Automatic Generation of Monitoring Code for Model Based Analysis of Runtime Behaviour

Michael Szvetits
Software Engineering Group
University of Applied Sciences Wiener Neustadt
Wiener Neustadt, Austria
Email: michael.szvetits@fhwn.ac.at

Uwe Zdun
Software Architecture Research Group
University of Vienna
Vienna, Austria
Email: uwe.zdun@univie.ac.at

Abstract—**Software systems are getting increasingly complex, which makes them inherently harder to understand and instrument when their behaviours should be analyzed and adapted. Observing a system requires an examination of its implementation and writing the appropriate monitoring code. This process can be both time consuming and error prone, especially if high-level system properties should be analyzed which are not directly reflected in the implementation. Furthermore, analysis needs often arise at runtime and are handled in an unsystematic way that is not reusable: For similar analysis tasks, the process of examining the system and writing the monitoring logic must be repeated. In this paper we present a language to specify recurring monitoring patterns which are automatically expanded into monitoring code for given models of the analyzed system. As a consequence, the effort for writing monitoring code is reduced and recurring analysis tasks are better supported through automatic code generation.**

Keywords-**generation; traceability; model; monitoring**

I. Introduction

Software models support development teams in communicating ideas, abstracting technical details and analyzing high-level properties of the structure and behaviour of a system. Traceability is an important concept to realize links between such software models and their corresponding implementation artefacts to increase the overall understanding of a system, especially when dealing with software evolution and its accompanying change impacts [1]. Traceability links guide the stepwise refinement of requirements, architectural components and the software development process itself and usually represent predecessor-successor or master-subordinate relationships between software artefacts [2].

If the runtime behaviour of a modelled system element must be analyzed (e.g., to measure the performance of architectural components or modelled operations), one has to follow the traceability links, analyze the corresponding implementation and write the necessary monitoring code. However, analyzing runtime behaviour with the help of models requires that traceability links not only exist between software development artefacts, but also between logged runtime events and the model elements they originate from. This forces the analyst to write the monitoring code carefully and encode the traceability links into the monitoring instructions, which means that the analyst is unnecessarily concerned with technical details of the system environment and instrumentation tools. Furthermore, analysis needs often arise rather spontaneously at runtime when unexpected behaviour is noticed, which means that the running system must provide platform-specific adaptation facilities to dynamically load and interpret the written monitoring code. While the latter is more a technical challenge, two challenges remain for the analyst:

1. The process of manually examining the system and writing the monitoring code is tedious and error prone for high-level model elements since the corresponding implementation parts can be complex and scattered over multiple artefacts within the observed system.

2. Writing platform-specific monitoring code is likely focussed on the current analysis task at hand without reusability in mind. For similar recurring analysis tasks, the process of following the traceability links and writing the appropriate monitoring code must be repeated even if the target platform (i.e., the environment and set of frameworks which allow the system and the monitoring code to run) is the same. In an ideal setting, analysts would write the monitoring code only once for every target platform instead of doing repetitive work.

As a result, a more systematic approach is needed where high-level monitoring needs that arise at runtime are better supported regarding reusability of low-level, manually written monitoring code that is related to complex and scattered source code artefacts. We investigate the following research questions:

- *RQ1:* How can the effort of writing monitoring code for high-level runtime characteristics be reduced?
- *RQ2:* How can existing monitoring code be reused to support recurring analysis tasks?

The contribution of this paper is a systematic methodology paired with a pattern matching based template language to specify recurring, platform-specific monitoring patterns. Monitoring patterns are transformed into a code generator which takes a model of the analyzed system, a traceability model and a declaration of the desired runtime events as input. If the generator can match the models and the monitoring needs against the specified monitoring patterns, it expands the patterns into the monitoring code that fits the current analysis task. As a result, the effort for writing monitoring code is reduced, the analyst is not concerned with technical details and recurring analysis tasks are better supported.

