CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Model-driven Performance Measurement and Assessment with Relational Traces

Dissertation zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

Dipl.-Ing. Marko Bošković

Referees:
Prof. Dr. Wilhelm Hasselbring
Prof. Dr. Claus Möbus
Datum der Disputation: December 10th, 2009

Contents

List of Figures

List of Tables

Abstract

Model Driven Engineering is an emerging approach in software development which argues using models as primary software artifacts. The models can vary from prescriptive ones specified in Domain Specific Modeling Languages (DSMLs), for the domain specific software development (e.g. Graphical User Interfaces DSML), to descriptive models for analysis, such as queuing networks and Petri-nets. The main benefit of such approach is that it reduces the time and effort spent in implementation by transforming the domain specific prescriptive models to the code. Furthermore, performance characteristics of the implementation can be predicted during the design phase with usage of transformation of domain specific constructs to formal mathematical models.

For software performance, as an important software Quality of Service attribute, several approaches for prediction exist in MDE. However, MDE still lacks a systematic approach for performance measurement and metrics assessment. This thesis presents ***MoDePeMART***, an approach for Model Driven Performance Measurement and Assessment with Relational Traces. The approach suggests declarative specification of performance metrics in a domain-specific language and the usage of relational databases for storage and metric computation. The approach is evaluated with the implementation of a UML Profile for UML Class and State diagrams and transformations from the profile to a commercial relational database management system.

Zusammenfassung

Model Driven Engineering (MDE) ist eine neue Vorgehensweise in der Software-Entwicklung, die Modelle als primäre Software-Artefakte verwendet. Die Modelle variieren zwischen präskriptiven Modellen, die spezifiziert werden durch domainspezifische Modellierungssprachen (DSML) und für domainspezifische Software Entwicklung (z.B. DSML für graphische Benutzeroberflächen) verwendet werden, und deskriptiven Modellen für die Analyse von z.B. Warteschlangen und Petrinetzen. Der größte Vorteil einer solchen Vorgehensweise ist, dass sie die Zeit und den Aufwand bei der Implementierung durch Transformation der domainspezifischen Modelle in den Quellcode reduziert. Darüber hinaus können Leistungsmerkmale der Implementierung während der Entwicklung durch die Nutzung von Transformation domainspezifischer Konstrukte in formale mathematische Modelle vorhergesagt werden.

Für die Vorhersage der Leistungsfähigkeit von Software, ein wichtiges Qualitätskriterium von Software, gibt es im MDE verschiedene Ansätze. Allerdings fehlt im MDE bisher ein systematischer Ansatz zur Performanzmessung und Bewertung von Kennzahlen. Diese Arbeit präsentiert ***MoDePeMART***, ein Ansatz über Model Driven Performance Measurement und Assessment mit relationalen Traces. Der Ansatz schlägt die deklarative Spezifikation der Performanzkennzahlen in einer domainspezifischen Sprache und die Nutzung von relationalen Datenbanken für die Speicherung und metrische Berechnung vor. Der Ansatz wird mit der Umsetzung eines UML-Profils für UML-Klassen- und Zustandsdiagramme und Transformationen aus dem Profil zu einem kommerziellen relationalen Datenbank-Management-System evaluiert.

Acknowledgments

A PhD thesis is a work in which credits takes only the author. However, conducting a research and writing such work is impossible without support of a large group of people.

First of all, I am indebted to my research supervisors Prof. Dr. Wilhelm Hasselbring and Prof. Dr. Claus M¨obus for their guidance, encouragement, useful discussions and valuable suggestions throughout this investigation. I am thankful to Prof. Dr. Möbus for initiating my research. I express my gratitude to Prof. Dr. Hasselbring for his friendly and permanent advice, meticulous guidance, pertinent encouragement, useful discussions and valuable suggestions when my research went into the direction of Software Engineering. I am also thankful to for providing a creative and friendly atmosphere in the Software Engineering Department at the University of Oldenburg.

I own my gratitude Prof. Hasselbring also for organizing the TrustSoft Research Training Group, and accepting me as a scholarship holder of that school. TrustSoft Research Training Group enabled the financial support for carrying my research out. Student and supervisor members have given me many useful pieces of advice, interesting and valuable discussions and suggestions. I also express my gratitude to Manuela Wüstefeld and Ira Wempe for being more than secretaries in the department and giving me friendly pieces of advice and help.

I acknowledge all students at the TrustSoft Research Training Group for constant support, patience, useful suggestions, interesting and motivating discussions, and over all for simply being my honest friends during my life in Oldenburg. Their friendship has meant and still means a lot to me.

My acknowledgments also extend to friends in Oldenburg which were neither members of Software Engineering Department nor TrustSoft Research Training Group. I am particularly grateful to "ex-Yugos and all that like them" group of friends gathered around Department of Slavistics, "Flying Burex" soccer team and "Sputnik Radio" radio show with DJ Geistarbeiters.

At the end, I express my honest gratitude to my parents, family members, and all my professors and friends in Belgrade, for understanding, patience, motivation, constant and extensive support and for believing in my success.

Vielen Dank für Alles!
Thank you for everything!
Puno hvala na svemu!
Marko

1. Chapter Introduction

Software systems are increasingly becoming an important part of current technical systems, from the smallest devices in everyday life, e.g. digital watches and television sets, through business information systems, e.g. government and banking Internet applications, to systems with large complexity like embedded systems in nuclear power plants. Consequently, the world is becoming gradually dependent on the technological systems controlled by software. In such an expansion of dependency, software failures can have unpleasant, costly, harmful or even disastrous consequences.

The growth of dependency on software systems, and consequences of their failures, forces government, academia and industry to raise the question of software intensive system trustworthiness [Hasselbring and Reussner, 2006]. With the aim to use software intensive systems as dependable systems, means for quantification, verification and contractual trust of those systems have to be invented.

A trustworthy software system has to satisfy not only expected functional but also non-functional requirements, often called quality requirements [Hasselbring and Reussner, 2006]. With the aim to be useful for the intended purpose, software has to provide required services. The required services are known as functional requirements, because they define functionality which is the objective of the system. Non-functional requirements are constraints on system's functionality offered by the system like security, privacy, reliability, timeliness etc [Sommerville, 2007]. They are characteristics of functionality design and implementation. Different implementations of the same functionality can have different non-functional properties.

Significance of meeting non-functional requirements in trustworthy software intensive systems development, requires addressing them in the early design phases, in parallel to functional requirements. Not meeting functional requirements means having a system which does not provide the requested service. Nevertheless, not meeting the non-functional requirements can also lead to designing and implementing an unusable software system. For example, if a response time of a web server is too high, in the order of one hour, it would not be usable at all, regardless that it performs functionality. Often, in the process of software development, functional requirements are considered as the primary target, and the problem of meeting non-functional requirements is addressed in the later phases of a software design. This "fix it at the end" approach is not satisfying. Generally, reasons for not meeting non-functional requirements, like for example performance, lie in neglecting them at the early development phases. Problems of not meeting non-functional requirements cannot be later solved with tuning [Smith and Williams, 2001].

Some of the non-functional properties of a service, of particular interest to users, are often specified with the Quality of Service (QoS), and in this thesis, it is considered that QoS consists of three attributes [Hasselbring and Reussner, 2006]:

- Availability—probability of readiness for correct service,
- Reliability—probability of correct service for a given duration of time,
- Performance—degree to which objectives for timeliness are met [Smith and Williams, 2001]. It describes timing behavior of a software system and it is measured with metrics like throughput and response time.

Model Driven Engineering (MDE) [Kent, 2002] is an approach which tries to handle both, functional and non-functional requirements in all stages of software intensive system development. It is an approach for software engineering which suggests building families of related languages and processes for software development. Languages can be either for prescriptive or for descriptive models [Seidewitz, 2003]. Prescriptive models specify desired systems and their behavior, like for example modeling languages from which code implementation is automatically generated [Kelly and Tolvanen, 2008]. Descriptive models abstract the system for the purpose of deducting new statements about the potential behavior. For example in queuing networks [Gross and Harris, 1985] model software system entities are represented as a set of related servers with waiting queues of requests, with the purpose of performance analysis and prediction. The MDE software systems engineering approach combines usage of these two model kinds to address both, functional and non-functional requirements at all stages of software development [Selic, 2003]. Strong reliance on the set theory [Favre, 2004], graph and graph transformation theory [Mens et al., 2006], promotes MDE as a promising attempt for making software development a mathematically founded engineering discipline. Furthermore, Domain Specific Modeling Languages [Kelly and Tolvanen, 2008], languages for solving recurring problems of a particular domain, which are the central idea of MDE moves software development from implementation toward the domain the software intensive system is intended for. Examples of domains are insurance systems and call processing [Kelly and Tolvanen, 2008].

With the purpose to provide legally trusted services, their QoS is commonly being contractually specified with Service Level Agreements (SLA) [Bouman et al., 1999] between the service provider and service consumer. The QoS are dependent on the service consumer. For different consumers, and different domains required values of QoS attributes can vary. What is considered as acceptable in one domain can be unacceptable in another one. For example, performance in embedded systems, like air bag in cars, is certainly more restrictive then performance of large bank applications. While in air bag, response time of the system which blows air always has to be between predefined lower and higher thresholds, in large bank applications response times can be below some threshold as long as the number of responses which are below is large enough. Furthermore, an SLA defines proper service usage in which the agreement holds.

Contractual agreement on provided service quality requires verification of the quality observed. During the design of a software system, estimation of non-functional properties is often done. Nevertheless, at the end, when the software system is finally implemented and deployed, performance predictions have to be empirically proven and reached quality level validated.

Empirical verification requires measurement and assessment of quality properties of interest. With measurement, standing for collection of data about execution, and assessment,

meaning computation of metrics and reasoning about fulfillment of non-functional requirements, non-functional properties are made observable, quantifiable and comparable. With the aim of performing measurements, additional pieces of code, called software probes, need to be added. The process of software probes insertion is called instrumentation. Probes, during execution at run-time, collect the data about the system execution. This data is later stored and analyzed, and finally the decisions on the fulfillment of non-functional requirements are made.

Instrumentation can be very hard and error prone. Implementation of functional requirements already makes software systems complex. With the addition of probes, programs become even more complex and require significant effort for splitting code for functionality implementation and probes. Furthermore, with growth of complexity and size of the functionality implementation code and number of measurement points and granularity of data collected, the probability of faults in the code for data collection also raises. Moreover, instrumenting software systems at the points of interest alone not only reduces the probability of error, but reduces the amount of collected data for later analysis to the needed, and perturbation of the system due to measurements. Finally, after the assessment, faults in the system can occur as a result of incomplete probes removal.

Not only instrumentation, but the data storage, with the purpose of metrics computation for analysis, and analysis itself require significant effort. Execution data storage raises problems of the appropriate granularity of collected data. Appropriate granularity stands for the minimal amount of data needed for uniquely describing an event at one hand and sufficing the appropriate amount of data for analysis on another. Furthermore, it raises the problem of structuring data for later analysis. For the reason of analyzing the stored data, they have to be stored in a way that enables a reconstruction of execution and what is more important, computation on metrics of interest.

Designing the code for instrumentation, execution data storage, and data analysis have to be done in synergy. Data granularity and the number of measurement points are strongly related to both instrumentation code and storage data structures, as well as the code for analysis. Not designing them together can lead to inappropriate insertion of measurement points and amount of collected data. Furthermore, it can lead to impossibility or failures of analysis.

The central topic of this thesis is integration of performance measurement and assessment in the Model Driven Engineering approach. It complements to model-based software estimation [Balsamo et al., 2004], where software design models are transformed into simulation and analytical prediction models. While estimation of software performance is very important for guiding the design in such a way that the early design decision are made by paying attention to non-functional requirements, in the final phase, the predictions have to be empirically verified and the satisfaction of non-functional requirements validated.

The main contributions of this thesis are a process, definitions of a metamodel of software performance measurement and assessment, transformations from the metamodel to Libkin's algebra [Libkin, 2003] as an abstraction of Relational Database Management Systems (RDBMS) [Atzeni et al., 1999]. Furthermore, the contributions are elements of evaluation, and those are implementations of the metamodel in the Model Driven Architecture (MDA) [Miller and Mukerji, 2003] approach for MDE. The approach is integrated in MDA in the form of a UML Profile [Object Management Group, 2007], and implementation of coupled transformations from this profile to commercial RDBMS and execution platforms.

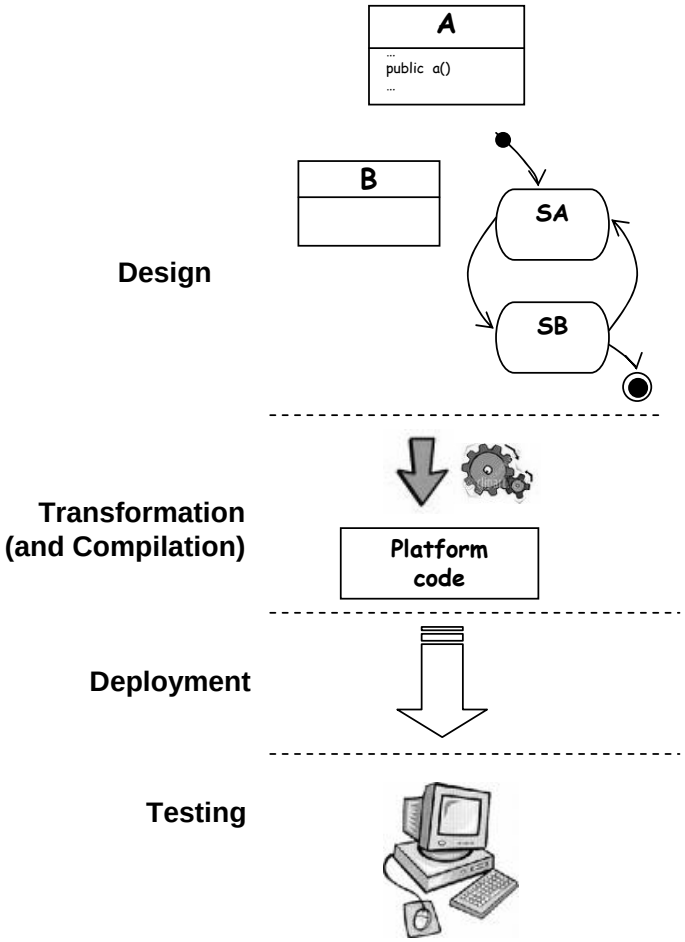


Figure 1.1.: Software development process in Model Driven Engineering

The explanation of integration in MDA is given in more detail is in the following.

The software development process in MDE is presented in Figure 1.1. Unlike the current practice, where the focus of software development is code, in Model Driven Engineering, the focus are models. Through families of languages, development is model-oriented, from the requirements to the final design. The requirements are collected at the top level, and software is designed through a process of refinement. At the final phase of development, the final model is designed. The final model is later transformed into the execution platform code and then compiled in executable code. Finally, the code is deployed and the application tested.

