Advanced Typing for the Kieker Instrumentation Languages

Reiner Jung
reiner.jung@email.uni-kiel.de
Kiel University

Christian Wulf
chw@informatik.uni-kiel.de
Kiel University

Abstract

The observation of software systems is a complicated task due to the heterogeneity of technologies and programming languages involved. In Kieker , we address this heterogeneity with two domain-specific languages (DSLs) which allow to define event types and monitoring probes independent from specific languages. The DSLs allow to extend event types individually and to adapt probes accordingly.

In monitoring, different event types are used together to observe a specific property, like call traces. In case additional attributes, like message size, must be observed, multiple types must be extended simultaneously. This is cumbersome for large sets of types and an error prone task. In case of missed types or wrongly initialized attributes, the errors may harm analysis results. We address these challenges with a new type extension mechanism and semantic constraints for attributes.

1 Introduction

Kieker is a monitoring framework and tool to realize system and application monitoring. It has been used in a wide variety of projects, like DynaMod [5] and iObserve [6], and supports many different instrumentation technologies, like aspect-oriented or interceptor-based instrumentation. The different technologies require specific configurations and implementations of monitoring probes and data structures for monitoring events (event types), which result in a complex instrumentation task. To reduce the complexity, we supplemented Kieker with two languages: The Instrumentation Aspect Language (IAL) for probe and weaving specification and the Instrumentation Record Language (IRL) for event types [7].

Event types are record like data structures with attributes that are typed with base types (e.g., like int and long) [1, p. 117]. In addition, an event type can inherit attributes from another event type and multiple template types. In contrast, template types can only inherit from other template types and they cannot be instantiated, providing only an interface.

Kieker comes with a wide variety a pre-defined event types. For example, Kieker provides 32 different event types to monitor call traces covering many facets of trace information, like, object instantiations and exceptions. Lets assume we want to monitor a cloud storage application to assess its performance. The standard trace event types can only represent the trace itself, but are unable to collect the size of an upload request. However, to understand the execution time, we must extend the existing event types. Therefore, we must adapt the 32 event types and adapt all probes used for monitoring accordingly.

Due to the multi-inheritance type systems of the IRL, such extension could be specified within a template type RequestSize. Subsequently, all 32 event types must be either inherit the new template type, which may break existing analyses and requires to rebuild Kieker, or the event types must be subtyped and those types must inherit from RequestSize. This is a cumbersome and error prone task. In case we forget to extend an event type, like AfterOperationFailedEvent (logs exceptions), this can cause failures later during analysis, which makes it hard to debug.

Another issue arises from probe declaration and the initialization of event types. As attributes are typed, only compatible values can be assigned. Unfortunately, attributes with the same type can be confused when specifying probes. For example, BeforeOperationEvent has two string attributes for the signatures of the class and the operation. In case they are used in the wrong order, the code still compiles, but during analysis, the events contain incorrect data. Both shortcomings result in hard to detect errors.

We address these limitations with two additions to the IRL type system: (a) an extensions mechanism based on model types [2] which allows to group event types and extend them together with one declaration, and (b) semantic annotation which allow ot introduce further constraints on attributes and support automatic probe generation.

In the remainder of this paper, we introduce model types as foundations for the extension mechanism in Section 2. We discuss the extension of the type system in Section 3 and the semantic annotations in Section 4. Finally, we conclude and provide an outlook on the future development of IRL and IAL in Section 5.

2 Model Types

In modeling, models are sets of instances of classes [2]. A collection of classes is called a metamodel, and a model conforms to a metamodel, if all classes used

for the model are present in the metamodel. In case of metamodel evolution, metamodels are altered to support new and refined concepts. This may result in models which do no longer conform to the altered metamodels. Therefore, the models become inaccessible after an alteration. To address this issue, Steel and Jézéquel [2] introduce model types to ensure conformance. For the IRL, we will use model types to group event types which we intend to extend.

A *model type* is, similarly to a metamodel, defined as a set of classes. It can be seen as an interface declaration for a metamodel. In addition to metamodels, model types can be extended based on an unidirectional *match relation* $<\#$. This relation is comparable to a class subtyping relation. Two model types M' and M match if and only if each class C in M has a corresponding type C' in M' with the corresponding set of attributes (signature) [2]. Therefore, if $M'<\#M$ and $M<\#M'$ then M and M' must be identical.