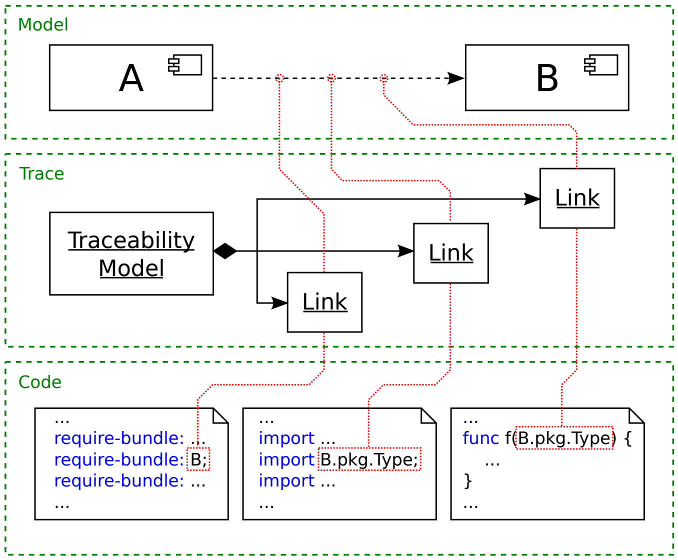

Fig. 1. Motivating example regarding code scattering

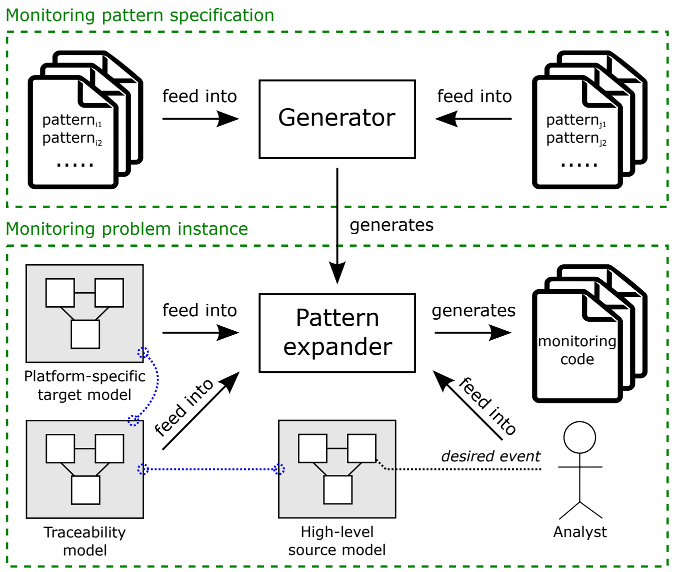

Fig. 2. Overview of the pattern based approach

II. Motivating Example

Consider a scenario where a user wants to analyze some characteristics of a connector between two components, like the count of calls between them or the time that is actually spent if one component accesses the other. In a component diagram, the model element of interest would be the dependency relationship between the two components. If traceability links to its corresponding implementation artefacts are present, the user can navigate those links and write the necessary monitoring code for the affected code fragments. However, the dependency relationship can manifest itself in various modules and locations, usually scattered over the system, which makes writing monitoring code a time-consuming and error-prone task. Fig. 1 shows this scenario, the red lines represent the source and target elements of the traceability links.

Now consider that a similar analysis task has to be performed again, either for the same system or a system that is implemented with the same target platform in mind (e.g., Java/OSGi). The previously obtained knowledge of writing the appropriate platform-specific monitoring code cannot easily be applied to the new problem: The process of following traceability links and writing the appropriate monitoring code must be repeated and might even lead to a large amount of duplicated code. Our approach allows a user to specify platform-specific patterns instead of writing the monitoring code in the first place, and let an automatic routine explore the observed system for those patterns to generate the needed monitoring code. This renders the need for writing repetitive code for similar analysis tasks unnecessary.

III. Approach Overview

Fig. 2 shows an overview of our approach. Our approach assumes that model elements of a source model are related to (possibly many) model elements of a platform-specific target model via traceability links. The traceability links may have references to each other to capture essential relationships between them. Our approach assumes that the target model closely represents the implementation code, and thus the system under observation. Specifically for Java, tools like MoDisco can be used for creating and maintaining a platform-specific target model of the system almost automatically.

If a high-level analysis task is performed, the analyst states the desired metric of a model element (e.g., the execution time of a modelled behaviour) in the source model. Such a metric is essentially an aggregation expression over the desired runtime events, for example the sum of time differences between start and end events of a behaviour to obtain the overall execution time. We define a runtime event as the most basic unit which enables the extraction of information from a running system at a specific point in time. A runtime event thus consists of a time stamp and arbitrary data, which is often called event arguments in object-oriented programming terms. A set of reusable event types and a demonstration of aggregating events are presented in our previous research results [3]. However, our previous research results do not address the generation of monitoring code which yields the runtime events that are necessary for performing spontaneously arising analysis tasks.