So far, in order for performance to be empirically evaluated, the developer would have to either add additional code, or use some tools for platform instrumentation. Nevertheless, this is not appropriate, because the constructs at the model level can be significantly different from constructs used in a particular execution platform. For example, a platform like Java [Eckel, 2002] does not support constructs like states, and if the code is produced from statecharts [Samek, 2002], it has to be implemented with a pattern of constructs allowed

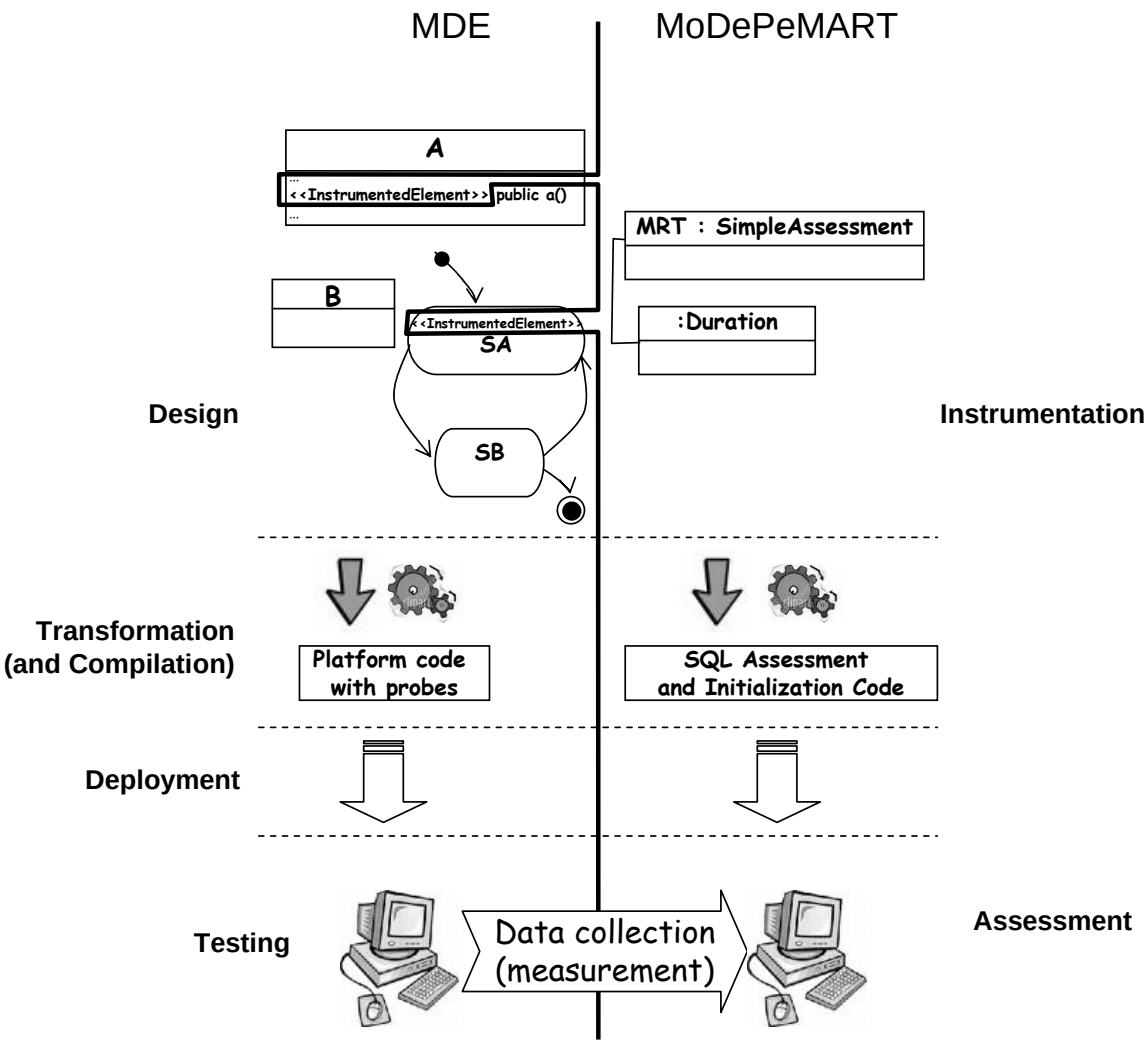


Figure 1.2.: Software development process with performance measurement and assessment in Model Driven Engineering

by the Java the platform. The case becomes even more complicated with Domain Specific Modeling Languages. In those cases, the developer or performance analyst, in order to appropriately collect the data about the timing behavior of a model driven developed application, has have to have the knowledge of both: the target platform and transformation. Furthermore, if there are multiple alternatives for code generation [Steimann and Kühne, 2005] the instrumentation and the performance assessment becomes even more complicated and can cause faults. For example, in the case of statecharts there are at least three alternatives, each having its own benefits and drawbacks in different non-functional properties [Samek, 2002]. In such a case the developer would have to know all alternatives, and the places where each of them is applied.

The ***Mo***del ***D***riv***e***n ***Pe***rformance ***M***easurement and ***A***ssessment with ***R***elational ***T***races (MoDePeMART) is the proposed process of software verification, presented in Figure 1.2. It extends the process of Model Driven Engineering with an activity in which a software developer or a performance analyst specifies in a declarative way the desired metrics and process

of interest and leaves to a transformation the joint production of code for instrumentation, data storage and computation of metrics. Furthermore, it extends the family of languages with a language for declarative performance metrics specification.

This approach integrates in MDE at the stage where the final design model is produced. In all phases of software development, as discussed before, predictions for software performance are made. The more the design model is refined, the more accurate the predictions are. Finally, predictions have to be empirically verified. This is left to be done, after the complete software system is implemented, and deployed. In MDE the complete implementation is done at the design phase where the final model, the model from which the code is produced, is developed. The Model Driven Performance Measurement and Assessment with Relational Traces defines this stage as a point in which performance measurement and assessment should be specified. After the specification, the code with probes for the target platform is generated from the model. It is later compiled, if needed, and deployed. In parallel, relational queries for performance assessment are also generated and deployed. The measurements are performed during execution of test cases intended for the application. After the data collection, during performance assessment, the assessment queries are simply executed and the developer or the performance analyst has only to see their values and reason for the fulfillment of defined requirements.

The MoDePeMART suggests a design language extension for instrumentation and attachment to the metamodel, an abstract syntax of the language for assessment specification. The extended modeling elements are instrumented elements and serve as an input to the declarative specification of metrics and execution context of interest in the language defined by the metamodel. The examples of metrics are mean, median, and standard deviation of response time.

Execution context is the set of past, current, and future events related to the event observed. The metamodel is defined with the assumption of a language supporting development of software intensive systems with combined reactive and transformational parts [Wieringa, 2003]. The transformational part is a part of the system that simply takes the input and transforms it into the output. Their output is always the same for the same input. Example of these kind of systems are mathematical libraries. Mathematical libraries take some value and transform it into another value according to the function they perform like for instance computing sinus of a value. Reactive systems are the systems which are in the constant interaction with their environment. Response to stimulus from the environment in this case does not depend only on the stimulus, but also on the previous environment stimuli which define the current state of the system. For this reason, the definition of execution context is the definition of a sequence of calls and state of the system in which an event of interest is observed.

In order for the performance to be evaluated, the developer, or performance analyst, has just to annotate the events of interest, with elements of extension and specify in which metrics and execution context he is interested. The code generation for measurement and assessment is automatically generated by transformation, and the rest of the measurement and assessment process follows as previously described.

The transformation specifies the code generation from the metamodel to the Structured Query Language (SQL) [Atzeni et al., 1999]. Furthermore, it gives formal semantics to the metamodel with Libkin's algebra. Libkin's algebra was invented to prove inability of expressing recursive queries in relational algebra. Moreover, it is an algebra with the open

set of aggregate functions. In this thesis the second characteristic is of main interest because with the specification of a transformation to Libkin's algebra, we relate statistical analysis provided with the metamodel, to the set of requested aggregate functions which the RDBMS has to support.

The evaluation of the approach is given as a proof of concept in the context of MDA. The MDA is an approach for MDE recommended by the Object Management Group (OMG), a non-for-profit consortium of members from industry and academia.

The MDA suggests a family of three modeling languages for [Miller and Mukerji, 2003]:

- Computation Independent Model (CIM), where the vocabulary familiar to the domain practicioners is used,
- Platform Independent Model (PIM), which is a design model independent of different platforms of similar type, and
- Platform Specific Model (PSM), which combines PIM with the characteristics of a particular platform. Often the PSM is not needed and code for a particular platform is automatically generated from PIM [Hubert, 2002], [Raistrick et al., 2004].

The MDA also suggests a high reliance on open standards [Booch et al., 2004]. In Computer Aided Software Engineering (CASE), predecessor of MDE and MDA, a large problem was interoperability of tools. A particular language or method was strongly related to a particular tool. Furthermore, a model designed in one tool could hardly be used with another tool, even if they were using the same notation. The MDA overcomes this problem with reliance on standards. Some of the important OMG standards are the Unified Modeling Language (UML) [Object Management Group, 2007] and the XML Metadata Interchange (XMI) [Object Management Group, 2005b], a standard for persistence and exchange of UML models.

Although the MDA does not specify the obligation of usage of the UML, it is de facto used as a basis for modeling. With the UML Profile, a mechanism for extending the original UML metamodel for a particular domain, it can be used as a basis for all three modeling languages.

In the evaluation of this thesis a part of UML as the software modeling language is used. The part of UML is concentrated on class and state diagrams as a basis for an executable PIM [Raistrick et al., 2004]. Furthermore, the metamodel for performance measurement and assessment is implemented with the means of the UML Profile. We provide the implementation of the UML Profile for ***PE***rformance ***M***easurement and ***A***ssessment (PEMA) in the MagicDraw [Neuendorf, 2006] tool for UML modeling. Moreover, implementation of the transformation from the design model and performance assessment model to target platform code, including probes, and RDBMS queries for assessment is also provided. The approach is evaluated for one threaded desktop application with transformation to plain Java, and for multi-threaded Servers to Java with RMI [Grosso, 2001]. Assessment queries are generated for MySQL [Tahaghoghi and Williams, 2006] an open source RDBMS. The transformations are implemented using openArchitectureware [Markus Völter, 2006], a set of languages and an environment model transformation.

This thesis is structured as follows. Part I is dedicated to foundations of approach. In this part in Chapter 2 the basics of Model Driven Engineering are clarified. Section 2.1 gives

the general ideas and benefits of the MDE software engineering approach. The central topic in MDE is engineering and using a language, and Section 2.2 explains what is needed to be specified when creating a language. Finally, Section 2.3 describes principles and technologies of Model Driven Architecture (MDA), one of the mostly used approaches for MDE.

After introducing elements of MDE, Chapter 3 explains software performance engineering. First, this chapter defines in Section 3.1 the software performance and performance metrics. Section 3.2 gives an overview of the approaches for software performance evaluation. Furthermore, Section 3.3 describes in more detail measurement and assessment, one of the approaches for software performance evaluation. Finally, Section 3.4 explains important issues in software performance measurement and assessment, which are later used for comparative analysis to related work.

One of the major shortcoming of many languages in the past is that they did not have a formal semantics. With the formal semantics the language constructs have the precise meaning and reduce abuses of the language. The formal semantics is usually given with mappings to some formal mathematical domain. To the language shown in this thesis formal semantics is given with mapping to Libkin's algebra. Chapter 4 defines this algebra.

Part II is dedicated to the main contributions of the thesis. Chapter 5, gives a detailed description of the ***MoDePeMART*** measurement and assessment process. Then, Chapter 6 explains the basic assumptions of the approach. This chapter explains in Section 6.1 transformational and reactive software intensive systems in more detail. Furthermore, it explains characteristics of these systems' modeling languages. Finally, it gives the basic assumptions on the relation between a model and the executing system in Section 6.2.

Chapter 7 provides detailed description of a metamodel for performance measurement and assessment. The metamodel consists of three parts. The first part of the metamodel represents performance metrics which can be specified for computation. This metamodel part is described in Section 7.1. Section 7.2 describes kinds of assessment which can be carried out with this metamodel. After the description of assessments supported by the metamodel, Section 7.3 explains the metamodel part used for specification of subjects of assessments, i.e., events in particular contexts. Finally, Section 7.4 of this chapter describes rules needed to be satisfied for having a well-formed model.

Chapter 8 defines formal semantics of this metamodel. It is given with a mapping to the Libkin's algebra. The specification of formal semantics goes in the opposite way to introducing the concepts of the metamodel. First, in Section 8.1 are defined prerequisites of the used RDBMS and its initialization. Then, in Section 8.2 are given formal semantics to the metamodel part for specifying events and their contexts. This formal semantics is followed by formal semantics of the metrics and the assessment metamodel parts in Section 8.3.

Evaluation of the approach is provided in Part III of this thesis. Chapter 9 documents the implementation of the approach as a UML Profile. In Section 9.1, a subset of the UML metamodel used in the evaluation is explained. Next, Section 9.2 provides the description of the PEMA profile, an implementation of the metamodel defined in Chapter 7. Finally, Chapter 10 gives and validates the transformation to the target platform. The target platform is Java with RMI for multi-threaded Client Server architectures and the transformation is described in Section 10.1. Section 10.2 validates the implemented tool and case study used. The final chapter of this part, Chapter 11 gives a comparative analysis of the ***MoDePeMART*** with other approaches for measurement and assessment.

The final part, Part IV, in Chapter 12 summarizes results of the thesis and explains how

valid this approach is for use in real systems. Finally, Chapter 13 indicates some promising directions for the future work.

Part I.

Foundations

2. Chapter
Model Driven Engineering

Model Driven Engineering (MDE) is a new paradigm for software engineering. It has evolved from Computer Aided Software Engineering (CASE). CASE is a software engineering approach in which sets of tools and methods are scientifically applied for software production. CASE uses software tools for the production of software in the same way as software tools are used for computer aided design in civil engineering and architecture. However, the MDE goes one step further. While in CASE software implementation is produced independent of the software model developed in a CASE tool, in MDE from these models full application code is generated. This chapter explains MDE in more detail and it is structured as follows. Section 2.1 explains the essentials of MDE. Models in MDE are specified in modeling languages, and engineering languages are described in Section 2.2. Finally, one of the major approaches in MDE is Model Driven Architecture and it is elaborated in Section 2.3.

2.1. Essentials of Model Driven Engineering

The central idea of Model Driven Engineering is that software engineering should use models for the complete specification of systems. The benefits of usage of models as primary artifacts has already been seen in other engineering disciplines, like civil and electrical engineering. Some of benefits of using models in software engineering are the following [Selic, 2006b]:

- Better understandability of software products to domain experts.
  One of the major problems of software engineering is specification of software requirements gathered from the domain experts, and their later implementation. The problem lies in the fact that the terminology and concepts used by the domain experts are not familiar to software engineers. Furthermore, the terminology and concepts used by software engineers are also not known to the domain experts. This gap in the communication is cause of many failed projects, because often the requirements are not implemented in the way they should. In MDE, this is solved bu usaging Domain Specific Modeling Languages (DSMLs). A DSML contains the terminology of the domain of interest, for example insurance. A domain expert uses DSML to specify his requirements. After the specification, the code and/or data for the domain expert requirements computation is automatically generated.

- Impedance of formal mathematical methods exploitation
  Formal mathematical methods are well applied in verification of software system properties. Some of the properties which can formally be verified are correctness, performance, reliability, safety, security, and several more. MDE suggests usage of formal methods in engineering DSMLs. The formal methods are integrated in such a way that

they are hidden from the expert of the domain of DSML. For example, let us have a software for an insurance company. And let that software be specified in a DSML for the insurance companies. Now, let us suppose that there is a need for performance prediction of a software specified in the insurance DSML. The insurance domain expert will not be aware of the formality used for the performance prediction. Such formality can be for example a queuing networks.

- Avoidance of state explosion problems
  In the usage of formal methods, all possible states of the software are often analyzed. This can cause the state explosion. With abstraction in DSML, state explosion problems can be avoided.

- Shorter time-to-market of the trustworthy software
  All previous benefits reduce the time to market. Furthermore, they improve the software development process so that both functional and non-functional runtime software system properties are predicted.

Model Driven Engineering exploits the fact that the specialization is a characteristic of our time. Domains of interest and activities continuously brunch into more specialized sub domains. In MDE such a process takes place in modeling languages also. Modeling languages constantly move from general purpose to domain specific.

Specialization from general purpose to domain specific languages are based on two already used and time-proven principles in compiler theory. Those principles are [Selic, 2006b]:

1. Abstraction
   Abstraction is used in the definition of DSMLs. With abstraction the domain of interest is conceptualized and specified in a language. The conceptualization of the domain should enable specification of solutions to problems in that domain.

2. Automation
   Automation is generation of a software product with usage of transformations from DSMLs to some implementation technology. Automation exploits the rare property of software that it allows direct evolution of models into complete implementations without discontinuities in the expertise, materials, tools, or methods. Transformations are written by a transformation writer. A transformation writer is an expert in the implementation technology who knows well the concepts of the domain of DSMLs [Steimann and Kühne, 2005].

Moving from general purpose languages to DSMLs causes also the multidimensional software development. In multidimensional software development each concern of interest is a separate dimension and it is modeled as a separate modeling language [Kent, 2002]. Examples of modeling dimensions are requirements, design modeling dimensions, but also concurrency, security, logging. The modeling languages of different dimensions either intersect, which is that they model a separate concern of one model, as for example control flow and security, or are connected with transformations from one to another, as for example from requirements to design model.

In this thesis it is assumed that the performance measurement and assessment is one of the dimensions in the software development process, and for that reason needs to be supported with a domain specific language.

The main artifact of MDE are, as already stated, models specified in some modeling language. The next section describes engineering a modeling language.

2.2. Engineering a Modeling Language

A definition of a modeling language consists of [Kühne, 2006]:

1. Abstract syntax
   Abstract syntax of a language defines the concepts of the language. It is independent of machine- and implementation-oriented structures and encodings. In this thesis the abstract syntax of the language for performance measurement and assessment is specified in Chapter 7.

2. Concrete syntax
   Concrete syntax of a language specifies rendering of concepts from the abstract syntax. It defines how the modeling constructs defined in the abstract syntax are visually represented. It is the actual implementation of the abstract syntax in the modeling tool. A modeler uses the concrete syntax to specify a model. Concrete syntax in this thesis is given in the form of a UML Profile, and it is described in Section 9.

3. Static semantics
   Static semantic defines well-formedness constraints. It defines structural constraints which cannot be adequately captured by a syntax description. Examples of well-formedness constraints are need for declarations of variables before their usage. Static semantics specify constraints which cannot be given in the abstract and concrete syntax specifications. The static semantics of the language for measurement and assessment is specified in Section 7.4.

4. Semantics of the language
   Semantics of a language is defined by the specification of mappings of concepts from the abstract syntax to a well-known target model. Semantics can be given to a language in several ways. It can be given with the description in a natural language. For example, executable semantics in UML is given by natural language. However, there are formal approaches for specifying semantics. In formal specification of semantics, abstract syntax is mapped with means of mathematical function to some already defined mathematical domain. Examples of mathematical domains are Petri-Nets, Queuing Networks, and algebras. In this thesis the semantics is given twofold. First, semantics to the domain of performance engineering is given with natural language in Chapter 7. Second, the semantic of the language with respect to computation is given with formal mappings to Libkin's Algebra and it is specified in Chapter 8.

