Ad-hoc Runtime Object Structure Visualizations with MetaLinks

Peter Uhnák
Department of Software Engineering
Faculty of Information Technology
Czech Technical University in Prague
Czech Republic
uhnakpet@fit.cvut.cz

Robert Pergl
Department of Software
Faculty of Information Technology
Czech Technical University in Prague
Czech Republic
perglr@fit.cvut.cz

Abstract

This paper describes an original method and a tool for an a posteriori analysis of a running object software system, specifically system's runtime structural properties. Highly context-dependent systems pose a challenge of understanding their runtime behaviour. The typical approach is to let the system run and manually observe its runtime properties, which is cognitively demanding and error-prone. Smalltalk, and Pharo in particular, focuses on providing live introspection and immediate feedback during the development. In our method, we take the advantage of these possibilities and combine them with the (relatively new) concept of metalinks as a way to hook into existing code without modifying it, and to set specific attributes to be observed. The result is an analysis tool focused on visualization of ad-hoc runtime structures – that is, providing an analyst with a live view of the stepwise construction and runtime changes of a system based on a UML instance visualization. We demonstrate the tool on analyzing the construction loop of Pharo's Spec UI framework.

1 Introduction

As Robert C. Martin (the famous "Uncle Bob") observes [7], "the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ... [Therefore,] making it easy to read makes it easier to write." It is obvious that the term "reading" means *understanding* the code, which requires building a mental model of the code's behavior in one's brain. Understanding complex code poses serious challenge to our limited cognitive capacities [8].

In this paper, we present a solution to an analysis problem of understanding the run-time behavior of a system that cannot be readily understood from the code itself. We utilize the Pharo live environment [3] and take advantage and extend its liveness to aid our understanding of a complex piece of code of Pharo's Spec UI framework. Our solution is called ad-hoc, meaning the user arbitrarily selects a part of system for analysis, performs the analysis, throws it away and moves on in their understanding, which is opposed to techniques of a systematic code analysis and reverse engineering.

The structure of the paper is as follows:

In section 2, we briefly present tools and concepts used and reused in our solution – Roassal, the visualization engine of choice, MetaLinks – reflective ad-hoc code injection system, and UML Instance Metamodel.

In section 3, we present the solution – a discussion of the metamodel and comparison with UML itself, and various capabilities of the tool.

In section 4, we demonstrate the application on a part of the Spec user interface framework.

Finally, in sections 5 and 6, we summarize our findings and propose future work.

2 Methodology

We briefly present several necessary parts used for our solution.

2.1 Roassal

Roassal [2] is a graphical visualization engine available in Pharo and VisualWorks, which is focused on easy and iterable (agile) creation of visualizations, and promoting the connection between the data and their representations, thus adhering to and enhancing the Smalltalk live environment concept. Roassal is used in a range of use cases such as data plotting, software analysis, and modeling. We chose Roassal as our visualization engine for instance visualization, as it is currently the most mature solution. It also plays well with the OpenPonk modeling platform [13]. We used OpenPonk's Roassal extensions, namely the *uml-shapes* library

IWST '17, September 4–8, 2017, Maribor, Slovenia

ACM ISBN 978-1-4503-5554-4/17/09...$15.00
https://doi.org/10.1145/3139903.3139912