Having the traceability model at hand, the selected model element of the source model can be traced to the corresponding model elements of the target model with the help of the traceability links. The process of finding the model elements of the target model also incorporates references between the traceability links. The result of this process is a subset of the model elements in the target model that are potentially relevant for the current analysis task.

Together with a declaration of the desired runtime events, this subset of model elements is then matched against known monitoring patterns to check if an automatic generation of appropriate monitoring code is feasible. The declaration of the desired runtime events is a necessary input for this step since for one and the same subset of model elements, the appropriate monitoring code differs depending on the monitoring desire (e.g., measuring the execution time and the call count of one and the same behaviour requires different monitoring codes).

Regarding the specification of the monitoring patterns, our approach provides a language which makes the steps of defining the monitoring desire, navigating the target model and writing the monitoring logic explicit (and thus, reusable). More precisely, the language allows the analyst to operate on the traced subset of model elements, utilize the relationships encoded into the traceability model and write the actual monitoring code. Specifying patterns for generating monitoring code is more flexible than other generation techniques like the factory pattern since it allows to integrate arbitrary generation instructions and generate code for arbitrary platforms.

IV. Approach Details

A. Pattern Matching Language

The motivating example in Section II demonstrated the need of an automatic routine to explore a model of the observed system for specific patterns to generate monitoring code. A well-known strategy for exploring a solution space for possible solutions that are restricted by constraints is backtracking. Backtracking guarantees to find all solutions by recursively exploring the solution space in depth-first order and reporting solution candidates if they fulfil the given constraints, usually defined in the form of predicate expressions. We use a backtracking based approach to find all model elements of interest in the target model by exploring the relationships that are encoded in the model. Constraints are specified by user defined predicates which narrow down the search and bind variables to model elements of interest which can then be used in template expressions.

For context-specific analyses, we were looking for a declarative way to define patterns that are extracted from the target model and a way to express hierarchical relationships between model elements. For the former part, we were inspired by Prolog because it has a declarative syntax and is inherently backtracking based, which fits the first need described above. For the latter part, we decided to provide predefined predicates largely inspired by XPath as shown in Table I. Although inspired by XPath, we decided not to extend one of its existing implementations because some of our predicates cannot be expressed easily in XPath (e.g., *event*) and its syntax does not fit the rest of our template language.

For the actual code generation, we use a template based code generator to process the elements of the target model that match the patterns described by the user defined predicates. The templates are independent from the process of creating the target model, which means that the applicability is not restricted to traceability-related approaches, but also covers arbitrary monitoring tasks as long as a target model exists. Furthermore, the backtracking logic is hidden from the analyst and can be adapted without changing existing, user defined monitoring patterns. As a result, monitoring patterns can be reused across many systems and monitoring tasks.

TABLE I
Predefined predicates that allow to navigate the target model

Predicate	Description	Example of Use
ancestor(X, P)	Finds parent P of X in a transitive manner.	Find an enclosing type or behaviour.
child(X, C)	Finds child C of X, i.e. X is a parent of C.	Find a method of a given class.
descendant(X, D)	Finds child D of X in a transitive manner.	Find a method in a given package.
element(X)	Finds model element X in the target model.	Declare a root model element, e.g. a class.
event(X)	True if the desired event of the transformation matches X.	X is the name of the desired event.
findall(X, P, L)	Enumerates all solutions for variable X in predicate P into list L.	Find all packages a class is located in.
parent(X, P)	Finds parent P of X, i.e. X is a child of P.	Find the method of a parameter.
sibling(X, S)	Finds element S with the same parent as X.	Find a related pair of getter and setter.