Abstract and concrete syntax are also known as linguistic metamodels [Atkinson and Kühne, 2003]. Beside the linguistic metamodels, which deal with the representation of the

domain concepts, exist ontological metamodels. Ontological metamodels conceptualize and structure the domain of interest. This thesis does not provide the ontological metamodel of the domain of performance engineering. It only specifies the linguistic metamodel of the language for performance measurement and assessment.

The concrete syntax of the language for performance measurement and assessment is given with the purpose of integration in the Model Driven Architecture, one of the major approaches for MDE. Model Driven Architecture is explained in the next section.

2.3. Model Driven Architecture

The Model Driven Architecture (MDA) [Miller and Mukerji, 2003] is one approach for MDE initiated by the Object Management Group (OMG). OMG is a consortium of software vendors and users from industry, academia and government. It is a conceptual framework which defines three viewpoints on a software model: Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM).

A CIM focuses on software requirements and does not care about the system structure. It should be written by domain experts and not by computer specialists. Therefore, the CIM is usually written in some Domain Specific Language.

A PIM specifies the implementation of required functionality without details of implementation in some particular technology. The current initiative for the platform independent modeling language is Unified Modeling Language [Object Management Group, 2007].

The full specification of the system is given by a PSM. A PSM is written in terms of some exact technology like .NET, Java, or CORBA. This kind of model comprises PIM with characteristics of the implementation platform. Source code is also considered as a PSM.

From the language engineering perspective, OMG's MDA is based on a four layer linguistic metamodeling architecture and suggests several adopted standards. The standards on which MDA relies on are the Unified Modeling Language (UML) [Object Management Group, 2007], XML Metadata Interchange [Object Management Group, 2005b], and Meta Object Facility (MOF) [MOF, 2004]. A profile is also a standard mechanism for specialization of UML for the purpose of some specific modeling [Miller and Mukerji, 2003]. The standards and the architecture are presented at Figure 2.1.

On the top of the architecture is the Meta Object Facility (MOF). MOF is an abstract language for specifying metamodels. It is also called metametamodel. One or several metamodels can be used to define a modeling language, like the UML. Furthermore, MOF specified metamodels also can be used to extend existing metamodels, and modeling languages. Metamodels are placed on the M2 level.

Modeling languages defined on the M2 layer are used to model real world things which are on the M0 layer. Models of real world are placed on the M1 layer. As an example UML can be used. The UML concept of a Class is defined on the M2 layer with a MOF language which is on the M3 layer. The concept of a Class on the M2 layer is used to model some real world entity, like a Person, which is on the M0 layer, and represent it with an element of type Class named Person in a model on the M1 layer.

Generally, there are three ways to define a modeling language: definition of a new modeling language from scratch, extension of an exiting language with domain specific concepts, and specialization of the concepts of an existing language [Selic, 2006a].

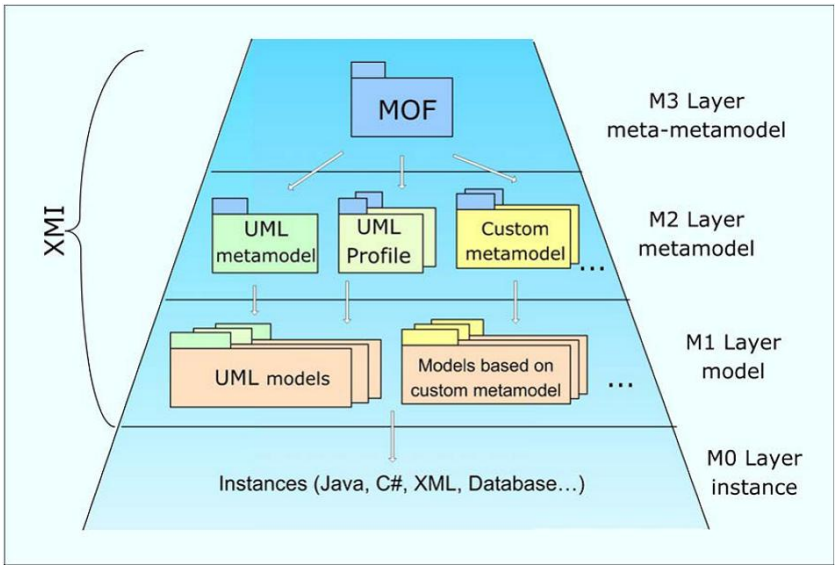


Figure 2.1.: MOF-based metamodeling architecture and relying standards [Djuric et al., 2005]

The definition of a new modeling language consists of the definition of domain-specific modeling constructs from scratch. Benefit of this kind of definition of a domain-specific language is that it has the potential for enabling full expressive power of the new language. Shortcomings are that it requires language design skills, and that there is no language support infrastructure. In the terms of MDA, this is a definition of a new MOF-based metamodel. For the new metamodel, one can define a domain-specific language with full expressive power for modeling the domain. However, missing of support infrastructure means that new specialized tools have to be implemented for using the domain-specific language.

Extension of an existing language means addition of new domain specific concepts into an existing language. It is also called "heavyweight" extension. The benefit of this kind of approach is that it requires less language design skills than the previous approach. Nevertheless, the infrastructure of the base modeling language is not sufficient for the extended version and often cannot be used. In MDA terms this would be the addition of new classes and associations to existing MOF-based metamodel. An example is addition of Petri-net behavioral formalism to UML 2. For this formalism UML 1.x tools cannot be used.

The refinement of an exiting language is a specialization of elements of an already existing modeling language for modeling of some particular domain. It is also known as "lightweight" extension. With this kind of approach a modeling language specializes for modeling a subset of the domain for which the basic modeling language is used. The benefits of this approach is that an infrastructure of the basic modeling language can be completely reused and it requires less language design skills. General shortcoming is that the expressiveness of the new modeling language is constrained by the basic language. In the case of UML profiles, stereotypes, constraints and model libraries can be used for specialization. An example, is UML Profile for CORBA [UML, 2002].

A MDA-linguistic metamodeling architecture facilitates all three kinds of domain-specific language engineering approaches. The only constraint is that the metamodel of the new domain-specific modeling language is defined in MOF. Metamodels and models can be ex-

changed between different tools with the OMG-adopted a XML based format XMI (XML Metamodel Interchange). XMI enables the interoperability between tools for software modeling. However, at present, the most commonly used language for software modeling is UML. UML with its extension mechanism, a profile, enables "lightweight" extensions.

A profile is a special kind of package containing stereotypes and model libraries that, in conjunction with the UML metamodel, define a group of domain-specific concepts and relationships [Selic, 2006b]. Examples of domain-specific profiles are the UML profile for Schedulability, Performance, and Time specification (SPT) [Object Management Group, 2005a] and the UML profile for Modeling and Analysis of Real Time Embedded Systems (MARTE) [Object Management Group, 2008].

In this thesis, a UML profile is developed as the concrete syntax of the performance measurement and assessment language. This UML profile is described in Chapter 9.

3. Chapter
Software Performance Engineering

The term of software performance is overloaded. There are several definitions of performance in software engineering. This chapter in Section 3.1 gives the definition of performance, and performance metrics used in this thesis. Section 3.2 explains alternatives in software performance evaluation. Measurement and assessment as one of the performance evaluation alternatives is explained in Section 3.2 in more detail. Finally, Section 3.4 explains the important concerns in software performance measurement and assessment.

3.1. Software Performance

In this thesis, performance is considered as the degree to which a software system or component meets its objectives for timeliness [Smith and Williams, 2001]. It is a characteristic of a system or a component which can be. in principle, measured with a timer for calender time. Performance is measured with response time and a throughput.

Response time is the time required to respond to a request. The definition of request can vary. It can be an invocation of a method, a transaction or an end-to-end time of a user request.

Response time of a system depends on several factors. One factor are characteristics of the executing platform. The executing platform is underlying hardware and software used for application execution. For example, different hardware characteristics, such as amount of operating memory, size of the processor cash, type of processor and so on, influence the response time of the system. Furthermore, decision on the operating system which is going to be used also impacts the response time of the intended application logic. Besides the executing platform, response time also depends on system usage. For example, if there is a significantly high number of concurrent requests, the response time of the system will be probably higher then for a single request. Finally, the response time depends on the characteristics of the external services the system uses. For example, if the response time of an used external service is high, the response the system will also be higher, and vice versa.

Due to the randomness of the influences, response time can not be computed in some deterministic way. It is rather computed and described in some statistical way. A statistical description is formed from the noticed statistical regularities in response time values. Some of the statistical descriptions used for response time description are mean, max, and min. Furthermore, from statistical observations a probabilistic model of response time can be created. A probabilistic model consists of a list of all possible outcomes and their respective probabilities [Trivedi, 2002]. Such a probabilistic model is used for predictions of response times. Some of the probabilistic descriptions used in response time specification are the density distribution function and the cumulative distribution function. Furthermore, also

are used functions which summarize values in one number. Such functions are skewness, standard deviation, various moments and percentiles, mean, median, mode, and so on.

Throughput is a number of requests a system can process in some specified time interval. The throughput is expressed by the number of requests completed in a time unit, e.g. number of jobs per second [Smith and Williams, 2001].

3.2. Software Performance Evaluation

Generally, software performance can be evaluated with analytical modeling, simulation, or via measurements and assessment [Lilja, 2000]:

- Analytical modeling is a technique where a system is mathematically described. Results of an analytical model tend to be much less accurate then real system measurements. However, it is a way to get some quick initial insight into the behavior of the system, or part of it [Lilja, 2000].

- A simulator is a program which enables imitation of a program execution. In the simulation, only important parts of an execution are imitated. It is less expensive than building a real system and than measuring and performing the assessment. It is generally very flexible. However, simulation is not as accurate as real systems.

- Measurements and assessment provide the most accurate results, as no abstractions are made. Measurement is the process of data collection during software execution. Assessment is the process of metrics computation and reasoning about the system execution according to the computed metrics. The major shortcoming of this approach is that measurements and assessment can be performed only when the software is already developed. This might be too late to remove performance problems.

All three previously mentioned ways of performance evaluation have their benefits and shortcomings. Restricting to only one mean of evaluation can lead to project failure, because the shortcomings of the chosen method might be critical. In general, these three methods complement each other. For this reason, the best method is to apply all three of them.

3.3. Software Performance Measurement and Assessment

In software engineering, instrumentation is the process of adding software probes in the program [Smith and Williams, 2001]. Software probes are additional pieces of code for collecting data about the software execution. Software probes can be added automatically using a variety of techniques and tools, or it can be manually inserted. Automatic adding software probes can significantly reduce a developer's or a system analyst's effort to gather data. However, it is better to insert probes manually because of the ability to make decisions about which measurements to perform.

Software probes can be implemented for different techniques of data collection. Generally, there are two techniques for collecting data, sampling and event tracing [Lilja, 2000]:

- Sampling is a technique where parts of a program are sampled during its execution in some time interval. It is a general statistical technique in which a representative sample of the data about the program during execution is taken. An advantage of this kind of approach is that the impact on performance of the program implied by the measurement does not depend on the execution of the program. It depends only on an interval and a duration of data collection. However, since it is a statistical technique, a general shortcoming is that data collected, when the same experiment is performed twice, will hardly be the same. Moreover, another shortcoming is the possibility that infrequent events can easily be missed. An example of this kind of approach is sampling the program stack to follow the execution of a program.

- An alternative to sampling is a technique called event tracing. It is a process of generating traces of events in the software. A program trace is a dynamic list of events generated by the program as it executes [Lilja, 2000]. A trace contains the time ordered events and can be used to characterize the overall program behavior. Problems which can be encountered with event tracing are system perturbations due the measurement and the amount of resources traces use. Each probe that is added into the program causes execution overhead. In case of very frequent events, the processing overhead can be significant. Another problem is the size of an event trace. Usually, a trace can be very large if we trace each instruction execution. This problem can also occur when very frequent events are instrumented. Large traces occupy resources like memory and, therefore, impact the overall performance of the program.

Because of the nature of a software artifact, there is a variety of ways to instrument software. Software can be instrumented by using already implemented tools, or instruments. According to the nature of tools, there are hardware, software, firmware, and hybrid instruments [Smith and Williams, 2001]. Software instruments are typically the programs for collecting the data about an execution. Hardware tools are external devices independent of the computer systems which are attached to computer systems by high resistance wires, called hardware probes. Likewise, firmware can be instrumented for monitoring. Nevertheless, there are instruments which combine hardware, software and firmware instrumentation. These kind of instruments are called hybrid instruments.

An alternative to the usage of tools for instrumentation is to do it manually. This kind of approach puts more effort on the developer. However, manual instrumentation is better at least for the following reasons:

- Precision—to measure exactly events which are of interest.
- Data granularity—to gather exactly data that is needed.
- Control—to add the possibility of turning on and off desired measurements.
- Deployment of tools—Installation of tools for measurement can be error prone.
- Usage of tools—Developers need to be educated for the usage of tools.

With a manual instrumentation of the system, a significant problem is source code complexity which is increasing with code for instrumentation. Understanding the basic functionality of the program becomes harder, because probes and the code of basic functionality are mixed. Furthermore, instrumentation increases the developer's effort.

Model Driven Measurement and Assessment takes the advantage of benefits of the manual software instrumentation. Furthermore, in order to collect the complete data about response times, used technique for the data collection is the event tracing.

3.4. Concerns in Performance Measurement and Assessment

For the reason of many influences on software system performance, measurement and assessment must be carried out with great attention. This section discusses the concerns in performance verification with measurement and assessment.

Appropriate response time statistical analysis is one of the major concerns for measurement and assessment. Performance, as considered in this thesis, puts time of system reaction to an event as the primary interest. Furthermore, as specified in the definition, the degree of meeting objectives is of interest. Generally, according to what the degree is, two kinds of systems can be recognized:*hard real-time systems* and *soft real-time systems*. In hard *real-time systems* each time a request occurs the deadlines must be met. Otherwise, it is considered as a failure. In *soft real-time systems* not all requests but just large enough number of requests must meet objectives. The degree is usually indicated with statistical analysis of requests which summarizes all occurrences in one number. For this reason, performance requirements are defined in the terms of statistical values, for example ***min*** and ***max*** for hard, and ***mean***, ***mode***, ***median***, ***standard deviation*** or ***density distribution*** of execution durations of software entities for soft real-time systems.

Parallel analysis of throughput and response times is also one of major concerns. The number of requests which occupy the same resources can be of significant importance for response time of systems. In peak hours of system usage the response time can be significantly higher, than in non peak-hours of usage. In peak usage hours it is very important for systems to keep their ability to process the requests and meet the required deadlines.

One more important concern are workload characteristics. Workload characteristics observations are important for validation of correspondence of prediction assumptions with real usage. Predictions of software intensive system performance are always be done with some assumptions on system usage. When performing performance measurement and assessment, reasons for large variations between predicted and computed metrics might be in the difference between the workload used in testing and the specified usage in predictions. Correspondence of workload specification can be identified with observation of the number of workload's requests, arrival rate and pattern.

Characteristics paths are also of significant interest in measurement and assessment. Path characteristics are used in performance predictions. Similarly to workload characteristics, reasons for large variations between the predicted and computed metrics can be differences path characteristics used in predictions and occurred during testing. Path characteristics are probabilities of alternative executions and numbers of iterations in loops.

Isolation of some particular business tasks and measurement and assessments of their response times is one of the major concerns in performance measurement and assessment. Not all business tasks are of the same importance in systems, and the requirements for the most important ones have to be met in any conditions. In performance estimation this issue

is called schedulability [Smith and Williams, 2001]. Assessment of response times of these business tasks is critical in providing a usable software intensive system.

Identification of execution contexts for critical business tasks is as important as identification of critical tasks themselves. Business tasks are identified not only by invocation sequence which implements them, but by the execution context in which the their invocation took place. For example, in the transformational systems the response time dependency on their invocation context is well studied by Rohr et al. [Rohr et al., 2008], Ammons et al. [Ammons et al., 1997], and Hamouladj and Lethbridge [Hamou-Lhadj and Lethbridge, 2004].

Performance analysis of software intensive systems have to be done for representative time periods. Response time metrics are statistical analysis of their values of some time periods. If a statistical analysis, like for example ***mean***, is done for a whole day time period, the value might not be representative for the peak period. Furthermore, not only a representative time period, but also assessment over time is important. Arlit et al. [Arlitt et al., 2001] show that daily and weekly patterns of usage can be distinguished, and that assessment over the time is critical for usability of the system.

Instrumentation transparency is also of great importance in measurement. Instrumentation code increases complexity of software intensive systems, because in the code for implementation of business logic and additional code for data collection is interwoven. In this way, understanding application logic becomes hard. Furthermore, the possibility of errors in removal of measurement code is also present. For this reason, techniques which enable separation of concerns of business logic and instrumentation are needed.