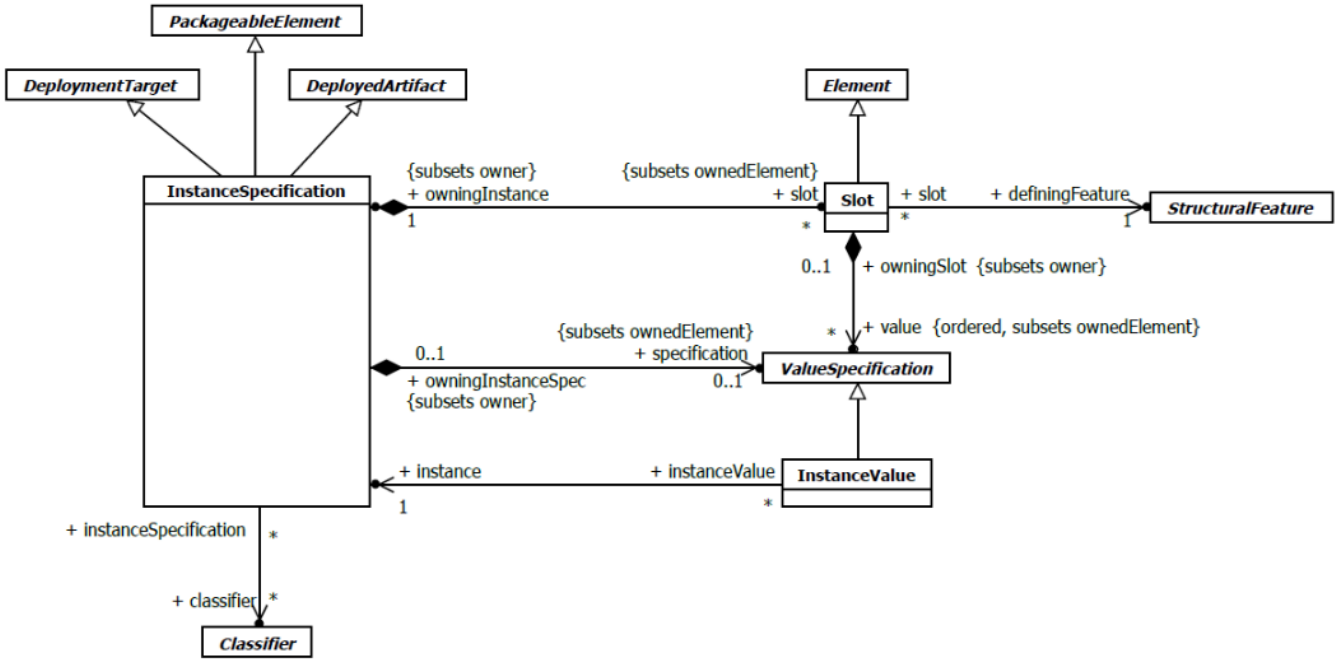

Figure 1. UML Instance Metamodel [9]

that provides pre-build graphical components for construction of UML components and reusing UML Class and Object Diagram notations.

2.2 MetaLinks

One of our goals was to design an unobtrusive solution that enables hooking up into a code without a need to change or annotate the code – a user merely chooses what code should be analyzed and how.

To achieve such flexibility, we utilize MetaLinks [5], a mechanism capable of dynamically hooking into existing code. By linking to abstract syntax tree (AST) nodes, a MetaLink can be used to extract context information such as metadata (the selector of the method, the AST node, etc.), as well as the data themselves (values of the arguments, the object instance under investigation and more). Depending on the kind of AST node being linked into, different options are available, such as the previous and current value of an assignment node.

We illustrate a simple usage of a MetaLink in Code Snippet 1 on a method shown in section 1 of the snippet. A MetaLink (section 2) consists of *metaObject*, which is an object that will receive the reified arguments via the *selector* message send; here we utilize a simple two-argument block that understands the *value:value:* message. In *arguments* we specify what kind of reifications we are interested in – here the object itself, and all the arguments provided to the method that we will link to later. Finally, *control* specifies when should be the MetaLink triggered in respect to the linked AST node, for example *#before*, *#after*, or *#instead*.

```
"1. premise: a class Person with a setter #age":
"Person»age: anAge"
" age := anAge"

"2. MetaLink construction"
metaLink := MetaLink new
      metaObject: [ :object :arguments |
            Transcript
                  log: object className; log: ' ';
                  logCr: arguments first
      ]
      selector: #value:value:;
      arguments: #(object arguments);
      control: #after.

"3. installing link to the AST of the method"
(Person»#age:) ast link: metaLink.

"4. sending a message that will trigger the metalink"
person := Person new.
person age: 32.

"5. removal of the link"
metaLink uninstall.
```

Code Snippet 1: Example of a simple MetaLink usage.

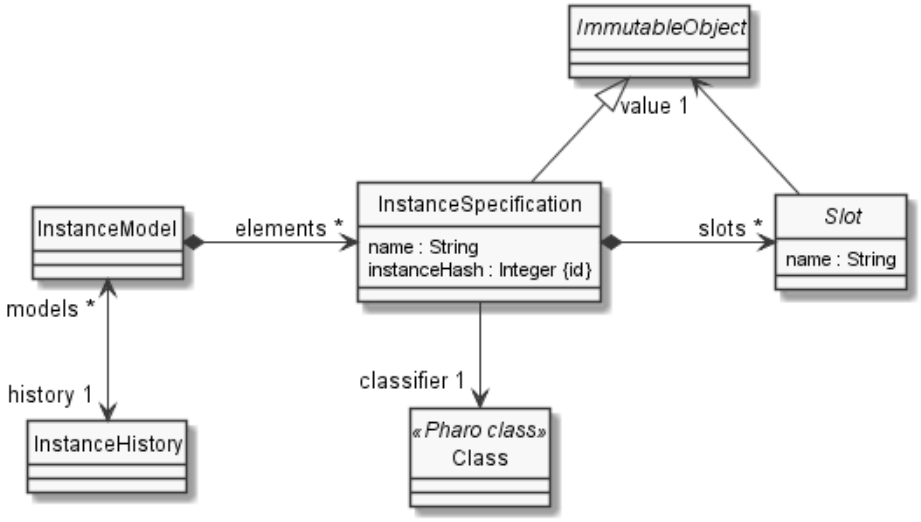

Figure 2. Custom Instance Metamodel

In section 3 of the snippet, we install the metalink to the *Person>>#age:* RBMethodNode[1], but we could choose to install it to e.g. the assignment node in the method[2].

Section 4 sends the *#age:* message to a Person object, which also triggers the MetaLink, executes the behavior in *metaObject* and displays *"Person 32"* in the Transcript log.