The key difference between our approach and other model-to-text transformation approaches is that a monitoring pattern can appear anywhere in a template, which then gets expanded as long as the backtracking algorithm is able to satisfy the constraints declared by the pattern. As a result, the analyst doesn't need to follow traceability links manually and write the monitoring code accordingly, but instead defines a monitoring pattern once and let the backtracking mechanism expand the monitoring instructions where necessary. The syntax of such a monitoring pattern is as follows:

⟨*pattern*⟩ ::= ⟨*constraints*⟩ '`{`' ⟨*template*⟩ '`}`'

⟨*constraints*⟩ ::= ⟨*predicate*⟩ [',' ⟨*constraints*⟩]

⟨*predicate*⟩ ::= ⟨*name*⟩ ['(' ⟨*params*⟩ ')']

⟨*params*⟩ ::= ⟨*name*⟩ [':' ⟨*type*⟩] [',' ⟨*params*⟩]

⟨*name*⟩ ::= 'a'|...|'z' | ⟨*name*⟩ ('a'|...|'z'|'A'|...|'Z'|)

⟨*type*⟩ ::= 'A'|...|'Z' | ⟨*type*⟩ ('a'|...|'z'|'A'|...|'Z'|)

⟨*template*⟩ ::= `template with params in scope`

A monitoring pattern consists of user defined constraints and a template which gets expanded if the backtracking algorithm is able to satisfy all of the constraints. Each constraint is a predicate with named parameters (see Table I for examples). When the transformation algorithm encounters a pattern, it stepwise tries to satisfy the given predicates by binding their named parameters to concrete model element instances of the target model. This constraint satisfaction step utilizes the hierarchical relationships between the elements found in the target model and the traceability model. If a predicate cannot be satisfied, backtracking takes place to find an alternative solution for the most recently satisfied predicate before retrying the failed one again. If all predicates can be satisfied, the corresponding template is expanded, whereas the bound parameters can be used in template expressions to tailor the generated code to various monitoring needs. Within a template, text and string literals are expanded as is, while expressions using the bound parameters are enclosed in pipe symbols to indicate special interpretation. The process is repeated for every combination of model elements that satisfy the pattern.

B. Application to the Motivating Example

To demonstrate the usage of the language, we create a monitoring pattern to solve the problem that was introduced by the motivating example in Section II. We choose the Java meta-model provided by the MoDisco project for the platform-specific target meta-model. This meta-model describes the building blocks of the Java programming language (e.g., class definitions, method signatures) and allows us to reference Java language concepts in the monitoring patterns.

We demonstrate the counting of method calls between the components based on the traced import statements. We can achieve this in a crosscutting manner with the help of generated AspectJ code in the following way (the pattern is part of a bigger template which is omitted here):

```
event(DataExchanged), element(imp : ImportDeclaration),
parent(imp, c : CompilationUnit), child(c, t : Type) {
    @Before("call(* |imp.importedElement.name|.*(..)
            && within(|c.package.name|.|t.name|+)")
    public void measureCallCount_|i++|() {
        // Yield event that cross-package call occurred:
        // DataExchanged event = new DataExchanged(...);
    }
}
```

The predicate *event* defines the desired event and must be communicated by the analyst through the modelling environment. This is necessary to disambiguate the desire of the specified monitoring pattern because two different patterns could target the same set of model elements in their predicates, but differ in the code that should be generated based on the runtime characteristic that should be measured. An example would be a behaviour for which either the call count or the execution time should be analyzed.

The predicate *element* searches the relevant part of the target model for a Java element of type *ImportDeclaration*, for which the predicate *parent* matches its enclosing compilation unit. The predicate *child* then locates a type (class, interface, etc.) of the compilation unit which contains the import statement of interest. In combination with the backtracking mechanism, we find all types that potentially make use of traced import statements and generate the appropriate monitoring code for each occurrence. Similar rules can also be written for other concrete dependency manifestations like bundle references or parameters (recall Fig. 1). Note that in the example above, the predicates *parent* and *child* can also be substituted by the predicate *sibling* (see Table I) to shorten the pattern while achieving the same result. Also note that no elements of the source meta-model are referenced, which means that the pattern is independent from the used modelling language.

V. Case Study

We illustrate the applicability of our approach in the context of a case study where runtime characteristics of an autonomously acting LEGO robot are assessed. The LEGO robot was built and implemented during a course at the University of Applied Sciences Wiener Neustadt. The robot is able to calibrate itself, follow a user-defined path, receive step-by-step directions from an external operator and discover its environment on its own. The robot software was realized in a model-driven manner by generating OSGi bundles and Java stubs from six UML models (use case, component, package, class, activity and state diagram) and implementing the corresponding Java code for the generated artefacts, leading to a system consisting of 4195 lines of code. Model transformation yielded the needed traceability model. We utilized MoDisco for obtaining the Java target model of the generated code. The resources of the case study (toolchain, models, model transformation scripts, monitoring patterns, generated monitoring codes and additional notes) are available online[1].