Keeping the consistency between the structuring of collected data and defined data structures for analysis is also of importance. Herewith are avoided failures in assessment due to difference of data structures used for storage and in analysis. For the analysis, data have to be stored in a way that enables a reconstruction of execution and, what is more important, computation on metrics of interest. When the format is established, each software probe has to collect and store data consistently with the defined form.

Reduction of measurement points is also one of the major concerns. It reduces measurement induced system overhead. Software probes for data collection bring additional computation into the original business logic and therefore has impact on performance. Instrumented software, software with inserted probes, occupies more memory, and collection routines additionally use system resources, like I/O. For these reasons, the impact of measurements always occurs and it should be kept as reduced as possible. The overhead can be reduced with reducing points of measurement, and implicitly the number of collected data, and size of collected data.

Overhead due to measurements cannot be avoided. Data collection routines inevitably increase response times. The additional time used for measurement routines either directly or by inference is characterized by a capture ratio. Capture ratio is the percentage of instrumented system response time spent in measurements, and it is one of the concerns when reasoning about validity of measurements results.

The previously described concerns in performance measurement and assessment have been used for the comparative analysis to related work in Chapter 11 of this thesis.

4. Chapter
Libkin's Algebra

Libkin's algebra [Libkin, 2003] is a relational algebra with an open set of aggregate functions. Currently, this algebra is an accurate formal specification of SQL languages implemented in various relational database management systems.

The major benefit of Libkin's algebra is that it defines the set of supported aggregate functions as an open set. Aggregate functions are functions computed on a set of values contained in one column of a relation. Furthermore, Libkin's algebra has the possibility of computing algebraic functions between values in different columns in relations. Algebraic functions are functions like addition, subtraction, multiplication, and division. Previous relational algebras did not have these characteristics.

The first relational algebra introduced by Codd [1970] did not have the aggregate functions at all. Furthermore, this algebra did not support the grouping operator. Finally, it did not contain the possibility of performing algebraic functions of values in different columns.

Klug's algebra [Klug, 1982] is another well-known relational algebra. It integrated a set of the basic five set of aggregate functions, defined in the SQL standard. However, this set cannot be extended. Furthermore, the possibility of computing algebraic functions was also not facilitated.

Libkin's algebra distinguishes two types of columns in relations: numerical and non-numerical. Numerical column type is denoted as **n**. The domain of numerical column is signed as **Num** and can be any numerical domain, for example $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, or $\mathbb{R}$. The non-numerical type is denoted as **b** and the domain of such columns is signed as **Dom**. Such a domain is for example **String**.

A type of a relation is a string over the alphabet *b, n*. A relation $\boldsymbol{R}$ of type $t_R = a_1...a_m$, has m columns. The number of columns is the length of the type and it is denoted as $|t_R|$. The type of the i-th column of the relation is denoted by $t_R.i$. For example, a relation **ExecutionTrace(ElementName, StartTS, EndTS)** has the type $t_{ExecutionTrace} = bnn$. The length of $|t_{ExecutionTrace}|$=3, and the type of, for example **StartTS**, is denoted as $t_{ExecutionTrace}.2$=**n**.

A *database schema* **SC** is a collection of relation names and their types; it is written $R_i : t_i$ if t_i is the type of R_i.

Libkin's algebra, $ALG_{aggr}(\Omega, \Theta)$ is parametrized with two sets. Set Ω is a set of functions and relations on **Num**. Examples of such functions and relations are +, -, *, <, >, and =. The Θ is a set of aggregate functions. Examples of such functions are *mean, mode, sum, standard deviation* and so on. Expressions in this algebra are given in three groups: standard relational algebra, arithmetic, and aggregation/grouping.

Standard relational algebra expressions are:

- **Schema Relation**
  If *R:t* is in SC, then R is an expression of type t,

- **Permutation**
  If e is an expression of type t and θ is a permutation of {1,...,|t|}, then the permutation $\rho_\theta(e)$ is an expression of type $\theta(t)$. The semantics of this operation is that each tuple $(a_1, ..., a_{|t|})$ is replaced by $(a_{\theta(1)}, ..., a_{\theta(|t|)})$,

- **Boolean Operators**
  If e_1, e_2 are expressions of type t, then so are $e_1 \cup e_2$, $e_1 \cap e_2$, and $e_1 - e_2$. These operators correspond to operators of standard relational algebra.

- **Cartesian Product**
  For $e_1 : t_1$, $e_1 : t_1$, $e_1 \times e_2$ is an expression of type $t_1 \cdot t_2$, $\cdot$ is a concatenation of two strings. Cartesian product semantics corresponds to Cartesian product of standard relational algebra,

- **Projection**
  If e is of type t, then $\pi_{i_1,...,i_k}(e)$ is an expression of type t' where t' is the string composed of $t.i_j$s, in their order. The semantics of projection is equal to the one defined in standard relational algebra,

- **Selection**
  If e is an expression of type t, $i, j \leq |t|$ and $t.i = t.j$, then $\sigma_{i=j}(e)$ is an expression of type t. The meaning of the the condition $i = j$ in the selection predicate is that the resulting relation contains only those tuples $(a_1, ..., a_{|t|})$ from the expression e, in which $a_i = aj$ holds.

The arithmetic expressions of the Libkin's algebra are:

- **Numerical selection**
  If $P \subseteq Num^k \rightarrow Num$ is a k-nary predicate from Ω, and $i_1, ..., i_k$ are such that $t.i_j = n$, for $j = 1..k$, then $\sigma[P]_{i_1,..,i_k}(e)$ is an expression of type t, for any expression e of type t. The semantics of this expression is that the resulting relation will contain only those tuples $(a_1, ..., a_{|t|})$ from the expression e, in which $P(a_{i_1}, ..., a_{i_k})$ holds,

- **Function application**
  If $f : Num^k$ is a k-nary numerical predicate from $\Omega, i_1, ..., i_k$ are such that $t.i_j = n$, for $j = 1..k$, and e is an expression of type t, then $Apply[f]_{i_1,...,i_k}(e)$ is an expression of type $t \cdot \mathbf{n}$. This expression replaces each $a_1, ..., a_m$ of expression e with $(a_1, ..., a_m, f(a_{i_1}, ..., a_{i_k}))$.

- **Constants** If c is a constant, then $Apply[c]_\epsilon$ is an expression of type $\mathbf{n}$. ϵ refers to c taking no argument, as a constant can be seen as a function of arity 0. It actually produces the relation c.

Finally, the grouping and aggregation expressions are:

- Aggregation
  Let $\mathcal{F}$ be an aggregate from Θ. For any expression of type t and i such that $t.i = \mathbf{n}$, $Aggr[i : \mathcal{F}](e)$ is an expression of type $t \cdot \mathbf{n}$. The semantics of this operator is that it replaces each tuple $(a_1, ..., a_{|t|})$ with $(a_1, ..., a_{|t|}, f)$ were f is a value of $\mathcal{F}$ computed from ith attributes of all tuples in the expression e.

- Grouping
  Let us assume that $e : t$ is an expression over $SC \cup S : s$. And let e' be an expression of type $u \cdot s$ over SC, where |u|=l. Then $Group_l[\lambda S.e](e')$ is an expression of type $u \cdot t$. The semantics of this operator is that it first groups tuples according to the first l attributes of the relation e''s tuples. Then it applies e to the relations produced with this grouping and denoted as S. For example, let e' be a relation of $(a_1, ..., a_4)$ tuples. And let $Sum \in \Theta$. Then the result of the expression $Group_2[\lambda S.Aggr[2 : Sum](e')$ returns a relation of type |t|+1 and $(a_1, ..., a_5)$ tuples. The attribute a_5 of a tuple a is the value of the Sum aggregate computed from a_4 attributes of all tuples having the same values of the first two attributes as the tuple a.

The Libkin's algebra is used for the formal semantics definition of the language. Furthermore, the formal semantics is used as the design model of transformations from the domain specific language for performance measurement and assessment to the relational database management system.

Part II.

MoDePeMART: Model Driven Performance Measurement and Assessment with Relational Traces

5. Chapter
Performance Assessment with MoDePeMART

The Model Driven Measurement and Assessment with Relational Traces, initially proposed by Bošković and Hasselbring [Bošković and Hasselbring, 2009], is an approach which integrates the process of measurement and assessment in MDE. This chapter discusses how is performance measurement and assessment carried out in the ***MoDePeMART***.

Model Driven Engineering, as discussed in Chapter 2, suggests multidimensional modeling [Tarr et al., 1999] with a language for each dimension. Each dimension is related to a concern of interest in development of that software product. Concerns of interest are actually domains of interest, like security, functionality software should provide, privacy, redundancy and so on, for the particular software intensive system. Languages for each dimension enable modeling of that particular domain. Domain models combined according to predefined intersections and mappings, form the software intensive system model.

For each project the dimensions have to be stated at the beginning of the project. It is unlikely that the same set of modeling languages and dimensions would be of interest for each project. Some of the dimensions can be standard across projects, like authorship and version, and some can be optionally integrated like concurrency and distribution [Kent, 2002].

Because of the significance of software intensive system performance, in this thesis it is considered that performance measurement and assessment is a standard process across projects. Evidence of performance as an important non-functional property is that it is one of QoS attributes, as discussed in Chapter 1. Furthermore, OMG defines the UML Profile for **S**chedulability, **P**erformance and **T**ime (SPT) [Object Management Group, 2005a] and his successor the UML Profile for **M**odeling and **A**nalysis of **R**eal-Time **E**mbedded **S**ystems (MARTE) [Object Management Group, 2008]. These standards are mostly dedicated to standardization of model annotation for performance prediction. Later, from annotated models, prediction models according to some analytical theory like Queuing networks, Petri nets, or Markov chains are created [Object Management Group, 2005a] [Object Management Group, 2008]. This thesis complements work on performance prediction and integrates the final step of verification of performance in MDE.

Having previously mentioned, in MDE, performance measurement and assessment can be seen as a separate dimension. Consequently, it has to be supported with the appropriate language and mapped to the system functionality definition modeling language(s).

The domain-specific language for performance measurement and assessment has to support domain-specific constructs for specification of performance measurements which enable assessment. Accordingly, a language for performance measurement and assessment requires

the ability of unique identification of the event of interest, execution context in which this event is of interest, and specification of the metric of interest.

The Model Driven Measurement and Assessment suggests declarative specification of the execution context and an event of interest in this context. Furthermore, it suggests usage of relational databases for data storage and metrics computation. Finally, it uses coupled generation of code for data collection and for data analysis. The detailed process is presented in Figure 5.1.

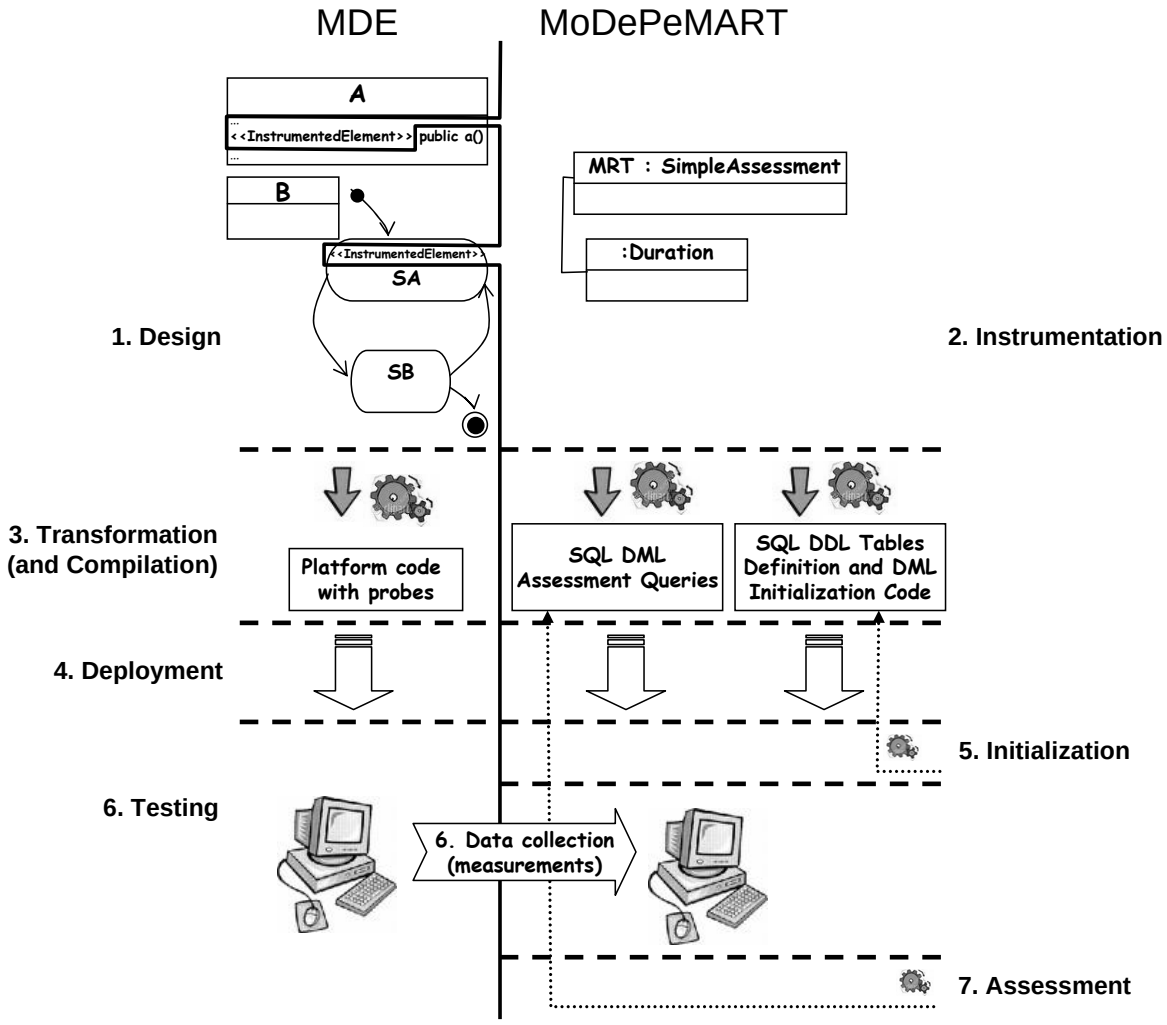


Figure 5.1.: Model Driven Performance Measurement and Assessment With Relational Traces Process. The figure demonstrates the order of activities and artifacts produced

This approach is integrated into the process of Model Driven Engineering at the point where the final design model is developed. The final design model is, in this thesis, the model from which the code is generated. The final design model is developed via collecting requirements and creating requirements models. Later, through refinement, more and more concrete models are created until the final design model is reached. The design process can be either iterative and incremental, such as Rational Unified Process (RUP) [Kruchten, 2003]

or waterfall process, as it is the case of Convergent Architecture [Hubert, 2002].

After establishing a final design model, a process of instrumentation proceeds. Instrumentation, in this approach means: annotating events of interest in the model, specification of relevant event's execution contexts, and specification of metrics of interest. The execution context consists of the state of the system during the occurrence of the event of interest and the sequence of invocations which the event is part of. More on the execution context, and metrics specification is described in Section 6.1 and Chapter 7.

In the process of transformation code is generated from the previously defined annotated model, context and metrics specification. The code consists of four parts: software functionality platform code with probes, code for database tables specification, code for database initialization, and code for desired metrics computation. Probes added to the platform functionality code consist of code for data collection and storage in database. Code for database tables for data storage are specified with SQL Data Definition Language (SQL DDL). For the database tables initialization and computation of desired metrics SQL Data Manipulation Language (SQL DML) code is generated.

The transformation is followed by compilation of platform code only. In order to be able to deploy the code, the platform code has to be compiled into a platform-specific executable code. The SQL code does not need to be compiled as the code is already executable in the target relational database management system.

Finally, deployment takes place. In this phase the executable application code and the assessment and initialization code are deployed to the target computer and relational database management system, respectively.

The measurement and assessment takes place at the testing phase of software development. It starts with the initialization of the database for data storage. Initialization consists of execution of SQL DDL code for tables needed for data storage, and SQL DML code for initial entries of tables required for metrics computation. More on the initialization is given in the Chapter 8.

The next step is execution of test cases and measurements. In this step software execution data is collected and stored in the previously initialized database.

Performance assessment is performed after the measurements. Metrics required for verification of predicted values are automatically computed. For automatic computation, previously generated SQL DML Queries are used. At this stage, a software developer or performance analyst needs only to execute them and check if the computed values conform to the predicted values and to requirements.

The approach presented in this thesis is not intended for performance debugging. It is the approach only for assessment of requirements and predictions fulfillment. In the case that computed metrics do not agree with the predicted values and requirements, performance analyst can try to seek for reasons of not fulfilling performance goals and disagreement with predictions with tools for complete system observations such as profiling tools.

