostat.currentTemp > 5

Fig. 3. Sample constraints

A constraint is typically defined from a particular point of view – a context element – to ease its design and maintenance [10]. A **constraint** is thus the tuple <context element, condition>. The *context element* defines for what model element a constraint applies. The condition is a statement that, evaluated on the context element, returns true if consistent or else false. Meta model constraints (C1-C3) define types of model elements as context elements (e.g., a Message) whereas model constraints (C4) define specific model elements as context elements (the class *WorkroomThermostat*).

The constraints in Fig.4 are merely a small sample of constraints that arise during the modeling of that kind of system. In summary, it is important to note that some constraints are generic (e.g., C1) whereas others only apply to a particular domain and/or application (e.g., C2). The former can be built into design tools but the latter not (*these have to be user definable!*). It is also important to note that some constraints are written from the perspective of the meta model (C1, C2, and C3) while others are written from the perspective of the application model (C4). With this

freedom to define constraints arbitrarily, it is obvious that incremental consistency checking not only has to deal with model changes but also constraint changes (C2-C4 could change at any time). Next, we discuss how our approach is able to do both.

3 Dynamic Constraints

3.1 Background

In previous work, we have demonstrated that our approach provides instant design feedback for ***changeable models but non-changeable constraints*** without requiring manual annotations of any kind [2, 13]. The instant checking of meta model constraints was achieved by observing the consistency checker to see what model elements it accessed during the validation of a constraint. To that extend, we built a *model profiler* to monitor the interaction between the consistency checker and the modeling tool.

It is important to note that our approach treats every evaluation of a meta model constraint separately. We thus distinguish between a ***constraint*** and its ***instances***. The constraint defines the condition (Fig. 4) and its context. It is instantiated for every model element it must evaluate (the ones identified through the context). For example, the meta model constraint that checks whether a light is connected to at least one light switch is instantiated for every light in the model – and each instance checks the validity of the constraint for its light only. For the house model presented in Fig.2 with its two light switches, our approach thus maintains two constraint instances, one for *WorkroomLight* and one for *LivingroomLight*. All instances are evaluated separately as they may differ in their findings. The instance evaluated for *LivingroomLight* is currently inconsistent because it is attached to a motion sensor instead of a light switch.

The role of the model profiler is thus to observe which model elements are accessed by which constraint instances. The model elements accessed by a constraint instance during its evaluation are referred to as the *scope* of a constraint instance. Only these model elements are relevant for computing the truth value of the constraint instance. And, more importantly, only changes to these model elements can trigger the re-evaluation of its constraint instance. Since the scope is maintained separately for every constraint instance, we are thus able to precisely identify what constraint instances to re-evaluate on what context elements ***when the model changes***. In [10], we showed that our scope is complete in that it contains at least the model elements that affect a constraint instance's truth value. It is not necessarily minimal in that it may contain more elements than needed - thus also causing some, but fairly few unnecessary re-evaluations of constraint instances.

Both the monitoring of constraint instances during constraint checking and the deciding what constraints to re-evaluate are done fully automatically. Since our approach never analyzes the constraints, any constraint language can be used. This gives the engineers considerable freedom in how to write constraints. Furthermore, since our approach does not require constraints to be annotated, this greatly simplifies the writing of constraints.

3.2 Contribution of This Work

The research community at large has focused on a limited form of consistency checking by assuming that only the model but not the constraints change (the latter are pre-defined and existing approaches typically require a complete, exhaustive re-evaluation of the entire model if a constraint changes!). ***The focus of this work is on how to support dynamically changeable constraints*** – that is constraints that may be added, removed, or modified at will *without losing the ability for instant, incremental consistency checking and without requiring any additional, manual annotations*. Such dynamic constraints arise naturally in many domain-specific contexts (cf. the example in the home automation domain described in Section 2). In addition to meta model constraints, this work also covers application-specific model constraints that are written from the perspective of a concrete model at hand (rather than the more generic meta model). We will demonstrate that model constraints can be directly embedded in the model and still be instantly and incrementally evaluated together with meta model constraints based on the same mechanism. For dynamic constraints, any constraint language should be usable. We demonstrate that our approach is usable with traditional kinds of constraint languages (e.g., OCL [5]) and even standard programming languages (Java or C#). Furthermore, our approach is independent of the modeling language used. We implemented our approach for UML 1.3, UML 2.1, Matlab/Stateflow and a modeling language for software product lines [6].