A. Effort Reduction

One of the robot software components controls the movement and the general behaviour of the robot, while another one is responsible for analyzing and finding paths when the robot explores the environment on its own. During exploration, these two components heavily interact with each other to steer the robot along the computed paths. For assessing the degree of coupling between these components, we wanted to measure the count of method calls between them using aspect-oriented techniques. This turned out to be a complex and time-consuming task for the analyst because for every pair of packages contained in the two components, a corresponding pointcut and advice must be formulated. Furthermore, writing these aspect-oriented constructs is highly repetitive since the code fragments only differ in their respective package paths.

As an alternative, we decided to write a monitoring pattern which is expanded for every pair of packages within two dependent components. In the target model, the dependency between two components is realized as an OSGi manifest attribute of the form *Require-Bundle: lego.path*. Having the traceability model and the target model at hand, it is quite easy to write a monitoring pattern which selects the manifest attribute, finds the corresponding packages and generates the necessary monitoring code (see the full template online):

```
event(DataExchanged), element(man : ManifestAttribute) {
    if (man.key.equals("Require-Bundle")) {
        /* Find parent bundle of "man" (source).
         * Find bundle named in "man.value" (target).
         * Iterate all packages of source bundle.
         * Iterate all packages of target bundle.
         * Generate the code for each pair. */ }
}
```

For the analyst, measuring the count of method calls between the two components is then just a matter of annotating the dependency between the components with the desired metric. The metric language is out of the scope of this paper, but essentially allows the analyst to apply various selection and aggregation functions (e.g., filter, minimum, maximum, average) over received runtime events and access their individual properties while remaining on the model level. The automatic routine extracts the desired event from the metric expression (here: *DataExchanged*), utilizes the traceability model to find the model elements in the target model which correspond to the annotated model element (here: the manifest attribute

[1]see: http://jarvis.fhwn.ac.at/case-study-robot/

that corresponds to the annotated dependency) and expands the pattern accordingly. The pattern can also be reused for annotating dependencies between other components without writing any additional monitoring code.

For a quantification of the reduced effort, we experienced that the monitoring pattern needed less lines of code than writing the generated aspect (45 vs. 55, see online resources). However, in our case the pattern was only expanded four times. We expect that our approach outperforms manual writing of monitoring code in a much larger scale when the system under observation tends to get bigger. In our particular case, the effort for writing the monitoring pattern is constant, while the effort for writing the monitoring code manually increases with the term $m * n$, where m and n are the count of packages in the dependent components. Another interesting fact is that writing the monitoring code manually for the presented case leads to 85.5% duplicated code for the whole aspect (see online resources). Our approach does not suffer from this repetition.

B. *Reusability*

Regarding reusability, we applied our written monitoring patterns to another system which is modelled using a different modelling language, but is also realized in a model-driven manner with the same target platform (and thus, the same target meta-model) in mind. The system is a simple online shop application which is modelled with the help of components and their contracts using SOA Designer[2]. The models and their corresponding code generator were created during a course at the University of Applied Sciences Wiener Neustadt for the purpose of teaching model-driven techniques, but the system was not fully implemented like the robot system.

The system consists of seven components and their respective interfaces which were transformed via model transformation into interacting OSGi bundles and Java interfaces. We wanted to measure the count of authentication requests (i.e., the number of requests between specific interfaces) and wanted to reuse the monitoring pattern of the robot system which already yields the necessary events of interacting components. While we were able to reuse the monitoring pattern, we had to adapt it slightly to generate the monitoring code only for the particular authentication interface and not for any type within the target component (which was the case in the robot system). This modification, however, was quite simple and only needed an additional clause in the pattern expression and swapping a wildcard expression with a concrete interface name.

Note that another pattern was tested regarding effort reduction and reusability between the two systems. The result demonstrates a complete reuse of the pattern without further modifications. This case can be found in the online resources.

VI. Discussion

Regarding applicability, although we use UML/SOA and AspectJ throughout this paper, our approach is not limited to these concepts since the monitoring patterns can be expanded to arbitrary monitoring code. Many other scenarios are conceivable, for example the analysis of business processes (e.g., modelled with BPMN) where the annotation of a process activity generates monitoring code that is attached to the underlying business process execution engine.