6. Chapter
Basic Assumptions

MoDePeMART is an approach for performance measurement and assessment of specific kind of systems, modeled with specific languages. This chapter explains characteristics of languages and systems for which this approach is intended. Furthermore, it describes the nature of models used for development and performance evaluation of software systems. Accordingly, the chapter consists of two sections. Section 6.1 explains the main characteristics of systems for which the approach is intended, and their modeling languages. Section 6.2 discusses model kinds in order to precisely specify relations between prediction, software development, and models for measurement and assessment. Furthermore, this section gives the formal relation between models and runtime systems.

6.1. Vertical and Horizontal Dimension of Software Modeling and Execution

Modeling languages in Model Driven Engineering raise the abstraction in software engineering, and shift the focus of software development from the implementation domain to the problem domain. According to the nature of the problems in the domain for which the software system is developed, DS(M)Ls of that domain support appropriate problem decomposition.

Generally, two kinds of software systems exist: transformational and reactive [Wieringa, 2003]. Transformational systems are systems which take some input value and transform them to some output value. An example of a transformational system is a library of mathematical functions. Function ***sin*** which computes sinus of a number, for instance, takes the input value and transforms it into output value. Contrary, reactive software systems are systems which constantly interact with environment in such way that they receive stimuli and, according to stimuli and current state, produce some action in the environment, and/or change the current state. Accordingly, reactions of a reactive system are not always the same for the same stimuli. An example of reactive systems is a graphical user interface (GUI). Depending on the current state of GUI, which is for example current screen, stimulus like, for example clicking on a button, can lead to changing the look of the screen. However, in the case of different current screen, clicking on the button can lead to changing to a different screen.

Correspondingly, two general paradigms for problem decomposition exist, transformational, also known as algorithmic, and reactive [Wieringa, 2003]. Transformational problem decomposition, divides problem into smaller, also transformational, subproblems. Later, subproblems are solved and composed into the final solution. This is a typical problem

decomposition in procedural programming. Some DSLs for transformational systems are developed by Stevenson and Fleck [1997] and Klarlund and Schwartzbach [1999], for image understanding algorithms and algorithms with sets of strings and trees, respectively. On the other hand, languages for reactive systems enable problem decomposition in such a way, that the system is separated into different entities which communicate to solve the problem. Furthermore, these languages support stimulus-response decomposition mechanism [Wieringa, 2003]. An example of DSL for reactive system behavior are BDL developed by Bertrand and Augeraud [1999] and Interaction Object Graphs invented by Carr [1997], for describing problems of concurrent behavior in object oriented languages and graphical user interfaces, respectively.

However, there are also classes of systems, like information systems, which have both, transformational and reactive behavior [Wieringa, 2003] and therefore, languages for such a class of systems have to support both transformational and reactive problem decomposition [Wieringa, 2003]. These systems, with twofold behavior, have vertical and horizontal dimensions, which correspond to transformational and reactive behavior, respectively [Katara et al., 2004][Kurki-Suonio, 2005]. Each dimension is modeled with a modeling formalism appropriate for that behavior kind.

This section provides the introduction to transformational and reactive philosophy of problem decomposition and vertical and horizontal dimensions of software systems. Furthermore, examples of formalisms for each dimension will be described on the example of the case study. The same case study will be used throughout the thesis.

6.1.1. Vertical Dimension

Vertical dimension of software development represents part of models used for transformational parts of software-intensive systems. Transformational philosophy of software development is the algorithmic philosophy of programming, introduced by Dijkstra [Dijkstra, 1976]. Algorithmic philosophy of programming defines a software-intensive system as a system that when started in an "initial state" will end up in a "final state." Happenings which take place upon activation of the mechanisms are specified as algorithms. Example is given in Figure 6.1 where the structural view of the model of application used throughout the thesis as the example case study is described.

The example application is an electronic items management application. It is an application for storing and organizing personal electronic items, such as music videos, music tracks, albums and audio books. The complete functionality of this application and its implementation is explained in this and in the next subsection through its vertical and horizontal dimensions. In this subsection the focus is only on the vertical dimension. The basic elements of vertical dimension are in the structural view of the model.

The structural view of the the application model consists of four entities ***UserInterface***, ***AudioItemFacade***, ***VideoItemFacade***, and ***ItemFacade***. These entities which group and implement functionality of user interface, audio items subsystem, video items subsystem, and electronic items subsystem, respectively.

ItemFacade is the class which implements a simple interface for obtaining database electronics items of any type. For this purpose the ***getItem*** method is implemented. Methods ***on*** and ***off*** will be explained in the next subsection, since they represent a part of the horizontal dimension.

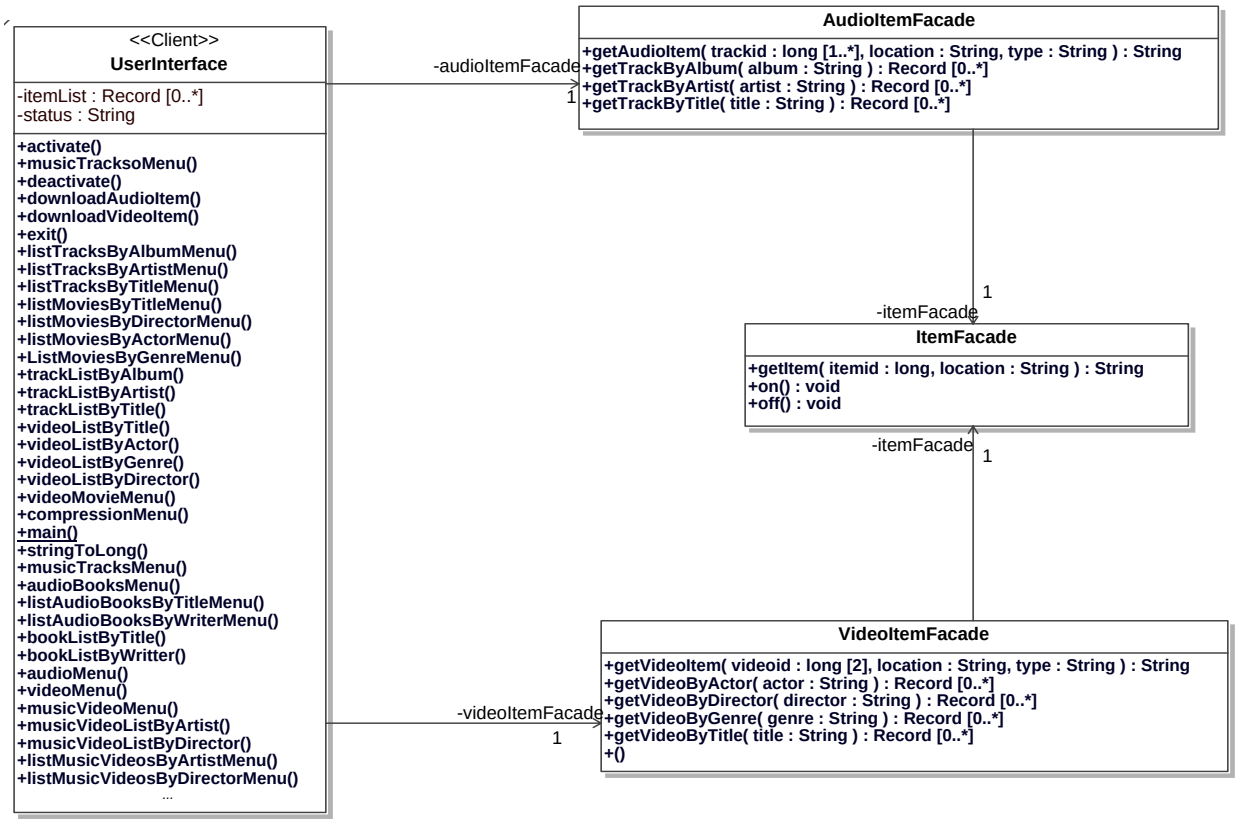


Figure 6.1.: UML Class diagram of electronic items management system case study

AudioItemFacade implements and clusters methods for audio items business logic. Methods ***getTrackByAlbum***, ***getTrackByArtist***, ***getTrackByTitle***, implement functionality for getting lists of audio tracks from appropriate album, artist, or with appropriate title, respectively. The method ***getAudioItem*** is the method which implements obtaining audio items from the database.

Similarly to the class ***AudioItemFacade***, class ***VideoItemFacade*** implements and clusters video items' business logic. Methods ***getVideoByActor***, ***getVideoByDirector***, ***getVideoByGenre***, ***getVideoByTitle*** implement obtaining lists of video items with appropriate actor, director, genre, or title, respectively. The method ***getVideoItem***, also similarly to ***getAudioItem*** of the class ***AudioItemFacade***, implements the obtaining appropriate video item from the database.

For definitions of transformational systems Dijkstra defines primitive and composite mechanisms. Primitive mechanisms are mathematical operations and atomic actions which change the state of a system with assignment of values to variables. Composite mechanisms, are actually concatenation of various number of composite and primitive mechanisms. It is an example of the Composite design pattern [Gamma et al., 1995].

Transformational problem decomposition defines two kinds of general relations between the algorithm subsequences, sequential composition, and invocation. In sequential composition the start of one sequence follows the end of its predecessor. In invocation, start of one sequence is nested, and is part of its predecessors sequence [Katara et al., 2004]. An example of decomposition relations is given in Figure 6.2 where a UML Sequence Diagram

of the ***getVideoItem*** method of ***VideoItemFacade*** is shown. UML Message Sequence Charts are a commonly used mechanism for modeling the transformational dimension of an application [Katara et al., 2004].

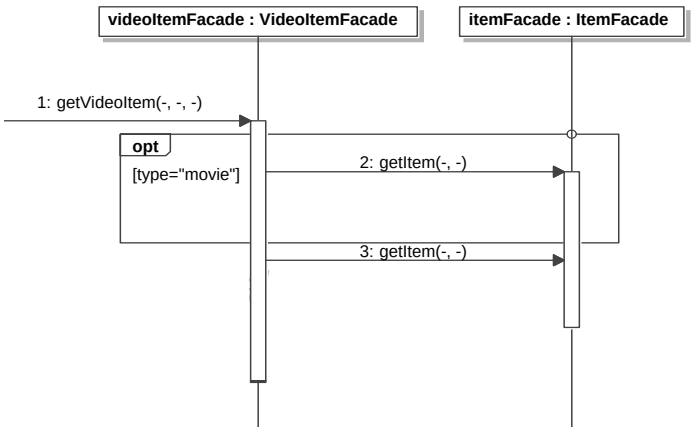


Figure 6.2.: Sequence diagram of class VideoItemFacade getVideoItem method

In the method ***getVideoItem*** two item types can be requested. One video item type is movie, and the second one is music video. Accordingly, control flow has two alternatives. The first alternative is the case when required video item is a movie. In this case, from the database, two files are obtained. One is these two files is the movie trailer, a small video file shows a résumé of the movie. The second file is the movie itself. In the case that the type of the video item is a music video, the trailer is not needed and there is only one invocation of ***getItem*** method, for the music video itself. This is, in the control flow represented with *Option* (opt) block. Option block is a block in which either a scenario inside the block is executed, or nothing happens. In the case that the requested video item is a movie, the first ***getItem*** invocation will be executed before the invocation of the same method for obtaining the movie. When the requested video item is a music video, the first invocation is not executed, and only the music video is obtained from the database.

If it is taken into the consideration only the request for a music video, invocation relations between ***getVideoItem*** and ***getItem*** methods can be recognized. In the case, when the video type required is a movie, sequential composition of two invocations of ***getItem*** can be seen.

The *Option* block in the previous example is a special kind of command, called guarded command. A guarded command consists of two parts, a boolean expression called ***guard***, and a simple or composite statement. A boolean expression has to be satisfied in order to execute simple or composite statements. Nevertheless, there is often the a set of guarded commands where each statement is executed in some particular case. In order to provide a programmer with the mechanism to specify a set of guarded commands where each command is executed in a particular case, Dijkstra defines the command called ***guarded command set***. This element can be seen in our example case study. In Figure 6.3, the sequence diagram of ***getAudioItem*** method of ***AudioItemFacade*** is shown.

Audio items can be classified in two categories, audio music tracks and audio books. For these two audio item types, three kinds of requests can be given: request for obtaining a

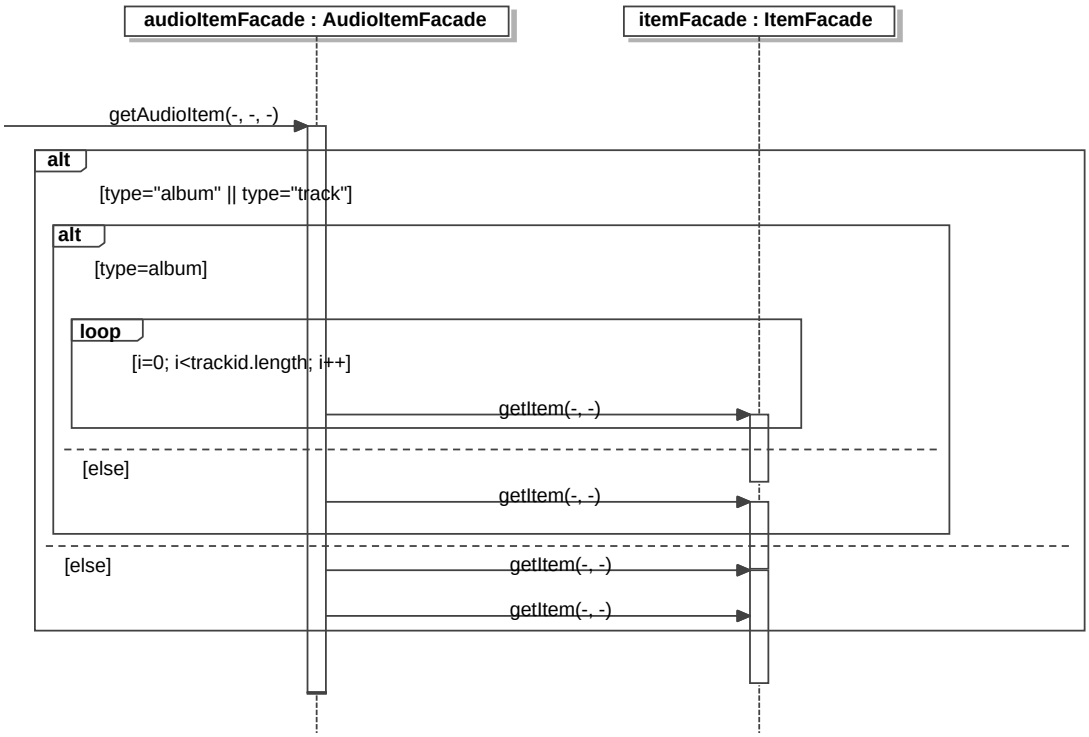


Figure 6.3.: Sequence diagram of class AudioItemFacade getAudioItem method

music album, single music track, or an audio book. Accordingly, the body of the method consists of two nested alternatives. The fist alternative is examining if the audio item type is album or item. If it is one of them, the control flow enters the upper choice of the first alternative, and that is second alternative. In this alternative if item type is a music album, the control flow enters the upper choice of second alternative. In this choice is the ***loop*** where all audio items are collected from the database. Loop is the last element defined by Dijkstra, for specifying an algorithmic program structure. It is a guarded command which activates when its guard is true, but after its termination, the examination of guard starts over again and, if it is still true, execution of the command takes place again. These steps repeat until the guard becomes false.

Alternative to obtaining an audio album is obtaining only one track. Extracting track from the database is performed with only one invocation of the ***getItem*** method of the ***ItemFacade*** class. Finally, if the electronic item type is neither music album nor track, the control flow of the method is executing the lower case of the first alternative and that is obtaining an audio book from the database. An audio book comes always with a written copy of the book in electronic form, e.g., in Portable Document Format (pdf). For that reason, in this case in the control flow two ***getItem*** invocations are performed, for getting both, audio book and an electronic written version.

One of the major characteristics of the vertical dimension of software modeling is used for specifying terminating processes. Terminating processes are processes which, after activation, end after a finite number of actions.

Traditional design methods and programming languages, and therefore current technologies and tools, mostly support the vertical dimension. This dimension supports interaction

with the environment for transforming input into output through terminating sequences of states. Terminating sequences, in current programming languages, are defined with ***single***, ***composite***, and ***guarded*** commands, ***guarded command sets***, and loops, which are composed either in invocations or sequentially. Opposite to transformational, reactive behavior consists of non-terminating state sequences. In reactive behavior a system interacts with the environment and, depending on the information it receives from the environment, changes its own state. Reactive behavior of a system is modeled with horizontal dimension of the model.

6.1.2. Horizontal Dimension

Vertical dimension and algorithms, as explained in the previous subsection, yield outputs completely determined by their inputs. They are metaphorically dumb and blind, and do not consider interaction with environment at all [Wegner, 1997].

Technological shift from mainframes to workstations and networks, embedded systems and graphical user interfaces induces a development paradigm shift from algorithmic to interaction based [Wegner, 1997]. Algorithms, because of their environment ignorance, cannot handle tasks which require interaction with surroundings. In order to enhance algorithms with interaction, and give "smartness" to algorithms, systems with algorithmic behaviors often have subsystems with interactive behavior. Interactive, also called reactive, behavior constitutes the horizontal dimension of architecture [Kurki-Suonio, 2005].