This is the basic principle of the way our tool dynamically retrieves information about objects, although we use additional wrappers and extensions of top of the raw MetaLinks.

2.3 UML Instance Metamodel

For this section, we presume basic knowledge of the Class-related UML concepts such as attributes (properties), classes, associations, and links. Further on, we speak about several meta-layers (objects, classes, objects representing instances of objects and their classes, etc.). As this can lead to a confusion, unless explicitly told in the text otherwise, once we refer to objects (instances of a class) by their class names, e.g. "InstanceSpecification", this refers to the *instance* of InstanceSpecification class, while "InstanceSpecification class" refers to the class itself. To also draw distinction between an object in the Instance Model and an object in the analyzed system, we refer to the latter as a *domain object* (and a *domain class* in the context of classes). Finally we use interchangeably the names Object Diagram and Instances Diagram.[3]

The metamodel we use to represent the captured information is based on UML Instance Metamodel [9] (Figure 1) consisting of the further described objects.

InstanceSpecification is an object in the Instance Model representing a domain object. InstanceSpecification is a partial reification of the domain object, including some user-chosen properties of it. The *classifier* link connects the InstanceSpecification to the class that the domain object is an instance of. Although UML permits multiple inheritance (thus the *N:M* multiplicity), in the context of Pharo[4] a direct association is sufficient.

Slot is the instance equivalent to an attribute of a domain class – that is, the concept of attribute such as its name, type, etc. A Slot refers to the domain attribute via the *definingFeature* link. The reified value of the attribute is pointed to from Slot via the *value* link.

ValueSpecification class is an abstract superclass of various value reifications. For primitive types such as Integer, String, etc., UML provides a set of LiteralSpecification classes – LiteralInteger, LiteralString, etc.; alternatively UML offers an Expression or an OpaqueExpression, which both represent a value that is computed from the specified expression. Finally, an *InstanceValue* is nothing more than a reference to another LiteralSpecification in a model.

UML also defines a diagram notation for Instance Models based on UML Object Diagram, which is a slight modification of the commonly-used Class Diagram. For example, as it is often the case in a Class Diagram, a user can visualize a link between two classes either as an attribute, or as a relationship. In Instance Diagram, one could achieve the same as shown in Figure 3. We follow this notation in our solution.

[1] AST node representing the entire method.
[2] *(Person>>#age:) ast assignmentNodes first* (RBAssignmentNode).
[3] The term object is already overloaded, and practically speaking the visualization is of the Instance Models, not the objects they represent.
[4] Pharo is a single inheritance language.

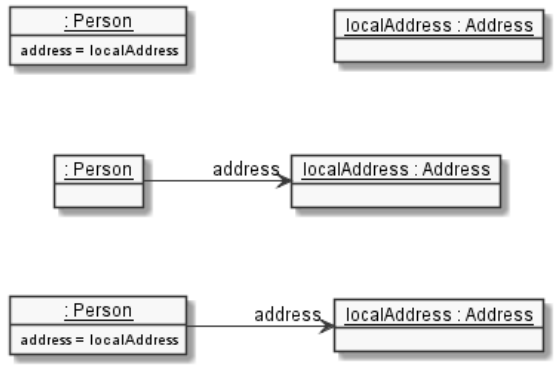

Figure 3. UML Instance visualization and the choice between attribute (top), link (middle), or both (bottom).

3 The Solution

The problem stated in Introduction is the complexity of understanding a complex system with dynamic behavior not easily perceivable from its static structure. In this section, we present a solution facilitating understanding of such a system by looking at the state and relationship changes between instantiated objects in the analyzed system. Furthermore, we discuss various aspects of our metamodel describing the objects and design decisions made in the implementation.

3.1 Differences of our Metamodel from the UML Standard

Our metamodel (Figure 2) differs from UML in two main points:

Firstly, at the time of creating this solution, there was no ready-to-use UML metamodel implementation in Pharo[5], so we implemented our own minimal version sufficient for the ad-hoc needs.

The second difference from the UML metamodel is extending the metamodel with the concept of history to enable the possibility of tracking the changes in the graph of instantiated objects throughout time.

3.2 Model Snapshots

As noted, we are interested in looking at the stateful changes occuring throughout the lifetime of a system. In case of a particular configuration, we seek to know what changes occurred in the system to arrive at such a point, one change at a time.