In our approach, we assume that writing monitoring code is equally complex as writing monitoring patterns for the target model since the model is an exact representation of the implementation. However, we could show that the degree of reusability is very high for the monitoring patterns. When writing monitoring code by hand (i.e., not using our approach), one could achieve better results through refactoring (e.g., to prevent duplicated code). However, we argue that monitoring code for an ad-hoc arising analysis task is seldomly written with good software design principles in mind.

Regarding the research question *RQ1*, the presented case study indicates that the effort of writing monitoring code can be reduced by a large amount if monitoring patterns must be expanded multiple times. We achieve this by making the exploration of the target model explicit using a pattern based transformation language which allows to define the exploration steps that would otherwise be done manually for every recurring analysis task. The resulting pattern expander not only allows to generate monitoring code for specific analysis tasks, but also abstracts platform-specific details of the monitoring code from the analyst. As a result, the analyst is not concerned with concepts of the target model or the monitoring technique (e.g., AspectJ) and can focus on writing high-level aggregation expressions of the yielded runtime events.

Regarding the research question *RQ2*, the presented case study demonstrates that monitoring patterns can be applied without or with minimal changes to recurring analysis tasks which concern systems running on the same target platform. The case study also demonstrates that monitoring patterns are independent of the modelling language used for designing the system, which allowed us to transfer the idea of model based monitoring seamlessly between the two software systems.

Summarized, our pattern based approach yielded promising results to be a viable alternative for manually written monitoring code while abstracting from the technical details of the actual monitoring environment. However, in this paper we linked the term *effort* very strongly to the amount of code lines that must be written, which highly depends on the coding style of the analyst. Many other characteristics are conceivable to compare the reduction of the effort, for example the time that an analyst actually spends on writing the monitoring instructions or the time that an analyst actually needs to manually follow the traceability links. However, these comparisons are much harder to perform within experimental settings since the analysts must receive a basic training in the used monitoring techniques, which is a task that cannot easily be performed without introducing a certain amount of bias.

VII. Related Work

The *models@run.time* research community studies models that are causally connected to a running system to enable monitoring and control on a higher abstraction level than the

[2]see: http://marketplace.obeonetwork.com/module/soa

code [4]. Various strategies have been proposed to maintain such a causal connection between the system and its models, like Triple Graph Grammars [5], model transformation [6] and layered interpretation of monitored data [7]. While models at runtime ensure that maintenance tasks can be performed on a more abstract level, they often require separate meta-models and transformation scripts for the monitoring infrastructure. Our approach utilizes existing models from the software design phase and traceability links that are created automatically when model transformations are performed. As a result, our approach is much easier to apply in real world scenarios where models of a system already exist.

We solved the problems introduced by our motivating examples with the help of aspect-oriented techniques. Various other approaches exist that utilize aspect-oriented programming in combination with model driven engineering techniques [8]–[10]. Many of the approaches only support the generation of stubs [11], [12] or focus on extending UML to transform custom aspect models to their corresponding code [12]. To the best of our knowledge, there exists no systematic approach that addresses reusability at the model level by utilizing traceability information combined with a model of the actual implementation. Furthermore, using behaviour models and full code generation with aspect-oriented techniques seems underrepresented in literature [8]. We showed that the utilization of traceability link relationships can be used for such cases.

Another tool that utilizes aspect-oriented techniques is Kieker [13]. While the tool provides a flexible architecture to add additional monitoring capabilities and an efficient runtime with minimal overhead, a model driven instrumentation using traceability information is currently not supported. As a result, reusability of generation logic is not fully supported at the degree that is theoretically possible. Two other tools, PEPP and AICOS [14], are specialized for performance measurements of parallel programs. These tools are bound to the programming language C and require separate models for the analysis (function program, functional implementation, monitoring and performance models) which limits the scope of the approach in terms of applicability to arbitrary meta-models.

There exist various model-to-text transformation languages that are comparable to our proposed template language, e.g., ATL, Epsilon, QVT, and Viatra [15]. While many model transformation languages provide meta-models for their traceability support, none of them makes advanced usage of the produced traceability information to detect non-trivial relationships between model elements. Although graph based transformation languages like Henshin [15] provide sophisticated mechanisms to extract relationships between model elements, they lack the ability to generate code from the matched patterns instead of creating and/or rewriting models according to specified rules.