3.3 Meta Model and Model Constraints (+ Their Instances)

Fig. 4 illustrates the relationships between the meta model/model constraints and their instances.

Constraint = < condition, context element>
Meta Model Constraint: context element is element of meta model
Model Constraint: context element is element of model

Meta model constraints are written from the perspective of a meta model element. Many such constraints may exist in a meta model. Their conditions are written using the vocabulary of the meta model and their context elements are elements of the meta model. For example, the context element of constraint C1 in Fig. 3 is a UML Message (a meta model element). This implies that this constraint must be evaluated for every instance of a Message in a given model. In Fig.3 there are three such messages. Model constraints, on the other hand, are written from the perspective of a model element (an instance of a meta model element). Hence, its context element is a model element. For example, C4 in Fig. 3 applies to the *WorkroomThermostat* only – a specific model element.

Fig. 4 shows that for every meta model constraint a number of constraint instances are instantiated (top right) – one for each instance of the meta model element the context element refers to. On the other hand, a model constraint is instantiated exactly once – for the model element it defines.

Constraint Instance = <constraint, model element >

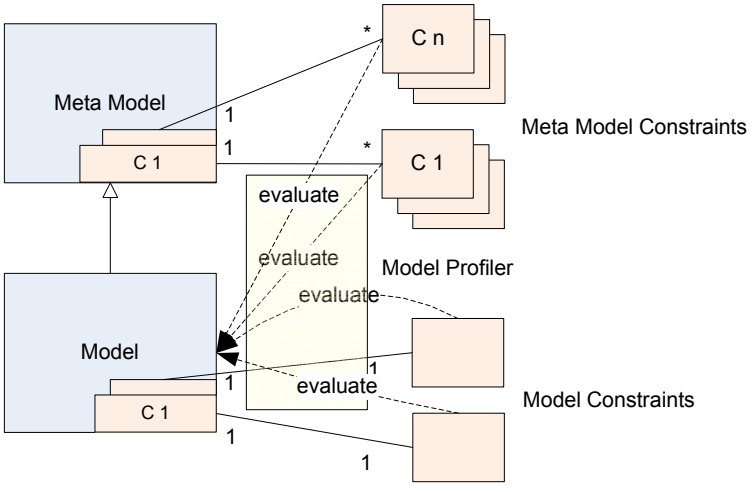

Fig. 4. Relationship between meta model and model constraint definitions and constraints

While the context elements differ for model and meta model constraints, their instances are alike: the instances of meta model constraints and the instances of model constraints have model elements as their context element. The only difference is that a meta model constraint results in many instances whereas a model constraint results in exactly one instance. Since the instances of both kinds of constraints are alike, our approach treats them in the same manner. Consequently, the core of our approach, the model profiler with its scope elements and re-evaluation mechanism discussed above, functions identical for both meta model constraints and model constraints as is illustrated in Fig. 4. The only difference is in how constraints must be instantiated. This is discussed further below in more detail.

As discussed above, we support the definition of both meta model and model constraints in Java, C#, and OCL. These languages are vastly different but our approach is oblivious of these differences because it cares only about a constraint's evaluation behavior and not its definition. The key to our approach is thus in the model profiling which happens during the evaluation of a constraint. During the evaluation, a constraint accesses model elements (and their fields). For example, if C1 defined in Fig. 3 is evaluated on message *turnOn()* in Fig.3 (a constraint instance denoted in short as <*C1, turnOn*>), the constraint starts its evaluation at the context element – the message. It first accesses the receiver object *light* and asks for the base class of this object, *WorkroomLight*. Next, all methods of this class are accessed ({*isOn*, *turnOn*, *turnOff*, *setLevel*}) and their names are requested. This behavior is observed and recorded by the model profiler. We define the model elements accessed during the evaluation of a constraint as a *scope* of that constraint. Our approach then builds up a simple database that correlates the constraint instances with the scope elements they accessed (<Model Element, Constraint Instance> pairs) with the simple implication that a constraint instance must be re-evaluated if and only if an element in its scope changes:

ScopeElements(Constraint Instance)=Model Elements accessed during Evaluation
ReEvaluatedConstraints(ChangedElement) = all CI where ScopeElements(CI) includes ChangedElement

Next, we discuss the algorithm for handling model changes analogous to the discussion above. Thereafter, we discuss the algorithm for handling constraint changes which is orthogonal but similar in structure.

3.4 Model Change

If the model changes then all affected constraint instances must be re-evaluated. Above we discussed that our approach identifies all affected constraint instances through their scopes, which are determined through the model profiler. In addition to the model profiler, we also require a change notification mechanism to know when the model changes. Specifically, we are interested in the creation, deletion, and modification of model elements which are handled differently. Fig. 5 presents an adapted version of the algorithm for processing model changes published in [10]. If a new model element is created then we create a constraint instance for every constraint that has a type of context element equal to the type of the created model element. The constraint is immediately evaluated to determine its truth value. If a model element is deleted then all constraint instances with the same context element are destroyed. If a model element is changed then we find all constraint instances that contain the model element in their scope and re-evaluate them. A model change performed by the user typically involves more than one element to be changed at the same time (e.g. adding a class also changes the *ownedElements* property of the owning package). We start the re-evaluation of constraints only after all changes belonging to a group are processed, i.e. similar to the transactions concept known in databases. Since the model constraints and meta model constraints are alike, our algorithm for handling model changes remains the same. This algorithm is discussed in [10] in more detail.

```
processModelChange(changedElement)
if changedElement was created
 for every definition d where type(d.contextElement)=type(changedElement)
   constraint = new <d, changedElement>
   evaluate constraint
else if changedElement was deleted
 for every constraint where constraint.contextElement=changedElement
   destroy <constraint, changedElement>
for every constraint where constraint.scope contains changedElement
 evaluate <constraint, changedElement>
```

Fig. 5. Adapted algorithm for processing a model change instantly (adapted from [10])

3.5 Constraint Change

With this paper, we introduce the ability to dynamically create, delete, and modify constraints (both meta model and model constraints). The algorithm for handling a constraint change is presented in Fig. 6. If a new constraint is created then we must instantiate its corresponding constraints:

1) for meta model constraints, one constraint is instantiated for every model element whose type is equal to the type of the constraint's context element. For example, if the meta model constraint C1 is created anew (Fig. 3) then it is instantiated three times – once for each message in Fig.3 (<*C1, getDevices*>, <*C1, press*>, <*C1, turnOn*>) because C1 applies to UML messages as defined in its context element.
2) for model constraints, exactly one constraint is instantiated for the model element of the constraint's context element. For example, if the model constraint C4 is defined anew (Fig. 3) then it is instantiated once for the *WorkroomThermostat* as defined in Fig.2 (<*C4, workroomThermostat*>) because this constraint specifically refers to this model element in its context.

Once instantiated, the constraints are evaluated immediately to determine their truth values and scopes. If a constraint is deleted then all its instances are destroyed. If a constraint is modified all its constraints are re-evaluated assuming the context element stays the same. If the context element is changed or the constraint is changed from a meta model to a model constraint or vice versa, then the change is treated as the deletion and re-creation of a constraint (rather than its modification).

```
processConstraintChange(changedDefinition)
if changedDefinition was created
 for every modelElement of type/instance changedDefinition.contextElement
   constraint = new <changedDefinition, modelElement>
   evaluate constraint
else if changedDefinition was deleted
 for every constraint of changedDefinition, destroy constraint
else if condition of changedDefinition was modified
 for every constraint of changedDefinition, evaluate constraint
else
 for every constraint of changedDefinition, destroy constraint
 for every modelElement of type/instance changedDefinition.contextElement
   constraint = new <changedDefinition, modelElement>
   evaluate constraint
```