Based on the concept of model types, we can derive a extension mechanism for IRL event types. A model type is then formed based on sets of event types and then subtyped to create new event types.

3 Composition of Event Types

The present IRL supports template and event types (cf. Section 1). Both types support multi-inheritance based on template types. To extend multiple event types together, we propose a model type based mechanism for the IRL, which consists of two declarations: (a) the declaration of the model type, and (b) the extension of a model type.

Model types As described in Section 2, model types are sets of classes. The template and event types of the IRL are like classes without operations and without references to other classes, like, aggregations and compositions. Therefore, we can apply the concept of model types to the IRL.

Due to the set character of model types, it seems obvious that a model type can be declared by enumerating event and templates types. However, this does not recognize the inheritance of the IRL type system which includes hierarchical subtyping and multi-inheritance. Therefore, a method is required to resolve both kinds of inheritance in context of model types: (1) The hierarchical subtyping of event types must be kept intact. This means if AbstractEvent is part of model type, and BeforeOperationEvent is a subtype of (<:) AbstractEvent then BeforeOperationEvent must be extended accordingly. Therefore, all subtypes of an extended event type must inherit the extension, as otherwise they are no longer subtypes [1, p. 182], resulting in incompatible model types. (2) template types form a directed acyclic inheritance graph. In case one template type belongs to the model type, all the inheriting types must also belong to the model type to satisfy the subtyping relation. This can result in a large set of types which are not directly mentioned in the enumeration of types. Developers may not be able to infer the extent of a model type they are specifying. Thus, we supplement the editor with a dedicated view depicting the complete model type. This allows to specify complex model types in a concise manner and still provide a comprehensive overview.

The syntax to specify model types (depicted in Figure 1) has a *name* and comprises a set of event and templates types, identified by their *type-name*.

⟨*ModelType*⟩ := '`model`' ⟨*name*⟩
⟨*type-name*⟩ ('`,`' ⟨*type-name*⟩)*

Figure 1: Grammar rule for model types

The corresponding semantics for this *ModelType* (depicted in Figure 2) comprises two rules: Subtype Inclusion defines, in conjunction with subtyping rules (cf. [1, p. 182ff]), that if a event or template type (*CType*) is element of *ModelType* then also the subtype T' is element of *ModelType*. Type Collection corresponds with line 2 of the grammar rule and defines the collection of all types in *ModelType*.

Subtype Inclusion
$$\frac{T' <: CType \qquad CType \in ModelType}{T' \in ModelType}$$

Type Collection
$$\frac{i = \{1 \dots n\} \quad CType_i \in M}{\{CType_1, \dots, CType_n\} \subseteq ModelType}$$

Figure 2: Semantic rules for the model type collection

Type extension The extension of a model type (cf. Figure 3) can be realized either by a collection of properties and constants, like a template and event type, or by specifying a set of template types. The latter allows to use template types to specify the extension. This has the benefit that template types can encapsulate concepts, e.g., request size, and name them to support a better understanding of the extension.

⟨*ModelSubType*⟩ := '`sub`' ⟨*name*⟩ ⟨*model-type-name*⟩
('`{`' (⟨*Property*⟩ | ⟨*Constant*⟩)* '`}`'
| ':' ⟨*templ-type-name*⟩ ('`,`' ⟨*templ-type-name*⟩)*)

Figure 3: Grammar rule for model subtypes

The semantics for the second grammar rule is based on record types with subtyping [1]. The property collection (line 3) for the extension is handled like in record properties, and the inherited properties are inferred via subtyping and model matching.

4 Semantic Annotations

The IAL requires to explicitly specify which Observed Attribute (OA) is assigned to a specific Event Type

Attribute (ETA) declared with the IRL. Each ETA typed to ensure a type safe assignment of its associated OA (in the IAL). However, different ETAs share the same type, and, therefore, the two languages, IRL and IAL, cannot ensure that an OA is assigned to the correct ETA. For example, the class and operation signatures of the BeforeOperationObjectEvent can get confused when specifying probes with the IAL. Therefore, additional information is required to ensure the correct attribute assignments. This can be realized either with specific datatypes or semantic annotations.