To achieve fine granularity where we can track each individual change, we perform a model clone of the latest model version available and then apply a new change to it. Cloning the previous *snapshot* is necessary, as the analyzed domain objects do not necessarily form a connected graph, and therefore it would not be feasible to recreate the entire snapshot from scratch. Additionally, some Slot values in the model are result of an Expression that was evaluated in the context of earlier stages of the model and should not be reevaluated again.[6]

3.3 InstanceSpecification And Identity Tracking

In general case, any change can occur within a domain object, therefore we cannot assume existence of a domain identifier that could be used as the domain object identity identifier. Instead, every InstanceSpecification in a model has an automatic unique identifier *instanceHash*. The hash is determined from the *identityHash* of the domain object. Compared to regular domain object's hash, which typically changes when objects (attributes) contained in the object change, identityHash does not change during the entire object lifetime irrespective to changes. These even include *#become:/#becomeForward:* operations which perform pointer and classifier swapping, as demonstrated in Code Snippet 2.

```
a := #(item).
a class. "Array"
a hash. "263321495"
a identityHash. "57596928"

a becomeForward: OrderedCollection new.
a class. "OrderedCollection"
a hash. "264960"
a identityHash. "57596928"

a add: #item2.
a hash. "9550253"
a identityHash. "57596928"
```

Code Snippet 2: Changes in hashes after modyfing operations.

As the identityHash does not change, we use it as a reliable mechanism to track a domain object and connect it correctly to its InstanceSpecification. Note that we cannot use the reference to the actual object, as the object can cease to exist and be garbage collected at some point of the run time of the analyzed system.

3.4 Slot Value Immutability

As we noted earlier, every new snapshot of the model starts by cloning a previous snapshot and then applying changes, which has practical implications in respect to the immutability of values of the slots.

Our solution leaves it to the judgment of the user to decide what value or value reification should be stored in a Slot, however such value cannot be completely arbitrary due to ensuring data consistency.

The value provided by the user is deep-cloned by Pharo's *deepCopy* every time a new snapshot is initialized, where

[5]We briefly return to this in Future Work.

[6]Expressions that are reevaluated during each clone are subject to future versions of the analysis tool.

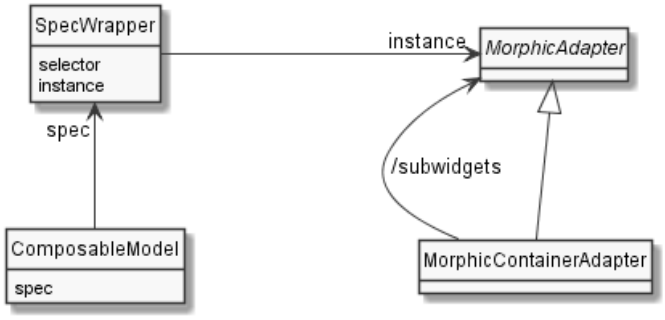

Figure 4. ComposableModel is connected with Adapters via a wrapper class.

deepCopy clones the object representing the value, as well as all its attributes, but no deeper. This means that a *deepCopy* will not correctly clone a graph or a tree of depth larger than two. This problem could be theoretically fixed in some cases with *veryDeepCopy* at the expense of additional memory usage and decreased performance, however we do not consider this a practical problem, as in such cases the provided value would not be a first-class citizen of the Instance Model. Instead, if one were to visualize the instance state of a graph, the preferred approach would be to reify each node of the graph into an InstanceSpecification with links[7] between them. Such approach would not need veryDeepCopy, as all nodes would be reified into first-class citizens (InstanceSpecifications) of the model.

The above also implies that a value in a snapshot (model) should not be affected by any future changes, and therefore storing non-reified deep object as a value could result in data inconsistency. We depict this in the metamodel diagram in Figure 2 via the *ImmutableObject* interface. A way to guarantee that values are not mutable lies in exploring the advancements in object immutability in Pharo [1].

3.5 Garbage Collection Of InstanceSpecifications

The lifecycle of an object, and the end of it in particular, has to be reflected in the Instance Model.

In respect to the end of domain object's lifecycle reification (InstanceSpecification), we identified two context-dependent cases that could be considered the end of the InstanceSpecification's life.

The first case is knowing when the object has been truly purged by the system during a garbage collection (GC) operation. Pharo calls the *finalize* method of an object during a GC operation to which we can hook up and reify the removal of the domain object. This approach is problematic however, as the time of the nearest GC cycle is undeterministic for any practical purposes, as without explicitly triggering the GC, we do not know whether it will occur in a few seconds, or a few minutes. This would force the user to wait undisclosed period of time before the information they are interested in are made available to them.

The second case is the removal from the observed domain. In this context, we remove an InstanceSpecification from a model even though other parts of the system can still point to the domain object being reified. This approach is useful when we consider only a limited view of the entire system, as from the perspective of such a view an InstanceSpecification that is no longer being pointed to from other InstanceSpecifications doesn't exist anymore – if there are no connections within the observed view, then there is no way for the viewed subsystem to recreate the connection, assuming the object still exists elsewhere. We chose this as the default behavior of the tool; when the last link to an InstanceSpecification has been removed (within the model), then the InstanceSpecification is also removed. This default behavior can be changed to preserve the object, but then it is the explicit responsibility of the user to manually clean up orphaned instances when needed.

Note that the solution automatically creates InstanceSpecifications when they are needed, so if an object has been removed from the model and later it was connected back from other part of a system not subject to the analyzed view, the model will automatically reflect this and reify the object back into the model. As the identity is determined by the unique instanceHash, the user could still perform valid comparisons between the newest snapshot and those before the reification has been removed.