For modeling reactive systems, special formalisms which enable stimulus/response behavior modeling are used [Wieringa, 2003]. Examples of formalisms which enable stimulus/response modeling behavior are statecharts [Harel, 1987], and their object oriented version [Harel and Gery, 1996]. The object oriented version is in the core of UML, and it is called state machines.

A reactive part in the example application used in this thesis is the user interface. It is modeled with the ***UserInterface*** class in Figure 6.1. A part of a state machine of this class is given in Figure 6.4.

User interface is generally a reactive part in the most of applications which interact with a user. They can be, for instance, **G**raphical **U**ser **I**nterfaces (GUI) or command line interface. Looking in broader sense, not only a human, but a set of sensors for observations of environment conditions are also interfaces. Examples are thermal or light sensors in embedded systems. These systems have in common that they do not always react on the same stimuli in the same manner. They react depending on both, current state and received stimuli.

The user interface in this case study is a set of menus for browsing and downloading electronic items. For the brevity of this section, not the complete reactive model of application is explained. Here is depicted only the part of reactive model which are of importance for for understanding horizontal dimension and problems related to performance measurement and assessment in systems with the horizontal and the vertical dimension. More detail on the application can be found in Appendix A where the complete model of the application is given.

The first state after the activation, marked through the transition from the initial pseudo state, is the ***MainMenu*** state. This transition is done on the signal ***activate*** from the environment. The signal ***activate*** from the environment is given when the application is started.

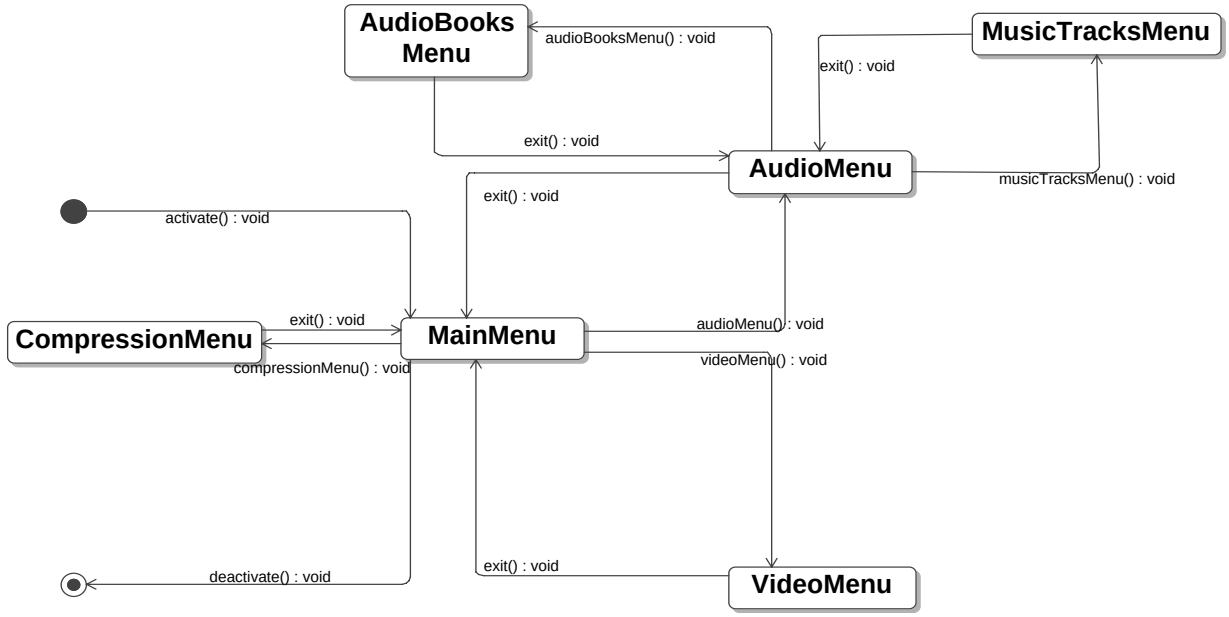


Figure 6.4.: Part of the UserInterface class UML State Machine

MainMenu state is the top menu of the application. This menu offers possibilities of going into sub menus for browsing video files, audio files, or managing compression. The menus for browsing video files, audio files, and managing compression are the ***VideoMenu***, the ***AudioMenu***, and the ***CompressionMenu*** states, respectively. From the state ***MainMenu***, transition to the ***VideoMenu***, the ***AudioMenu***, and the ***CompressionMenu***, is done by invoking the ***videoMenu***, the ***audioMenu***, and the ***compressionMenu*** methods of the ***UserInterface*** class, respectively. For the reasons of brevity, possible transitions from the ***CompressionMenu*** and the ***VideoMenu*** are not considered, except those back to the ***MainMenu*** state. Only transitions from the ***AudioMenu*** are studied in detail.

AudioMenu is the top menu for browsing through audio items of the example application. As already explained in the previous subsection, two audio electronic items and three use cases of the application exist. Audio item types are audio tracks and audio books. The use cases of this method are obtaining audio tracks only, obtaining complete albums, and, finally, obtaining audio books. For this reason, in the application exist two sub menus in which user can go from the ***AudioMenu*** state:***AudioBooksMenu*** and ***MusicTracksMenu***. ***AudioBooksMenu*** encapsulates options for browsing audio books, and ***MusicTracksMenu*** encapsulates options for browsing audio tracks and albums. For the same reason as in the previous cases, further transitions from ***AudioBooksMenu*** and ***MusicTracksMenu*** are not analyzed here.

In the following major differences between transformational and reactive systems, and therefore, models for horizontal and vertical dimensions of software will be outlined.

It has already been mentioned that the horizontal dimension is used for non-terminating processes modeling. Non-terminating processes do not terminate in a finite number of steps defined with the input value. They can terminate only on requests from the environment, if there exist such a response to a stimulus. Without activating the special transition for termination they remain continuously active.

In the case study, exit from the application is invocation of the method ***deactivate*** of the ***UserInterface*** class when it is in the ***MainMenu*** state. The number of transitions between other possible states can be infinite. Consequently, the application generally can always be active and, for example, available for access over the Internet.

Another difference, already pointed out earlier in this subsection, is that a reaction on a stimuli from the environment of the reactive systems, depends on the stimuli and the current state. In the previously defined example there are several exiting transitions which are triggered with the invocation of the method ***exit*** of the class ***UserInterface***. If the ***UserInterface*** is in the ***AudioMenu*** state, after the execution of the ***exit*** method, the instance of ***UserInterface*** will end up in the ***MainMenu*** state. If the ***UserInterface*** class is in the ***MainMenu*** state, an invocation of the ***exit*** method will produce not a single reaction, because there is no outgoing transition from the ***MainMenu*** state which is triggered with this method. Accordingly, it can be also noticed that each state defines both, the subset of possible inputs to which the system in current state reacts, and appropriate responses to the given input.

Finally, it should be clarified what is considered as a state in systems with horizontal and vertical dimension.Usually, a set of system's variables is considered as a state of the system. These understanding of state is essence of Dijkstra's algorithmic philosophy of programming. Pure understanding that values of variables is a system state often can lead to state explosion in analysis, and, therefore, this kind of interpretation is usually impractical. In systems with horizontal and vertical dimension of an architecture there is a distinction between variables and states. In these systems variables are considered as a quantitative aspect of behavior, and states as a qualitative aspect [Selic et al., 1994]. Qualitative aspect means that state is part of systems' control flow specification. Quantitative aspect of behavior does not impact the steps that execute in the system behavior but can impact number of steps in control flow execution, e.g., in loops. Implicitly, the change of variables does not imply change of the system state. States can have an impact on some particular constraints and conditions in functionality, and variables are products or operands of computation.

The previously explained difference can be seen in the example. Variables ***status*** and ***itemList*** of this class contain the list references to electronic items chosen according to some criteria, and the ***status*** contains the status of performed transaction with the database. A state of an ***UserInterface*** object does not depend on values of these variables. These variables depend only on the criteria given for the listing items, and the status of transaction. The control flow stays the same no matter what are values of the variables.

The final part of the horizontal dimension of the case study is the state machine of the ***ItemFacade*** class. The state machine is shown in Figure 6.5.

The state machine in Figure 6.5 manages compression in a data transfer. The compression option is added with the purpose of response time improvement. Depending on the size of the file, compression can differently impact response time. In the case of large files, and a bad connection with the database, turning on the data compression can reduce response time. In this case, compressing data at server side, transferring compressed data, and decompressing them at the client side, can last shorter comparing to transferring them uncompressed. Nevertheless, the compression can influence the response time in the opposite way also. Depending on the size of files and the capacity of the communication between the database and the instance of ***ItemFacade***, compression and decompression routines and the transfer can take a larger amount of time than the transfer of an uncompressed file. The decision on

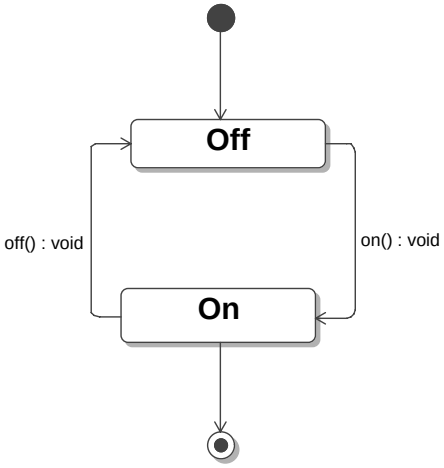


Figure 6.5.: Item Facade class UML State Machine for managing data compression

turning the compression on and off is left to the user. With respect to the state machine of the ***ItemFacade***, the compression is turned on when the object of the ***ItemFacade*** class is in the state ***On***. Correspondingly, the object is in the ***Off*** state when the compression is turned off. The default state of an instance of the ***ItemFacade*** class is ***Off*** state.

The horizontal dimension of software models is a consequence of software intensive systems usage evolution, from large computation-oriented mainframe servers, to embedded systems and interaction oriented software intensive systems. Current support in implementation technologies for horizontal dimension is low. Nevertheless, as the importance of this dimension is recognized, the emerging techniques like design patterns [Gamma et al., 1995], and design approaches, like Aspect-Oriented Programming [Kiczales and Hilsdale, 2001] are starting to integrate the horizontal dimension modeling in up-to-date technologies [Kurki-Suonio, 2005].

6.2. Model Kinds and System at Runtime

In Chapter 2 the foundations of metamodeling and relations between metamodel and model of a system are described. Although it is very important to understand language engineering, it is also important for a language to have a clearly defined relation between the real system which is modeled, called system under study, and models. In this Section, model types according to intent of use and according to mapping of modeling constructs and elements of the system under study are explained. Finally, the formal definition of relation of models and runtime systems are given in the last subsection.

6.2.1. Descriptive and Prescriptive Models

Generally, according to the intent of use and mapping of modeling constructs and elements of the system under study, two kinds of models are recognized, descriptive and prescriptive [Seidewitz, 2003].

Descriptive models are usually models used in traditional scientific disciplines, such as physics and chemistry. The main purpose of this kind of models is to isolate the aspect

of interest, and to abstract for some particular use. These models are commonly used for judging some property of a system without measuring it. This model kind is, for example, a drawing of a conductor with its resistance (R) and potential difference between the ends of it (U) in physics. This model is correct if statements stated in the model correspond to observations of the system under study [Seidewitz, 2003]. In this example potential difference can be measured with voltmeter. If the voltmeter that shows the value of the potential difference equals the value written in the model, then the model is correct with respect to potential difference.

Reasoning in descriptive models is a way to deduct new statements about the system under study and it is done with theories [Seidewitz, 2003]. For example in this model, in order to deduct a value of electric current (I) a theory that the electric current equals quotient of potential difference and resistance can be used. Furthermore, new statements deducted from the old ones have to be consistent with old statements, and two statements are consistent if they can both be true for system under study [Seidewitz, 2003]. In this example, one would measure the electric current, resistance, and potential difference of the system under study. If they both are true, then they are consistent.

In software engineering this kind of models can be queuing networks for software performance [Lilja, 2000]. With queuing networks software intensive system can be described, and for instance, according to arrival rate, service and waiting time, compute utilization and response time. The formula for computation of utilization and response time is in this case a theory. Descriptive models can also be UML and architectural models, when they describe some already existing software for documentation.

Alternatively to descriptive models, ***prescriptive models*** exist. Prescriptive models are traditionally used in engineering disciplines. These models are used for specification of systems that still do not exist in reality. In this case, potential systems are now systems under study. Examples of these models are electrical schemes in electronics, blueprints in civil and mechanical engineering.

Prescriptive models are used for specifying systems for particular needs, and for proving that these needs will be achieved before the system is built. In these models, theories also intensively used. They are used in oder to specify systems with characteristics that fulfill requirements. For example, the theory which relates electric current, resistance and potential difference of a conductor is used in electric schemes in order to specify values of scheme elements so that output value of electric current or potential difference meet ones requirements.

In software engineering UML models can also be prescriptive. Gathering requirements, and specification of software intensive system, with use case, class, sequence and other diagrams is a process of prescriptive models development. These models can be annotated with some particular values, like for example, values specified in SPT and MARTE profiles and then with usage of, for example, queuing theory, performance metrics can be computed, and predicted for the system that is still not developed.

Model Driven Engineering uses the fact that both software and models are similar artifacts, namely bits and bytes in memory. This property allow direct evolution of models in to complete implementation without discontinuities in the expertise, materials, tools, or methods Selic [2003]. In electrical and civil engineering models are different artifacts compared to the artifacts produced. Furthermore, in order to build the artifact which is specified, persons, materials, tools and methods used, are different from ones involved in modeling.

The central question of both descriptive and prescriptive models, is the interpretation of the model. The interpretation of a model is actual mapping of the model's elements to the elements of a system under study. This mapping is important in order to be able to determine the truth of the statements in the model from the system under study [Seidewitz, 2003]. The next subsection gives an overview of model types with respect to interpretation.

6.2.2. Token and Type Models

Beside the distinction with respect to purpose, the crucial question for usage of models is their relation to the system under study. According to their relations to system under study two kinds of models can be distinguished, token and type models [Kühne, 2005].

Token Models

Token models are most often the models people have in their minds. Blueprints of houses and motors in mechanical engineering, schemes in electrical engineering, and geographic maps are typical representatives of token models. The main characteristic of this kind of models is their one-to-one mapping to entities of the system under study. Typical example of this kind of models is a map, depicted in Figure 6.6

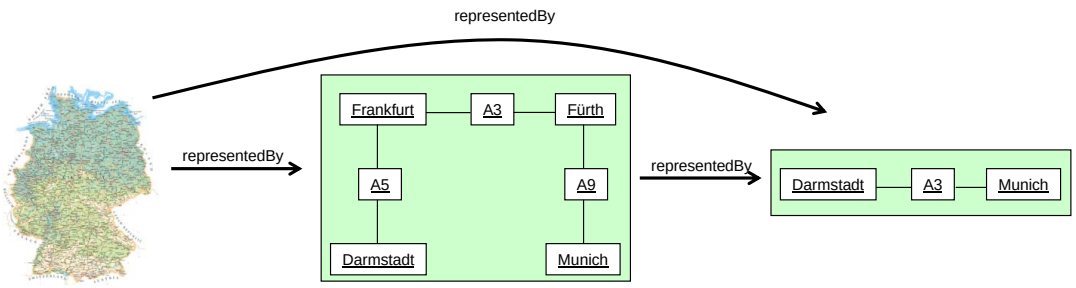


Figure 6.6.: A map as an example of a token model [Kühne, 2005]

The purpose of token models is to isolate singular aspects of interest of the system under study. For example, the model in the middle, which can be a part of the road map, isolates the path of highways from Darmstadt to Munich. The model captures cities at the path and motorways that connect them. If capturing a map in a larger scale is of interest, or capturing less information, it is possible to have a model of the middle model, and in the figure this is the model at the right side. Finally, if the figure at the left side is considered, it is a geographic map, which is, also, a model of system which exists in reality, in this example Germany. Here should be noticed that a model at the left side is not a metamodel of the model in the middle, although it is a model of it. Furthermore, it should be noticed that a model at the right side is also a model of the geographical map, and the system under study. Accordingly, all three models in the figure are at the M1 layer in the metamodeling stack.

Token models are often called "instance models" and "snapshot models". They are called, "instance models" because they model instances and not types and "snapshot models" because they can capture a configuration of a system at some time point [Kühne, 2005]. In UML, Object diagram is used for modeling instance models.

In software engineering token models are not extensively used. They are used as basis for simulations and for capturing some important system configurations. Nevertheless, for performance estimation token models are essential. Servers in queuing networks are instance models of the system. Furthermore, SPT and MARTE standards explicitly state that their models are instance models.

Type Models

Type models use the power of human ability to classify objects according to their properties which are of our interest, and use these classifications for drawing conclusions [Kühne, 2005]. This kind of models captures general characteristics of a set of objects, and group objects according to them, contrary to instance models which do not group objects, but rather just isolate an aspect of interest.