Fig. 6. Algorithm for Processing a Constraint Change Instantly

4 Model Analyzer Tool

Our approach was implemented as a plugin for the IBM Rational Software Modeler [8]. The incremental change tracker for the IBM Rational Software Modeler is partly provided by Eclipse EMF [11] though we also implemented this approach for non-EMF technologies such as the Dopler product line tool suite [6] and IBM Rational Rose [12]. Fig. 7 depicts two screenshots of the tool. The right shows the IBM Rational Software Modeler. An inconsistency is highlighted in red. The tool displays the deployed constraints (bottom left) and constraints (bottom right shows the constraints for the selected constraint). The left shows a constraint in more detail. A constraint is defined by its name, context element, and OCL/Java code.

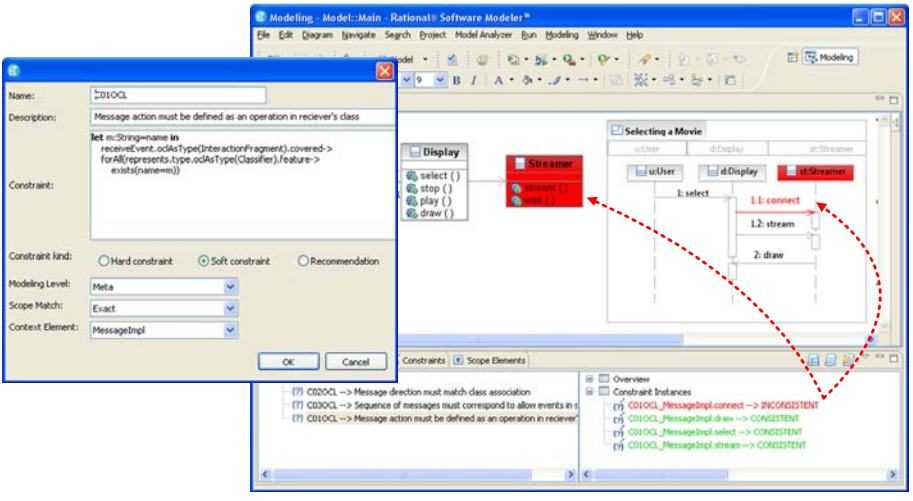

Fig. 7. Snapshot of the Model Analyzer tool

5 Validation

We empirically validated our approach on 24 constraints covering mostly the coherence between UML class, sequence, and state diagrams including well-formedness rules, consistency rules, completeness rules, and general "good practice" rules. The constraints defined in Fig. 3 are examples of the kinds of constraints included in this study. In total, the constraints were evaluated on 20, mostly third-party models with the models ranging in sizes between a few hundred elements to over 100,000 elements.

5.1 Scalability Drivers

To determine the computational complexity of our approach we need to distinguish between the initial cost of creating/changing a constraint and the incremental cost of maintaining it thereafter (with model changes). The initial cost for meta model constraints is a factor of the number of instances per constraint (*#C*) times the number of scope elements they will access during their evaluation (*#SE*). In other words, a new meta model constraint requires the instantiation of #C instances, and each instance must then be evaluated which results in #SE model elements to be looked at. A changed condition of a meta model constraint does not require the re-instantiation of constraints but requires their re-evaluation only. Since the cost of instantiating a meta model constraint is small and a constant (it is the same for every constraint irrespective of the complexity of the condition), we ignore it. The computational complexity for creating and modifying meta model constraints is thus $O(\#C * \#SE)$. In the case of model constraints we need to create and modify a single instance per constraint ($O(\#SE)$).