4 A Case Study: Analysis Of Spec's Construction Loop

Spec UI framework [14] is a framework commonly used in the Pharo environment for building composable user interfaces. The most common use cases, such as building forms and windows by composing basic input widgets are well supported, however introducing smarter widgets or creating a more dynamic composites poses a comprehensibility challenge.

[7] Slots with values to other InstanceSpecifications.

Spec itself is not the UI visible to the user, but rather an abstract representation of it – a model[8] of the visual representation. For the visual representation, Spec uses a lower-level library Morphic[9] to which Spec delegates and with which it communicates through a series of Adapters that implement the mapping behavior.

The authors identified a complex UI building mechanism of *SpecInterpreter* as one of the reasons of Spec's complex behavior. It is responsible for the instantiation of both the Adapters and their Morphic counterparts, as well as for connecting all the parts together – connecting Spec with Adapters, and Adapters with Morphic. The actual selection of the appropriate Adapters is based on a so-called Spec Layout which mandates the correct adapters for each specific Spec object, as well as the layout organization of all UI elements.

To further complicate the matter, the exact spec layout is flexible, and based on the programmer's choice different layouts are used for the very same Spec composite. The layout can also change during the UI's lifetime, so the appropriate parts have to be rebuilt at respective moments.

Additional requirements during a non-systematic evolution of Spec resulted in a series of rules and behavior tweaks distributed incoherently throughout the implementation, without any precise specification or an easy way to understand them.

The necessity to understand Spec given its complexity was actually the motivation to develop the presented tool: to be able to peek into the changes occurring in the Spec/Adapater/Morphic during building and rebuilding of a Spec UI. Combined with other analysis approaches, this resulted in a new implementation of the SpecInterpreter with well-documented rules.

Here we present a demonstration of our tool applied to a specific part of Spec responsible for connecting objects shown in Figure 4. For the purpose of demonstration we look at four attributes/methods: *SpecWrapper>>#selector:*, whose responsiblity and content wasn't clear to us from the code itself. *SpecWrapper>>#instance:* and *ComposableModel>>#spec:* to see the connections between the presentation layers. And *AbstractMorphicAdapter>>#add:* to capture the composition information, which is an information that is no longer practically available after the UI has been build. We also show this in Figure 4 as a derived association *subwidgets*.

4.1 InstanceSpecification Descriptions

The first part of devising the analysis describes what objects the user is interested in and how they should be reified into InstanceSpecifications.

Every InstanceSpecification description presented in Code Snippet 3 consists of a classifier (a Pharo class reference), which is also used to decide what reification description should be applied to a particular domain object. Further a convenient name of the reified object, which could be provided as a fixed string, or a BlockClosure expression. Such expression is evaluated when a domain object is reified for the first time, thus cloning a model snapshot doesn't reevaluate the expression. Last part of the description is a list of named slots and their optional initial value, which can again consist of an immutable value, or a once-evaluated Expression.

```
modelDefs := LIVInstanceCompositeSpecificationBuilder new.
modelDefs addSpec: [ :spec |
    spec classifier: SpecWrapper.
    spec name: #identityHash.
    spec
        addSlot: #selector -> nil;
        addSlot: #instance -> nil.
].
modelDefs addSpec: [ :spec |
    spec classifier: ComposableModel.
    spec name: #identityHash.
    spec addSlot: #spec -> nil.
].
modelDefs addSpec: [ :spec |
    spec classifier: MorphicContainerAdapter.
    spec name: #identityHash.
    spec addSlot: #added -> [ OrderedCollection new ]
].


history := LIVInstanceHistory new.
history specificationBuilder: modelDefs.
```

Code Snippet 3: Describing properties that should be reified.

Domain objects not explicitly specified can still be reified automatically into an InstanceSpecification with no additional slots. As we mentioned, MorphicContainerAdapter does not hold direct references to the elements it holds (its subwidgets), but because we are interested in them, we have added a new slot *#added* to the InstanceSpecification description that will remember the subwidgets as they arrive to the container. Thus the slots do not need to map to real attributes of a domain object.

4.2 MetaLinks behavior description

Code Snippet 4 expresses the behavioral part – what elements should be connected based on what observations. The API mostly follows the API of metalinks themselves, as shown in Code Snippet 1, but with certain simplifications. We infer *arguments* based on the argument names of the *action* block, and the *target* was streamlined, so the user expresses the targeted method alongside the specification, although the linking (installation) occurs later.