Type models are mostly used in software engineering, and a typical representative of type models of a program is source code of an application, Figure 6.7

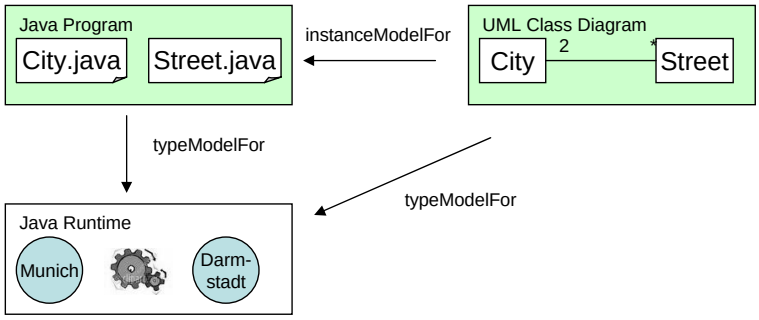


Figure 6.7.: Java classes and UML Class Diagrams as type models of a running program [Kühne, 2005]

Java program code for implementation of the road map presented in the previous subsection is the type model of the running application of road map in a computer. The mapping from the model to the system under study, the running application in this case, is not one to one as in token models, but rather one to many, because a concept in the Java program code, the class ***City.java***, maps to all of city objects of the running program. Java class only specifies the characteristic of a city object, which is, to how many streets it can be related. In MDE, when UML is used as a modeling language, a class diagram is also a type model of the running program. It should be noticed the relation between the Java code and class diagram of the application. Class diagram is a token model of the Java code, because the mapping is one-to-one. This characteristic will be shown as important for the measurement and assessment approach of this thesis.

6.2.3. The Runtime System and Model in Model Driven Measurement and Assessment with Relational Traces

A formal connection of the system under study and the model is of significant importance for ensuring the design intent and accuracy of models. Many modeling techniques had

limited success because it was not always clear how the concepts used in models mapped to the underlying implementation [Selic, 2003]. In the Model Driven Performance Measurement and Assessment with Relational Traces we assume the following relations of models, depicted in Figure 6.8

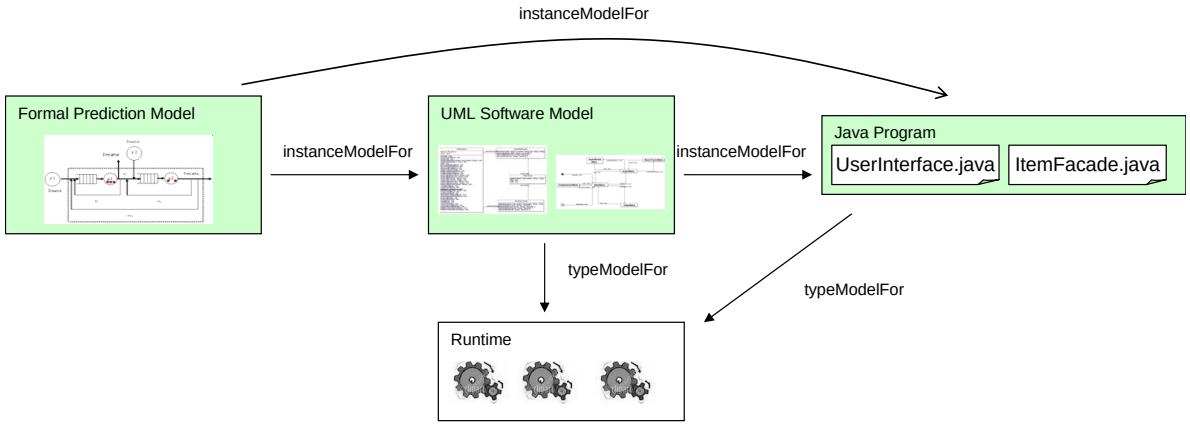


Figure 6.8.: Relations between UML software prescriptive model, formal performance prediction model, and Java application at runtime

The formal prediction model is the instance model of the UML prescriptive model. As discussed before, a UML prescriptive model is a type model of an executing system. As discussed in Section 2.2, semantics of a UML model is given with the mapping to some already defined domain. In the ***MoDePeMART***, the semantics to the UML model is given with a mapping to code. As explained before, this relation is an ***instance model for*** relation. Following the rule of transitivity, the relation of formal performance prediction model to code is that the prediction model is an instance model for code constructs.

At the end, formal definition of the execution of running application needs to be done. In this approach we assume that a model of software systems models a control flow which can be replicated for the purpose of serving multiple users concurrently. The way that the concurrency is achieved is not of importance. It can be achieved with threads, processes or sessions. The execution of the system needs to be formally defined in order to precisely specify the language for measurement and assessment and to avoid it's misuse.

For the formal definition of the relation between the prescriptive model and the system execution, the discrete time model is adopted. In the discrete time model, the occurrence time of an event is marked by timestamps and it is isomorphic to natural numbers [Clifford and Tansel, 1985]. The time line is, in this model, divided into ordered closed time intervals, called chronons. A chronon is a smallest duration of time that we can represent, and it is an interval between two respective timestamps. The size of the chronon, or the amount of time it represents is called the chronon granularity.

Here should be noticed that when an event is marked with a timestamp, it means that that event happened during the chronon which starts at that particular timestamp. There could be two events that happened at different time, but if both of them happened during one chronon, they will be marked with the same time stamp [Dyreson and Snodgrass, 1993].

The execution of a software system can formally be defined as follows.

Let the M be *a set of all elements* of the software system functionality prescriptive model (left hand side model in Figure 5.1). Let Th be a set of non-communicating processes/threads/sessions for concurrent execution, i.e., processes/threads/sessions that do not exchange messages between themselves. And let $\boldsymbol{T}$ the set of all timestamps. Finally, let the Ti be $Ti = T \times T$. Then an execution of the software system $\boldsymbol{E}$ is:

$$E \subset M \times Th \times Ti$$

In common language an execution of a system is a set of executions of software parts modeled with modeling elements in M. These software parts execute in some threads/processes/sessions from Th in some time interval Ti. The first in the pair that is an element of the Ti denotes the start timestamp of that event. The second one, the end timestamp.

In this execution formulation the Cartesian product of model and set of threads/processes/session identifiers is used to represent possibility of concurrent execution of the same service. However, we consider that these services do not exchange information between themselves. An example of this kind of execution model is implemented in Session Beans of JEE systems.

With respect to time characterization of modeling elements, here it is considered that all modeling elements can be classified in two general types, elements which occur only at one time point in reality and, theoretically, do not have a duration, and elements which do. This characterization of elements is already widely used in temporal databases [Snodgrass, 2000]. In the formal model, for modeling executions of modeling elements with no duration, elements of Ti with equal values in pairs are used. The execution of a modeling element in a thread/process/session with no duration is represented with pairs whose first and second element differ.

7. Chapter
A Linguistic Metamodel for Performance Measurement and Assessment

The central contribution of this thesis is a performance measurement and assessment modeling language linguistic metamodel. This linguistic metamodel, MM_{pema}, defines a modeling language for declarative specification of execution context and metrics of interest. Later, that context and metrics are transformed into a code for measurement and metrics computation. This chapter describes the abstract syntax of the language in the form of sets. UML Class Diagrams are, in this chapter, used for visualization of the linguistic metamodel. Later, these UML Class diagrams are used as elements of the concrete syntax. The concrete syntax is implemented in the form of a UML Profile, and is described in Section 9.

Conceptually, the linguistic metamodel can be divided into the three parts: metrics, assessment, and context specification part. Formally, this is specified as:

$$MM_{pema} = MM_{metrics} \cup MM_{assessment} \cup MM_{context}$$

The metrics part of the linguistic metamodel defines metrics which are computed for later assessment. The assessment part of the linguistic metamodel is described in Section 7.2. This part enables specification of time intervals for which specified metrics are computed. Furthermore, it is used for specification of various statistical analysis of computed values. Finally, the context part of the linguistic metamodel, the part which facilitates specification of an event and its execution context, is described in Section 7.3. Besides explanation of the metamodel, this chapter describes its static semantic. The static semantic is defined in Section 7.4. The complete metamodel can also be found in Appendix B.

7.1. The Metrics Part of the Metamodel

Metrics for performance assessment defined in this linguistic metamodel correspond to performance definition of this thesis, and UML SPT and MARTE standard metrics. The metrics metamodel part is presented in Figure 7.1.

Formally, this metamodel part is defined as:

$$\begin{aligned} MM_{metrics} = \{ & Metric, OccurrenceRate, OcurrencePercentage, \\ & Analysis, StatisticalAnalysis, Statistic, Distribution, DistributionKind, \\ & IntervalSet, \end{aligned}$$

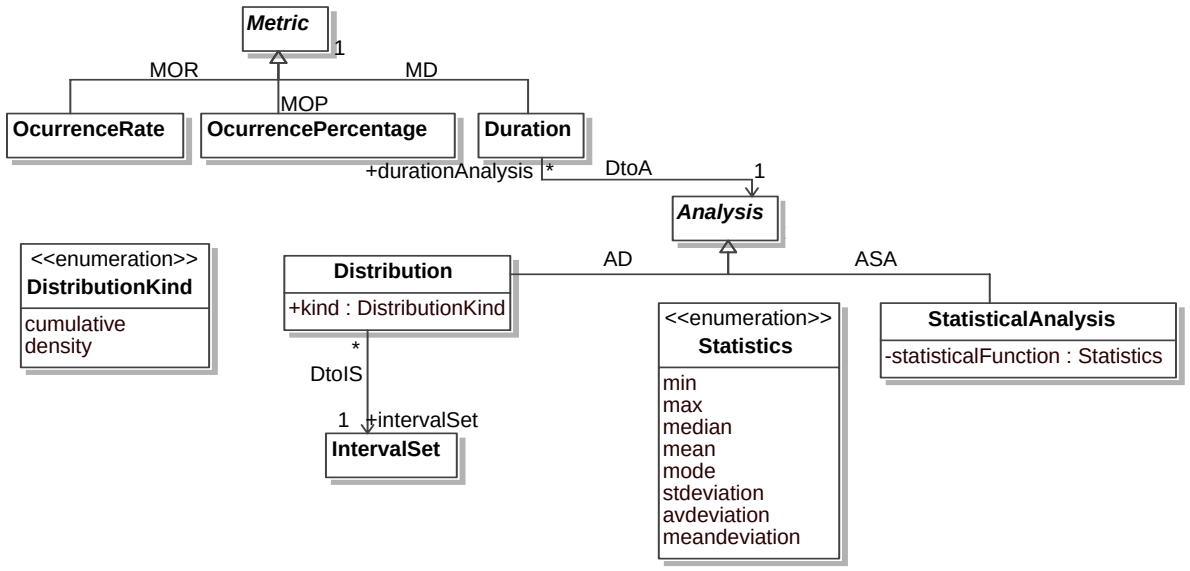


Figure 7.1.: Metrics part of the linguistic metamodel

$DtoA, DtoIS,$
$MOR, MOP, MD,$
$AD, ASA\}$

Metrics are, in the language defined with this lingusitic metamodel, specified as instances of metaclasses specialized from the ***Metric*** metaclass. Metaclass ***Metric*** is specialized into three metaclasses ***OcurranceRate***, ***OccurrencePercentage***, and ***Duration***.

The metaclass ***OccurrencePercentage*** is used for the specification of computation of execution percentage of each event in a set of predefined events. Event, in this thesis, is considered as execution of a modeling element in a particular execution context. Execution of a modeling element is actually an execution of a part of the application modeled with that modeling element.

In performance prediction probability of an event occurrence in a control flow is also considered when computing metrics estimates. For this reason, in definitions of execution sequences, SPT and MARTE UML profiles allow the annotation of each alternative in branching with the probability of execution. ***OccurrencePercentage*** is used for assessment of probability of execution of alternative sequences.

Performance, as defined in this thesis, is measured with throughput and response time. To throughput and response time correspond ***OcurranceRate*** and ***Duration*** metaclasses of this linguistic metamodel. Names of throughput and response time in the definition and SPT and MARTE standards originate from performance evaluation of transformational software systems. In reactive systems, a state does not have a response time. Instead, a system spends some time in a state and, therefore, it is more appropriate to call this characteristic a duration. For the similar reason it is better to rename throughput into occurrence rate. A state can not be a job, it can rather occur during execution. From the transformational perspective, this means that a duration of a service is a response time, and the occurrence rate of service execution is the throughput.

OcurranceRate is the metaclass which enables specification of computation of the num-

ber of events in a time unit. It is computed by dividing the number of occurrences in a time interval by the length of that interval. Specification of an interval length for which occurrence rate is measured is enabled with metaclasses from the assessment part of the metamodel, described in the next section.

Specification of measurement and computation of duration of the execution of a particular event is specified with the metaclass ***Duration***. The duration of execution of a part of program modeled by some modeling element is seldom a constant value. For this reason, duration is presented with some statistical function. The statistical function of interest can vary. In hard real time systems, of interest are minimum and maximum values, because all occurrences have to be above or below some predefined thresholds. Such real time system is the airbag system. If it an airbag is opened to early or too late it will not succeed in saving the person in the car. In soft real time systems, only a large enough number of occurrences have to satisfy required thresholds. In oder to show the level of satisfaction of thresholds, values like mean, median, standard deviation, average, mean deviation and so on, of duration are used. Furthermore, durations of events are often predicted and specified with density and cumulative distribution functions. For this reason, the ability for computation of summarizing statistics and density and cumulative distribution functions are needed. In order to provide previously stated requirements, in the metrics metamodel part, statistical analysis of a set of event durations is specified with the attribute ***durationAnalysis***, whose type is ***Analysis***. ***Analysis*** metaclass is later specialized into the ***StatisticalAnalysis*** and ***Distribution*** metaclasses.

StatisticalAnalysis is class for specification of a summarizing statistical function of interest. Set of statistical functions which can be specified are given in the enumeration ***Statistics***. This set of functions can be easily extended, and statistical functions which are not in this set added.

Distribution metaclass facilitates ability of measuring and computing distribution functions. Distribution function is defined in the attribute ***kind***. This attribute is of ***DistributionKind*** enumeration type and can be either ***cumulative*** or ***density***. These enumerations are used for specification of computation of histogram and cumulative histogram, respectively. For this reason attribute ***intervalSet***, of type ***IntervalSet*** always contains a set of intervals, which are cells of the resulting histogram. More on ***IntervalSet*** is given in Section 7.2.

In the following the definition of this linguistic metamodel in set theory is given. This is done in order to precisely define the relations between classes of this linguistic metamodel, and to define its abstract syntax.

Let ***Metric***, ***OcurrenceRate***, ***OcurrencePercentage*** and ***Duration*** be sets of metric, occurrence rate, occurrence percentage and duration specifications, respectively. Then specializations of the ***Metric*** metaclass to the ***OcurrenceRate***, ***OcurrencePercentage*** and ***Duration*** classes are defined as:

$$
\begin{aligned}
MOR = \{(m, or) : {} & (m \in Metric) \wedge (or \in OccurrenceRate) \wedge \\
& \nexists (m_1, or_1) \in MOR \\
& (((m = m_1) \wedge (or \neq or_1)) \vee ((m \neq m_1) \wedge (or = or_1))) \wedge \\
& \nexists op \in OccurrencePercentage, d \in Duration(((m, op) \in MOP) \vee \\
& ((m, d) \in MD))\},
\end{aligned}
$$

$$MOP = \{(m, op) : (m \in Metric) \wedge (op \in OccurrencePercentage) \wedge \nexists(m_1, op_1) \in MOP (((m = m_1) \wedge (op \neq op_1)) \vee ((m \neq m_1) \wedge (op = op_1))) \wedge \nexists or \in OccurrenceRate, d \in Duration(((m, or) \in MOR) \vee ((m, d) \in MD))\},$$

$$MD = \{(m, d) : (m \in Metric) \wedge (d \in Duration) \wedge \nexists(m_1, d_1) \in MD (((m = m_1) \wedge (d \neq d_1)) \vee ((m \neq m_1) \wedge (d = d_1))) \wedge \nexists or \in OccurrenceRate, op \in OccurrencePercentage (((m, or) \in MOR) \vee ((m, op) \in MOP))\},$$

$$Metric = \{m : \exists or \in OccurrenceRate, op \in OcurrencePercentage, d \in Duration (((m, or) \in MOR)) \vee ((m, op) \in MOP) \vee ((m, d) \in MD)\}.$$

$$OccurrenceRate = \{or : \exists(m_1, or_1) \in MOR(or = or_1)\}$$

$$OccurrencePercentage = \{op : \exists(m_1, op_1) \in MOP(op = op_1)\}$$

$$Duration = \{d : \exists(m_1, d_1) \in MD(d = d_1)\}$$

The previous set of definitions *shows the pattern for specifying a specialization relation* throughout this thesis! Instances of a metaclass which is a specialization of some other metaclasses are related to instances of the general metaclass. This relation is ***injective***, which means that each instance of a specialized metaclass can be in relation with only one instance of the general metaclass. Furthermore, an instance of the general metaclass can be in specialization relation with only one instance of specialized metaclasses. With this set of definitions it is achieved that an instance of specialized metaclass and an instance of the generalized metaclass form one separate unit, and can be treated as one separate object. This is actually the actual case in object oriented languages.