The computational cost of constraint changes is different from the computational cost of model changes. A constraint as a whole is not re-evaluated with model

changes. Rather, some of its instances *may be*. Changing the model thus affects a subset of the instances of *all* meta model constraints: This subset *#A* must then be re-evaluated by accessing in average the same number of scope elements #SE as above. The computational complexity of re-evaluating the consistency after a model change is thus *O(#A * #SE)*. We will demonstrate next that #C increases linearly with the size of the model (but not the number of constraints), #A increases linearly with the number of constraints (but not the size of the model), and #SE is essentially constant (affected by neither the size of the model nor the number of constraints). We will also demonstrate that this cost still allows for quick, instant evaluations of models.

Our approach scales well for local constraints. Local constraints refer to constraints that must investigate a small number of model elements to determine their truth values. Our approach performed much better than traditional approaches for local constraints and no worse for global constraints. The 24 constraints we evaluated in our study were all local constraints.

Fig. 8 shows the evaluation times associated with creating/modifying meta model constraints and maintaining them thereafter. We see that the cost of creating or modifying a constraint increases linearly with the size of the model. The figure depicts the evaluation time in milliseconds for changing a single meta model constraint. Still, the cost is reasonable *because it is a onetime cost only and we see that this one-time cost is less than 1 second for most constraints (note the error bar which indicates this onetime cost for all 24 constraints with a confidence interval of 95%).* Since constraints do not get changed nearly as often as the model, this cost is acceptable and causes minor delays only.

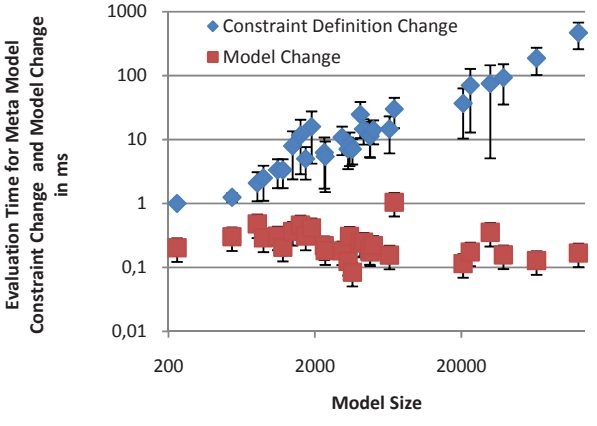

Fig. 8. Eval. time for Meta Model Constraint Changes and corresponding Model Changes

More important is the cost of maintaining a constraint with model changes. *Model changes are recurring (not onetime) and frequent (happen within seconds).* Its cost must thus be much smaller than the cost of changing a constraint for our approach to be reasonable. After instantiation and evaluation, a new constraint is evaluated exactly like discussed in [10]. Each constraint has a chance for it to be affected by a model change. In practice, however, few constraints and few of its instances are affected.

Fig. 8 shows the evaluation time associated with maintaining a *single meta model constraint* with model changes. Since a model change affects only very few instances of a constraint, we see that the evaluation time is in average less than 1ms per model change and constraint. This obviously implies that the evaluation time of a model change increases linearly with the number of constraints but given its little cost, we could maintain hundreds of constraints (of similar complexity) with ease.

The evaluation efficiency of constraint changes is affected by the size of the model whereas the evaluation efficiency of model changes is affected by the number of constraints. Both cases are several orders magnitude more efficient than traditional batch processing, especially in large models.

Fig. 9 shows the evaluation times associated with creating/modifying model constraints and maintaining them thereafter. Model constraints were also evaluated on the same set of 20 UML models; however, since many of these models did not contain model constraints, we added them through random seeding. The seeded model constraints where derived from the meta model constraints and their complexity is thus analogous to them (and thus directly comparable). In our experience, model constraints are no more complex than meta model constraints – the findings presented next are thus applicable under this assumption. We see that the cost of creating or modifying a model constraint stays constant with the size of the model. The figure depicts the evaluation time in milliseconds for changing a single model constraint. *Note that Fig. 9 shows the evaluation time associated with maintaining a single model constraint* with model changes. Since, in average, a model constraint accesses a small number of scope elements only, the probability that a model change affects one of these scope elements is small. The larger the model, the smaller the probability gets. This obviously implies that the evaluation time of a model change decreases linearly with the size of the model. However, our experience is that unlike meta model constraints, the number of model constraints is expected to increase linearly with the size of the model. Thus, a larger model likely contains more model constraints than a smaller model. The data in Fig. 9 shows that the number of model elements is allowed to increase linearly with the model size for the cost of a model change to become constant (as in Fig. 8).