As for the behavior itself, we can see that each link contains *history inNewState: [:model | …]*. This code is responsible for creating a new model snapshot and providing it back

[8] Not to be confused with a model containing the data displayed.
[9] In the future, libraries such as Bloc are planned to be supported.

```
links := MTMetaLinkRegistration new.

links addLink: [ :link |
  link
    target: SpecWrapper»#selector:;
    action: [ :object |
    history inNewState: [ :model |
      model setValueFrom: object to: object selector at: #selector ]]].

links addLink: [ :link |
link
  target: SpecWrapper»#instance:;
  action: [ :object |
    history inNewState: [ :model |
      model setLinkFrom: object to: object instance at: #instance ]]].

links addLink: [ :link |
link
  target: ComposableModel»#initialize;
  control: #before;
  action: [ :object |
    history inNewState: [ :model | model ensureInstanceFor: object ]]].

links addLink: [ :link |
link
  target: ComposableModel»#spec:;
  control: #before;
  action: [ :object :arguments |
    (object spec = arguments first) not & object spec isNotNil ifTrue: [
      history inNewState: [ :model |
        model unsetLinkFrom: object at: #spec ]]]].

links addLink: [ :link |
link
  target: ComposableModel»#spec:;
  action: [ :object |
    history inNewState: [ :model |
      model setLinkFrom: object to: object spec at: #spec ]]].

links addLink: [ :link |
link
  target: AbstractMorphicAdapter»#add:;
  action: [ :object :arguments |
    history inNewState: [ :model |
      model addLinkFrom: object to: arguments first at: #added ]]].

links install. "links uninstall"
history inspect.
```

Code Snippet 4: Describing the behavior of the metalinks.

to the user. Deciding when a new history state should be created lets the user decide what changes should be considered atomic, and thusly the user is controlling how fine-grained the differences between snapshots are.

The connection or disconnection of object in the Instance Model is controlled by two methods and their alternatives:

The method *model setValueFrom: aSourceInstanceSpec to: aTargetObject at: aSlotName* stores an ImmutableObject (String, Collection, etc.) into a slot. The visualization shows this as an attribute value.

The method *model setLinkFrom: aSourceInstanceSpec to: aTargetObject at: aSlotName* reifies *aTargetObject* into an InstanceSpecification, and adds a link between the source and the target. This method reifies all objects irrespective of whether they were previously described in the InstanceSpecification descriptions, but of course without any additional slots.

Both methods have their collection-based counterparts (#addValue…, #addLink…) and all of them have their removal counterparts (#unset…, #remove…), as well as additional methods for various tasks (such as forcibly removing an InstanceSpecification from the model).

The API remains relatively low level, as it is meant as a foundation for creating smarter libraries around it. For example, we have experimented with automatic and dynamic[10] retrieval of all assignment nodes in a class hierarchy and uniformly managing them and installing MetaLinks into them en masse.

4.3 Visualization

For visualization, we use the UML Instance notation (Object Diagram notation), as implemented in the OpenPonk platform [13]. The visualization is embedded as an GTInspector [4] extension of the *InstanceModel* class. To see the visualization, the user inspects the history and steps trough the created models.

Because the diagrams can quickly grow in size, to meaningfully show the differences between each snapshots, we apply a diff between the shown model and its predecessor, and we color the differences. If a diff between more distant models is desired, the user can use the same code and perform a diff inside the GTInspector window (Figure 7).

The diagram is also automatically generated, therefore we have to apply automatic layouting. At the moment, we use a grid-based layout where elements are placed in a grid in the order of their addition. Although this is not optimal in respect to some visual aesthetic aspects[11], it preserves the mental map of the user – the user roughly remembers where previously added objects are placed and can focus more on the newly added ones. If we changed the placement in every snapshot, the user would not move through the history that naturally. We have deemed it best in this situation to prioritize the mental map aspect, but balancing of mutually competing visual aspects is still subject to ongoing research [12].

```
ui := DynamicComposableModel new.
ui instantiateModels: #(btn ButtonModel).
ui btn label: 'Btn'.
layout := SpecLayout composed
    newRow: [ :row | row add: #btn ];
    yourself.
ui openWithSpecLayout: layout.
```

Code Snippet 5: Creating a simple window with a button.

To finalize the Spec example, we apply the aforementioned analysis on a simple code creating a Spec window with a single button. Code Snippet 5 results in a window in Figure 5.

[10] In the sense that any change in the source code will be automatically reflected.

[11] Such as line overlapping, hierarchy, etc.

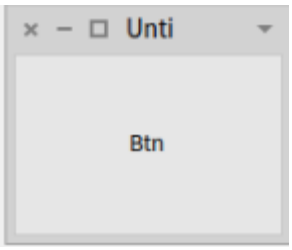

Figure 5. Created window with a button.

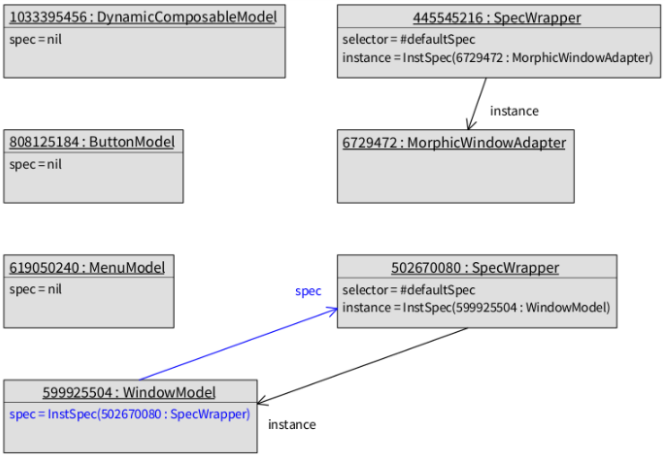

Figure 6. An Instance Diagram with an unexpected link from SpecWrapper to WindowModel

After the construction of the UI has been completed, the analysis provided us with a total of 86 model snapshots providing us with a step-by-step history.