VIII. Conclusions

In this paper we presented a language to specify recurring monitoring patterns which are automatically expanded into monitoring code for given models of the analyzed system. The language allows to define the steps of exploring a platform-specific target model for model elements and their relationships that fit the current analysis task at hand and offers a template based code generation mechanism to create the necessary monitoring code automatically. The exploration steps and writing the monitoring code would otherwise be done manually for every recurring analysis task. A case study showed that the effort for writing monitoring code is reduced and recurring analysis tasks are better supported through automatic code generation without disturbing the analyst with technical details of writing platform-specific monitoring code.

References

[1] M. A. Javed and U. Zdun, “A systematic literature review of traceability approaches between software architecture and source code,” in *18th International Conference on Evaluation and Assessment in Software Engineering (EASE 2014)*, May 2014.

[2] “Ieee standard glossary of software engineering terminology,” *IEEE Std 610.12-1990*, pp. 1–84, Dec 1990.

[3] M. Szvetits and U. Zdun, “Reusable event types for models at runtime to support the examination of runtime phenomena,” in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, Sept 2015, pp. 4–13.

[4] G. Blair, N. Bencomo, and R. B. France, “Models@ run.time,” *Computer*, vol. 42, no. 10, pp. 22–27, Oct. 2009.

[5] H. Giese, L. Lambers, B. Becker, S. Hildebrandt, S. Neumann, T. Vogel, and S. Wätzoldt, “Graph transformations for mde, adaptation, and models at runtime,” in *Proceedings of the 12th international conference on Formal Methods for the Design of Computer, Communication, and Software Systems: formal methods for model-driven engineering*, ser. SFM’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 137–191.

[6] H. Song, Y. Xiong, F. Chauvel, G. Huang, Z. Hu, and H. Mei, “Generating synchronization engines between running systems and their model-based views,” in *Proceedings of the 2009 international conference on Models in Software Engineering*, ser. MODELS’09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 140–154.

[7] S.-W. Cheng, D. Garlan, B. R. Schmerl, J. a. P. Sousa, B. Spitznagel, P. Steenkiste, and N. Hu, “Software architecture-based adaptation for pervasive systems,” in *Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing*, ser. ARCS ’02, Karlsruhe, Germany, 2002, pp. 67–82.

[8] A. Mehmood and D. N. A. Jawawi, “A comparative survey of aspect-oriented code generation approaches,” in *Software Engineering (MySEC), 2011 5th Malaysian Conference in*, Dec 2011, pp. 147–152.

[9] J. Zhu, C. Guo, Q. Yin, J. Bo, and Q. Wu, “A runtime-monitoring-based dependable software construction method,” in *2008 The 9th International Conference for Young Computer Scientists*, Nov 2008, pp. 1093–1100.

[10] K. s. Lee and C. G. Lee, “Model-driven monitoring of time-critical systems based on aspect-oriented programming,” in *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement*, June 2011, pp. 80–87.

[11] J. Bennett, K. Cooper, and L. Dai, “Aspect-oriented model-driven skeleton code generation: A graph-based transformation approach,” *Sci. Comput. Program.*, vol. 75, no. 8, pp. 689–725, Aug. 2010.

[12] K. Cooper, L. Dai, S. Dascalu, N. Mehta, and S. Velagapudi, “Towards aspect-oriented model-driven code generation in the formal design analysis framework.” in *Software Engineering Research and Practice*. Citeseer, 2007, pp. 628–633.

[13] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, “Continuous monitoring of software services: Design and application of the kieker framework,” Kiel University, Research Report, November 2009.

[14] R. Klar, A. Quick, and F. Soetz, “Tools for a model-driven instrumentation for monitoring,” in *Proceedings of the 5th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Citeseer, 1992, pp. 165–180.

[15] E. Jakumeit, S. Buchwald, D. Wagelaar, L. Dan, A. Hegedüs, M. Herrmannsdörfer, T. Horn, E. Kalnina, C. Krause, K. Lano, M. Lepper, A. Rensink, L. Rose, S. Wätzoldt, and S. Mazanek, “A survey and comparison of transformation tools based on the transformation tool contest,” *Sci. Comput. Program.*, vol. 85, pp. 41–99, Jun. 2014.