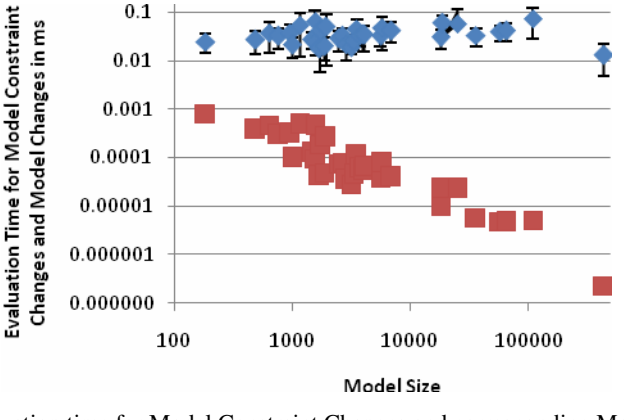

Fig. 9. Evaluation time for Model Constraint Changes and corresponding Model Changes

5.2 Memory Consumption

The memory consumption required for storing the scopes of constraints is the same as presented in [10]. The memory cost grows linear with the size of the model and the number of constraints. It is thus reasonable.

5.3 Threats to Validity

We evaluated our approach on 24 constraints. All of them were scalable which is a good indication that many constraints in general are scalable. However, this does not proof that all constraints are scalable. The works of [13, 14] show that certain kinds of model checking can be exponentially (non linear) expensive. In such cases, our approach would still perform better (certainly no worse) then batch or type-based triggered checkers [2, 3, 8].

It is important to observe that the third-party model we used in our evaluation did not contain model constraints. We therefore used random seeding of meta model constraint instances to evaluate the scalability of model constraints. We believe that this is valid because in our experience, model constraints are no more complex/elaborate than the meta model constraints, which were used for seeding.

6 Related Work

Existing approaches can be characterized based on how they evaluate a model when the constraint or model changes. We see a division between those that perform batch consistency checking and those that perform change-triggered consistency checking. It is also worthwhile distinguishing those that check the model directly [2, 3, 8] and those that check the model after transforming it to a third, usually formal representation [13, 15, 16]. The latter group is more problematic for incremental checking because they also require incremental transformation in addition to checking. And, finally, we see a division between those that allow constraints to be defined/modified at will and those that require constraints to be pre-defined. Most approaches do allow constraints to be added/removed, however, they also require the design model to be recheck in their entirety rather than checking the impact of the constraint change. The approach presented in [17] requires only parts of OCL constraints to be re-evaluated after model changes but works for model constraints only.

Modeling tools such as the IBM Rational Software Modeler [8] or ArgoUML [2] allow engineers to define custom constraints. They even have a notion of incremental checking but both require the engineer to annotate constraints with model element types where the constraint is re-evaluated if any instance of those types changes. We refer to these kinds of approaches as type-based triggering mechanisms because the manual annotation essentially defines what type of change should trigger what types of constraint re-evaluations. This mechanism is essentially a coarse-grained filter that improves the performance of batch consistency checking, however, this mechanism still does not scale because the bigger the model, the more instances of the triggering types it contains.

xLinkIt [3], a XML-based environment for checking the consistency of XML documents, is perhaps the only other technology in existence today that is capable of incremental consistency checking without additional manual overhead. However, the scalability data published in [3] shows a non-instant performance and thus the technology is likely not usable in our context where we care to provide design feedback instantly in real time. Moreover, xLinkIt defines a constraint language that is limited in its expressiveness. This is necessary because the approach analyzes the constraint itself to calculate its triggering conditions which is a complex task.