In Figure 6, we see a SpecWrapper surprisingly pointing with *instance* to WindowModel, even though it should point in the other direction (to an Adapter) – the implementation uses this attribute as an ad-hoc storage for its values. From further snapshots[12], we observe additional behavior, such as adapters and morphs being created, connected, and then thrown away needlessly and replaced by different ones.

Figure 7 shows a view of the inspector with manually performed diff between two distant models. A user could also use the diff information directly itself[13].

5 Related Work

Software visualization has been thoroughly studied and there are various solutions and approaches available [6, 10, 11]. In particular, a visualization of variables (pointers, structures, etc.) in a debugged context has been explored [15], but to our knowledge not applied in practice in any major ways, bar small extensions [16]. Such approach is constrained by availability of variables and by the memory representation

[12] Not shown here for brevity.

[13] A collection of added, modified, and removed properties

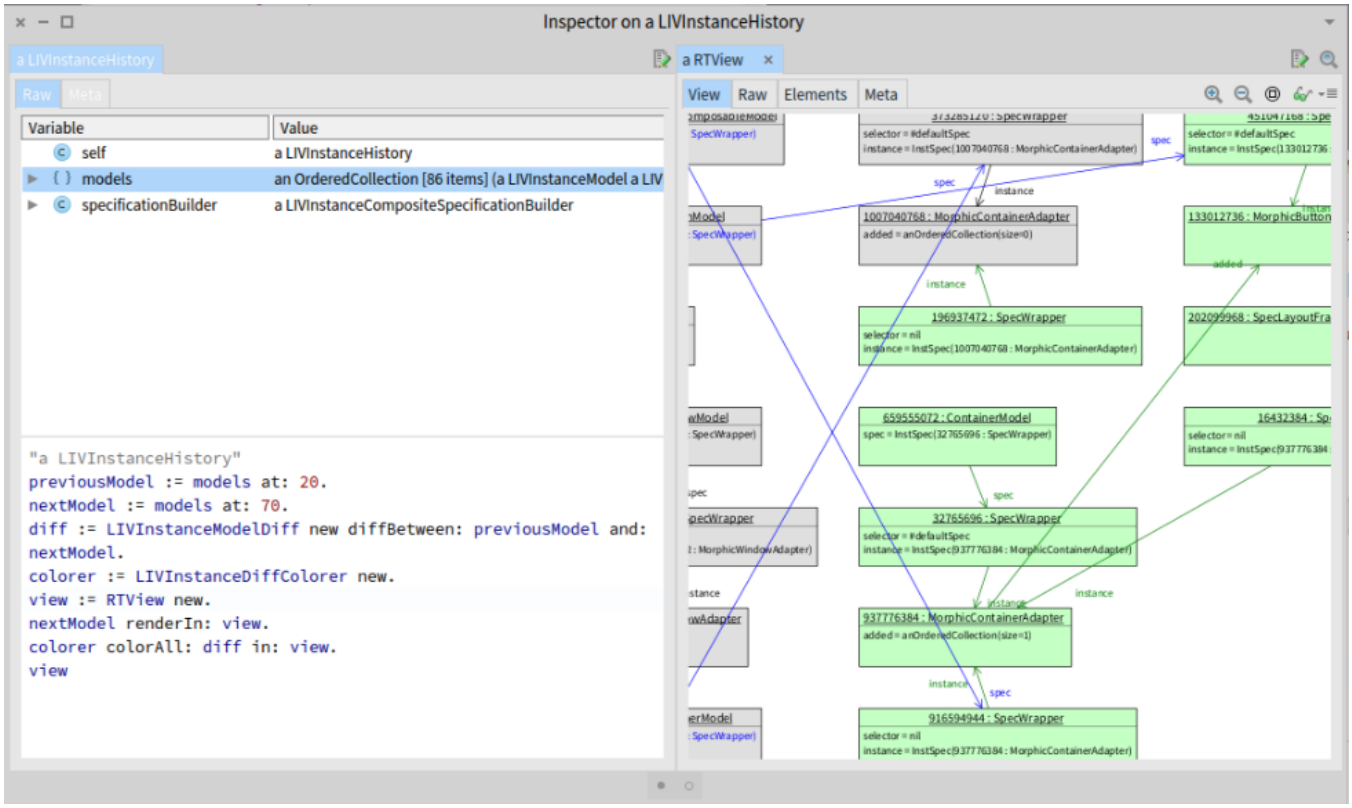

Figure 7. A GTInspector view on the history, with manually constructed diffs.

itself. Our approach circumvents this problem by prescribing what information should be captured, how, when, and where. This offers major flexibility advantage, as the user is not limited to a single debugging context, but they can specify what contexts should be combined together. Further, the user can combine co-related information that is not easily perceivable from the code or the memory state and relationships of objects, such as our presented example with the computed slot. Finally, we represent the captured information as a model, thus enabling further application and manipulation.