While specific datatypes for specific OAs provide a concise notation, they have two disadvantages: (a) they obfuscate the syntactical type nature of the attributes which have a different impact on the overall event size. (b) the number of base types must be extensible to support new attribute semantics, which makes type inference more complicated. In addition it induces compatibility issues with existing event type structures. Therefore, we use annotations which can be added dynamically via an extension mechanism without interfering with the type system (cf. Figure 4). These extensions provide annotations identified by a unique name and comprising code snippets for different programming languages and technologies[1]. For example, there must be different code snippets for the collection of the operation signature for AspectJ [3] and Servlet listeners [4].

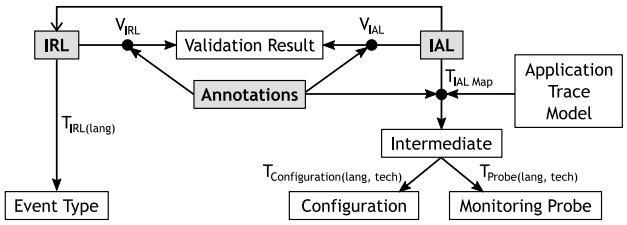

Figure 4: Utilizing of annotations in the IRL and IAL code generation and validation (depicted in gray)

The megamodel (Figure 4), illustrates where the annotation information is used in the generators and editors for event types, monitoring probes (advice) and configuration (pointcut). The transformations participating in code generation are labeled as T and validation transformations of editors are named V. The subscripts indicate the specific transformation task. The suffixes lang and tech indicated that there are different transformations for each programming language and technology. As Figure 4 shows, in context of the IRL the Annotations are only used to validate whether the used annotations in an IRL artifact exist and that their types match. Hence, the IRL can still generate event types for programming languages providing backward compatibility even if certain annotations are not available for a specific language. In contrast, the IAL uses the annotation code snippets during code generation to compose monitoring probes.

[1] IRL and IAL syntax and generators `https://github.com/kieker-monitoring/instrumentation-languages.git`

5 Conclusion

We presented in this paper two extensions to the IRL related to the type system. First, we introduced a mechanism to extend multiple event types together based on model types, reducing the effort necessary to extend Kieker event types. Second, we discussed semantic annotations for event type attributes, which provide constraints for event type construction and the automatic generation of probes.

While the extended typing of the event types and the annotations reduce the effort of instrumenting software systems, they still have limitations. The present naming schema for the model type based mechanism may result in shadowing of types, e.g., if multiple packages provide types with the same simple name. In future, it must be determined whether this limits the application of model types in Kieker.

Presently, the annotations must be implemented and specified as an Eclipse extensions which requires boiler-plate code and definitions. Therefore, we intend to create a DSL to specify annotations and code snippets to mitigate the construction of extensions.

Acknowledgement This work was supported by the DFG (German Research Foundation) under the priority program SPP 1593: Design For Future – Managed Software Evolution (grants HA 2038/4-1).

References

[1] B. C. Pierce. *Types and programming languages*. MIT Press, 2002, pp. I–XXI, 1–623.

[2] J. Steel and J.-M. Jézéquel. "On model typing". In: *Software & Systems Modeling* 6.4 (2007), pp. 401–413.

[3] R. Laddad. *AspectJ in Action*. 2nd ed. Manning, 2009.

[4] R. Mordani. *JSR 315: Java Servlet 3.0 Specification*. Java Specification Request. Oracle, 2009.

[5] A. van Hoorn et al. "DynaMod Project: Dynamic Analysis for Model-Driven Software Modernization". In: *Joint Proceedings of the Int. Workshops on Model-Driven Software Migration and on Software Quality and Maintainability*. 2011.

[6] W. Hasselbring et al. *iObserve: Integrated Observation and Modeling Techniques to Support Adaptation and Evolution of Software Systems*. Technical Report. Kiel University, Oct. 2013.

[7] R. Jung, R. Heinrich, and E. Schmieders. "Model-driven Instrumentation with Kieker and Palladio to forecast Dynamic Applications". In: *Proceedings Symposium on Software Performance: Joint Kieker/Palladio Days 2013*. Vol. 1083. CEUR, Nov. 2013, pp. 99–108.