Incremental consistency checking requires reasoning over design changes rather than reasoning over the entire model state. An explicit emphasis on changes is not new. For example, version control systems such as CVS [18] or SVN [19] deal with changes and they are capable of identifying some form of constraint violations. However, version control systems can at most enforce static, syntactic checks (often referred to as conflicts where multiple stakeholders manipulated the same element) but cannot ensure the rich set of semantic constraints in existence today. More recent work on operations-based model checking [4] shows that there is an increasing emphasis on change during consistency checking. However, their work also appears to require manual annotations to relate changes to constraints (or constraints are required to be re-written from the perspective of changes).

Related to detecting inconsistencies is fixing inconsistencies. This latter issue is out of scope of this paper but it has been demonstrated in [20, 21] it is possible to use the technology for detecting inconsistencies for fixing them at a later time.

7 Conclusion

This paper introduced an approach for the instant checking of dynamic constraints. Engineers can define and modify both meta model and model constraints whenever and wherever necessary and immediately benefit from their instant checking. Engineers need to provide the constraints only – no annotations or any other manual overhead are required. The results of our evaluation demonstrate that our approach is scalable even for large models with tens of thousands of model elements. Our approach provides instant or near-instant error feedback regardless of model and constraint changes.

Acknowledgement. This research was funded by the Austrian FWF under agreement P21321-N15.

References

[1] Fickas, S., Feather, M., Kramer, J.: Proceedings of ICSE 1997 Workshop on Living with Inconsistency, Boston, USA (1997)

[2] Robins, J., et al.: ArgoUML, `http://argouml.tigris.org/`

[3] Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: a consistency checking and smart link generation service. ACM Transactions on Internet Technology (TOIT) 2, 151–185 (2002)

[4] Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: 30th International Conference on Software Engineering, Leipzig, Germany, pp. 511–520 (2008)
[5] Object Constraint Language (OCL), http://www.omg.org/spec/OCL/2.0/
[6] Dhungana, D., Rabiser, R., Grünbacher, P., Lehner, K., Federspiel, C.: DOPLER: An Adaptable Tool Suite for Product Line Engineering. In: 11th International Software Product Line Conference, Kyoto, Japan, pp. 151–152 (2007)
[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns Elements of Reuseable Object-Oriented Software. Addison Wesley, Reading (1994)
[8] IBM RSM, http://www.ibm.com/software/products/de/de/swmodeler
[9] Voelter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: 11th International Software Product Line Conference, Kyoto, Japan, pp. 233–242 (2007)
[10] Egyed, A.: Instant Consistency Checking for the UML. In: 28th International Conference on Software Engineering, Shanghai, China, pp. 381–390 (2006)
[11] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/
[12] Egyed, A., Balzer, R.: Integrating COTS Software into Systems through Instrumentation and Reasoning. Automated Software Engineering 13, 41–64 (2006)
[13] Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: 5th International Conference on Generative Programming and Component Engineering, Portland, USA (2006)
[14] Larsen, K.G., Nyman, U., Wąsowski, A.: On Modal Refinement and Consistency. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 105–119. Springer, Heidelberg (2007)
[15] Engels, G., Güldali, B., Soltenborn, C., Wehrheim, H.: Assuring Consistency of Business Process Models and Web Services Using Visual Contracts. In: Applications of Graph Transformations with Industrial Relevance, Kassel, Germany, pp. 17–31 (2007)
[16] Campbell, L., Cheng, B., McUmber, W., Stirewalt, K.: Automatically Detecting and Visualising Errors in UML Diagrams. Requirements Engineering Journal 7, 264–287 (2002)
[17] Jordi, C., Ernest, T.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of System Software 82, 1459–1478 (2009)
[18] Concurrent Versions System, http://www.nongnu.org/cvs/
[19] Subversion, http://subversion.tigris.org/
[20] Egyed, A.: Fixing Inconsistencies in UML Design Models. In: 29th International Conference on Software Engineering, Minneapolis, USA, pp. 292–301 (2007)
[21] Küster, J.M., Ryndina, K.: Improving Inconsistency Resolution with Side-Effect Evaluation and Costs. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 136–150. Springer, Heidelberg (2007)