6 Conclusions and Future Work

The motivation for the ad-hoc run-time object structure visualization was to facilitate analyses of complex code. We developed it for a practical task of refactoring SpecInterpreter.

The tool is practically usable (although not thoroughly tested), however there are several open topics for future work.

We remind that the visualization presented is just a projection of the underlying Instance Model. So even though our original intention was limited, we are actively interested in extending it for other use cases presented further.

As we explained in the Introduction, this is just an ad-hoc tool. Naturally, formulating a more complete methodology of code analysis may be a next step. A systematic analysis approach would entail building a domain class model, against which the Instance Model would be compared to show adherence. From the opposite perspective, a domain model could be gradually constructed in an automated fashion by observing the run-time behavior and combining the Instance Models over time.

In our implementation, we took a preliminary simplified approach of connecting instances just to system classes. We are in ongoing development of the complete UML metamodel, which may be used instead the ad-hoc one in future. However, it needs considerations, as the tool may become too heavy by depending on the entire UML metamodel and by increasing the complexity associated with UML modeling. This could be addressed by decomposing the metamodel and loading only parts and packages that are actually being used.[14] Another path of advancement is better automation and management of the descriptions to simplify and streamline practical use. This includes further exploration of inferencing connections between objects and their attributes, as noted in the Spec Case Study. Further plans include introducing automatically evaluated Expressions on each snapshot update, and a possible integration with GTDebugger in some manner.

[14]Theoretically the Instance Metamodel depends on the entire UML metamodel, as all elements are tied together in some way, however in practice only small fraction of extra classes is necessary.

Acknowledgments

This research was supported by CTU SGS grant No. SGS16/120/OHK3/1T/18 and contributes to the CTU's ELIXIR CZ Service provision plan.

References

[1] Clément Bera. 2016. A low Overhead Per Object Write Barrier for the Cog VM. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*. ACM, 22. http://dl.acm.org/citation.cfm?id=2991063

[2] Alexandre Bergel. 2016. *Agile Visualization.* agilevisualization.com

[3] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, Jannik Laval, and Jérôme Bergel. 2013. *Deep into Pharo.* Square Bracket, [S.l].

[4] Andrei Chis, Tudor Gîrba, and Oscar Nierstrasz. 2014. The Moldable Inspector: a framework for domain-specific object inspection. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014).* http://esug.org/data/ESUG2014/IWST/Papers/iwst2014_The%20Moldable%20Inspector_a%20framework%20for%20domain-specific%20object%20inspection.pdf

[5] Marcus Denker. 2008. *Sub-method Structural and Behavioral Reflection.* Ph.D. Dissertation. University of Bern.

[6] Stephan Diehl. 2007. *Software Visualization Visualizing the Structure, Behaviour, and Evolution of Software.* Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg. OCLC: 300263699.

[7] Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1st edition ed.). Prentice Hall, Upper Saddle River, NJ.

[8] George A. Miller. 1956. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review* 63, 2 (1956), 81–97. https://doi.org/10.1037/h0043158

[9] OMG. 2015. UML v2.5. (March 2015). http://www.omg.org/spec/UML/2.5

[10] M.J. Pacione, M. Roper, and M. Wood. 2004. A novel software visualisation model to support software comprehension. IEEE Comput. Soc, 70–79. https://doi.org/10.1109/WCRE.2004.7

[11] Seonah Lee, Gail C. Murphy, Thomas Fritz, and Meghan Allen. 2008. How can diagramming tools help support programming activities? IEEE, 246–249. https://doi.org/10.1109/VLHCC.2008.4639095

[12] Peter Uhnák. 2016. Layouting of Diagrams in the DynaCASE Tool. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology. (2016).

[13] Peter Uhnák and Robert Pergl. 2016. The OpenPonk modeling platform. ACM Press, 1–11. https://doi.org/10.1145/2991041.2991055

[14] Benjamin Van Ryseghem, Stéphane Ducasse, and Johan Fabry. 2014. Seamless composition and reuse of customizable user interfaces with Spec. *Science of Computer Programming* 96 (2014), 34–51.

[15] Thomas Zimmermann and Andreas Zeller. 2002. Visualizing memory graphs. In *Software Visualization*. Springer, 191–204. http://link.springer.com/chapter/10.1007/3-540-45875-1_15

[16] Zoltan Ujhelyi. 2016. Debug Visualisation for Eclipse. (2016). https://marketplace.eclipse.org/content/debug-visualisation-eclipse