Association between ***Duration*** and ***Statistic*** is defined as relation ***DtoA***:

$$DtoA = \{(d, a) : (d \in Duration) \wedge (a \in Analysis) \wedge \nexists d_1 \in Duration, a_1 \in Analysis ((d = d_1) \wedge (a \neq a_1) \wedge (((d_1, a_1) \in DtoA))) \wedge \forall d_2 \in Duration \exists a_2 \in Analysis((d_2, a_2) \in DtoA)\}$$

The constraint given in the definition is for defining multiplicity of the association.

The ***Analysis*** metaclass specializes in two subclasses, ***Distribution*** and ***Statistical-Analysis***, and the specializations are defined similarly to the ***Metric*** metaclass:

$$AD = \{(a, di) : (a \in Analysis) \wedge (di \in Distribution) \wedge \nexists(a_1, di_1) \in AD(((a = a_1) \wedge (di \neq di_1)) \vee ((a \neq a_1) \wedge (di = di_1))) \wedge \nexists sa \in StatisticalAnalysis((a, sa) \in ASA)\},$$

$$ASA = \{(a, sa) : (a \in Analysis) \wedge (sa \in StatisticalAnalysis) \wedge$$

$$\nexists(a_1, sa_1) \in ASA$$
$$(((a = a_1) \wedge (sa \neq sa_1)) \vee ((a \neq a_1) \wedge (sa = sa_1))) \wedge$$
$$\nexists d \in Distribution((a, d) \in AD)\},$$

$$Analysis = \{a : \exists di \in Distribution, sa \in StatisticalAnalysis$$
$$(((a, di) \in AD) \vee ((a, sa) \in ASA))\}$$

$$Distribution = \{di : \exists(a_1, di_1) \in AD(di = di_1)\}$$

The ***StatisticalAnalysis*** metaclas is defined as:

$$StatisticalAnalysis = \{sa : sa = (statisticalFunction) \wedge$$
$$(statisticalFunction \in Statistics) \wedge$$
$$\exists(a_1, sa_1) \in ASA(sa = sa_1)\}.$$

The enumeration ***Statistics*** is defined as:

$Statistics = \{min, max, median, ..., stdeviation, avdeviation, meandeviation\}$, and can be extended for an open set of statistical functions.

The metaclass ***Distribution*** is used for specifying distribution measurements and computation, and it is defined in the following.

Let ***DistributionKind*** be a set defined to specify what kind of distribution is desired to be computed defined as:

$$DistributionKind = \{density, cumulative\}.$$

Then the ***Distribution*** metaclass is defined as:

$$Distribution = \{distribution : distribution = (kind) \wedge kind \in DistributionKind\}.$$

Finally, the ***IntervalSet*** is used for the definition of distribution histogram bars. The association between ***Distribution*** and ***IntervalSet*** is defined as the relation ***DtoIS***:

$$DtoIS = \{(di, is) : (di \in Distribution) \wedge (is \in IntervalSet) \wedge$$
$$\nexists di_1 \in Distribution, is_1 \in IntervalSet((di = di_1) \wedge (is \neq is_1) \wedge$$
$$(((di_1, is_1) \in DtoIS))) \wedge$$
$$\forall di_2 \in Distribution \exists is_2 \in IntervalSet((di_2, a_2) \in DtoIS)\}.$$

7.2. The Assessment Part of the Linguistic Metamodel

The assessment part of the linguistic metamodel enables specification of metrics computation and time intervals for which these metrics are computed. Furthermore, it enables

specification of metrics computed for particular time intervals. This is formaly specified as

$$MM_{assessment} = MM_{computation} \cup MM_{intervalset}$$

. The assessment linguistic metamodel part is presented in Figure 7.2

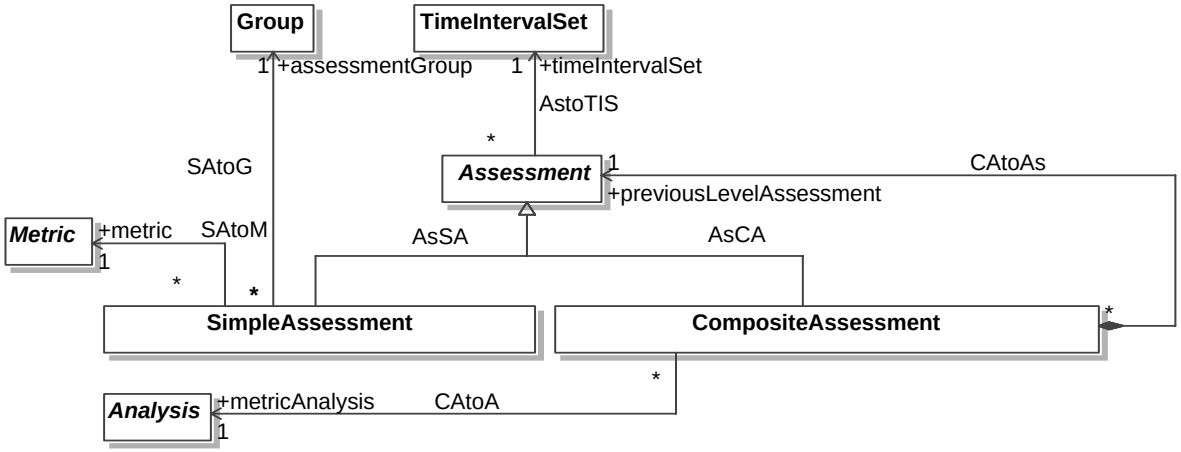


Figure 7.2.: Analysis part of the linguistic metamodel

Formally, $MM_{computation}$ is defined as:
$MM_{computation} = \{Assessment, SimpleAssessment, CompositeAssessment,$
$Metric, Analysis, Group, TimeIntervalSet,$
$AstoTIS, SAtoM, SAtoG, CAtoA, CAtoAs$
$AsSA, AsCA,\}$

Metaclasses ***Metric*** and ***Analysis*** are defined in the previous section. The central metaclass of this linguistic metamodel for measurement and assessment is the ***Assessment*** metaclass. This metaclass is the root of each measurement and metric computation specification. It specializes in two metaclasses ***SimpleAssessment*** and ***CompositeAssessment***. Formally, this is defined as:

$$AsSA = \{(as, sas) : (as \in Assessment) \wedge (sa \in SimpleAssessment) \wedge \nexists(as_1, sas_1) \in AsSA (((as = as_1) \wedge (sas \neq sas_1)) \vee ((as \neq as_1) \wedge (sas = sas_1))) \wedge \nexists ca \in CompositeAssessment((as, ca) \in AsCA)\},$$

$$AsCA = \{(as, ca) : (as \in Assessment) \wedge (ca \in CompositeAssessment) \wedge \nexists(as_1, cas_1) \in AsCA (((as = as_1) \wedge (cas \neq cas_1)) \vee ((as \neq as_1) \wedge (cas = cas_1))) \wedge \nexists sa \in SimpleAssessment((as, sa) \in AsSA)\},$$

$$Assessment = \{as : \exists sa \in SimpleAssessment, ca \in CompositeAssessment (((as, sa) \in ASsA) \vee ((as, sa) \in AsCA))\}.$$

$$SimpleAssessment = \{sas : \exists(as_1, sas_1) \in AsSA(sas = sas_1)\}$$

$$CompositeAssessment = \{cas : \exists(as_1, cas_1)(cas = cas_1)\}$$

SimpleAssessment is the metaclass for specification of basic metric computations. Basic metric computations are computations of occurrence percentage, statistical value of duration, and occurrence rate for specified event sets in specified time intervals. An example of basic measurements and metric computation are measurements and computation of occurrence rate of each minute of a day.

Specification of metrics, time intervals, and event sets, is facilitated with attributes ***metric***, ***timeIntervalSet*** and ***assessmentGroup*** of types ***Metric***, ***TimeIntervalSet***, and ***Group***, respectively. ***Metric*** metaclass is explained in previous section, ***TimeIntervalSet*** metaclass enable specification of set of time intervals for which the desired metric is computed. ***Group*** metaclass groups events for which the desired metric is computed. For example, with an instance of the ***Group*** metaclass, a set of events for which occurrence percentages are computed can be specified. Or in the case of computation of a throughput of a component, with an instance of ***Group*** metaclass all methods of that component would be grouped.

The association between ***SimpleAssessment*** and ***Metric*** is formally specified as relation ***SAtoM***:

$$\begin{aligned} SAtoM = \{(sa, m) : (sa \in SimpleAssessment) \wedge (m \in Metric) \\ \nexists sa_1 \in SimpleAssessment, m_1 \in Metric \\ ((sa = sa_1) \wedge (m \neq m_1) \wedge (((sa_1, m_1) \in SAtoM))) \wedge \\ \forall sa_2 \in SimpleAssessment \exists m_2 \in Metric((sa_2, m_2) \in SAtoM)\} \end{aligned}$$

Formal specification of the association between ***SimpleAssessment*** and ***Group*** is the relation ***SAtoG***. The relation ***SAtoG*** is formally defined as:

$$\begin{aligned} SAtoG = \{(sa, g) : (sa \in SimpleAssessment) \wedge (m \in Group) \\ \nexists sa_1 \in SimpleAssessment, g_1 \in Group \\ ((sa = sa_1) \wedge (g \neq g_1) \wedge (((sa_1, g_1) \in SAtoG))) \wedge \\ \forall sa_2 \in SimpleAssessment \exists g_2 \in Group((sa_2, g_2) \in SAtoG)\} \end{aligned}$$

Finally, in the same way is specified the association between ***Assessment*** and ***TimeIntervalSet***, entitled ***AtoTIS***:

$$\begin{aligned} AtoTIS = \{(as, tis) : (as \in Assessment) \wedge (tis \in TimeIntervalSet) \\ \nexists as_1 \in Assessment, tis_1 \in TimeIntervalSet \\ ((as = as_1) \wedge (tis \neq tis_1) \wedge (((as_1, tis_1) \in AtoTIS))) \wedge \\ \forall as_2 \in Assessment \exists tis_2 \in TimeIntervalSet((as_2, tis_2) \in AtoTIS)\} \end{aligned}$$

In previously mentioned example, the computation of occurrence rate in each minute of a day, value of ***metric*** attribute of the ***SimpleAssessment*** metaclass instance would be an instance of the ***OccurrenceRate*** metaclass. Moreover, the ***timeIntervalSet*** attribute

would contain an instance of ***TimeIntervalSet*** metaclass. This ***TimeIntervalSet*** metaclass would contain a definition of a set of 1440 (24×60) one minute intervals. The TimeIntervalSet metaclass is explained later in this section. In the attribute ***assessmentGroup*** would be specified a set of events for which throughput is computed. The specification of event sets is explained in Section 7.3.

CompositeAssessment metaclass enables higher level of performance measurement and assessment. It uses a basic or a composite assessment, and computes statistical functions of metric values in those assessments. The values to be analyzed are specified by the ***previousLevelAssessment*** attribute, and the desired statistical analysis by the ***metricAnalysis*** attribute. For example, the density distribution of throughput is one composite assessment. In this case, a value of the ***previousLevelAssessment*** attribute could be previously mentioned simple assessment throughput for each minute of the day. The specified metric in the ***metricAnalysis*** attribute would then be the density distribution.

Finally, the composite assessment is also performed for a time interval, or interval set. For example, if it is of interest to measure and compute throughput density distribution of each hour of a day for one day. Time interval set is specified in the attribute ***timeIntervalSet***, which is inherited from the ***Assessment*** metaclass. In this case the value of the ***timeIntervalSet*** attribute would be an instance of ***TimeIntervalSet*** containing the definition of 24 one hour intervals. If an observation of the throughput density distribution for whole day period is of interest, this instance would contain only one interval of 24 hours. Formal specification of this metamodel part is in the following.

Relation ***CAtoAs*** is the abstract syntax definintion of the association between ***CompositeAssessment*** and ***Assessment***:

$$\begin{aligned} CAtoAs = \{(ca, as) : (ca \in CompositeAssessment) \wedge (as \in Assessment) \wedge \\ \nexists ca_1 \in CompositeAssessment, as_1 \in Assessment \\ ((ca = ca_1) \wedge (as \neq as_1) \wedge ((ca_1, as_1) \in CAtoAs)) \wedge \\ \forall ca_2 \in CompositeAssessment \exists as_2 \in Assessment((ca_2, as_2) \in CAtoAS)\} \end{aligned}$$

The association between ***CompositeAssessment*** and ***Analysis*** is formally defined as relation ***CAtoA***:

$$\begin{aligned} CAtoA = \{(ca, a) : (ca \in CompositeAssessment) \wedge (a \in Analysis) \wedge \\ \nexists ca_1, \in CompositeAssessment, a_1 \in Analysis \\ (ca = ca_1) \wedge (a \neq a_1) \wedge ((ca_1, a_1) \in CAtoA) \wedge \\ \forall ca_2 \in CompositeAssessment \exists a_2 \in Analysis((ca_2, a_2) \in CAtoA)\} \end{aligned}$$

Attribute ***metricStatistic*** of the metaclass ***CompositeAssessment*** is of type ***Analysis***. This metaclass is described in previous section. The ***Analysis*** metaclass specializes in ***Distribution*** and ***StatisticalAnalysis*** metaclasses as shown in Figure 7.1. For ***Distribution*** metaclass, the attribute ***intervalSet*** is presented in Figure 7.3.

This part of the abstract syntax is represented as:

$$\begin{aligned} MM_{intervalset} = \{&IntervalSet, TimeIntervalSet, RealNumbersIntervalSet, \\ &TInterval, RealNumbersInterval, TimeInstant, double \end{aligned}$$

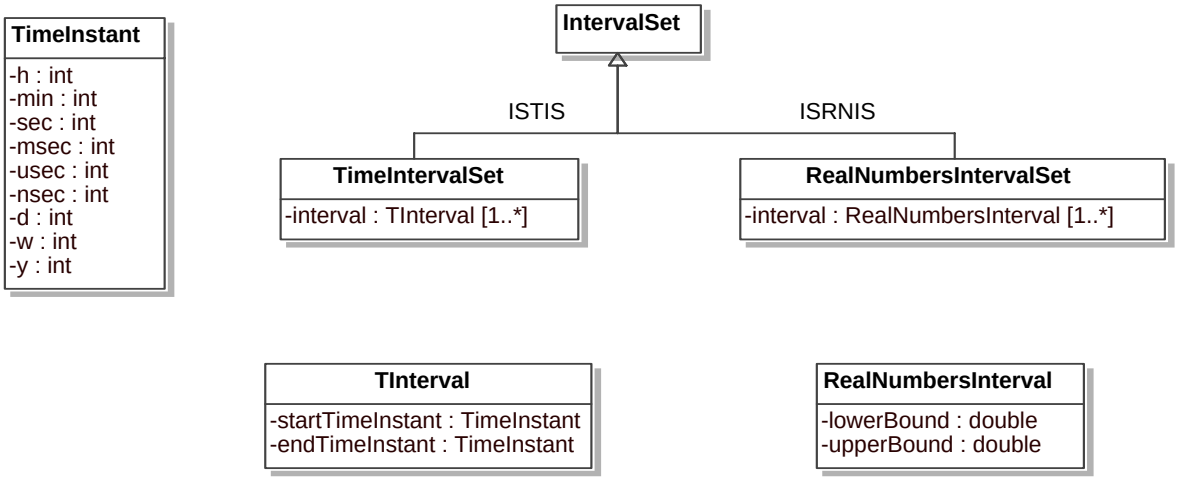


Figure 7.3.: Interval sets of the linguistic metamodel

$ISTIS, ISRNIS\}$

The datatype ***double*** is the standard datatype for real numbers, and it is not defined as a metaclass class in the Figure.

In this linguistic metamodel two kinds of sets of intervals for specification of measurements and metrics computation are distinguished, sets of time intervals, and sets of real numbers intervals. These sets are depicted with ***TimeIntervalSet*** and ***RealNumbersIntervalSet*** metaclasses, respectively. ***TimeIntervalSet*** class facilitates specification of sets of time intervals with a collection of ***TInterval*** instances in the ***interval*** attribute. A time interval is defined with two boundary time instants. Boundary time instant are in metaclass *TInterval* defined as ***startTimeInstant*** and ***endTimeInstant*** attributes of type ***TimeInstant***. The granularity of discrete time model can vary and variations result in changes of attributes of ***TimeInstant*** class.

Similarly to the ***TimeIntervalSet*** metaclass, ***RealNumbersIntervalSet*** has a collection of ***RealNumbersInterval***s in the attribute ***interval***. Each ***RealNumbersInterval*** is bounded with ***upperBound*** and ***lowerBound*** of type ***double***.

Let ***ISTIS*** and ***ISRNIS*** be relations of specialization defined at the end of this section. Formally, the metaclasses and their relations of this metamodel part are defined in the following.

$$TInterval = \{tInterval : tInterval = (startTimeInstant, endTimeInstant) \wedge (startTimeInstant \in TimeInstant) \wedge (endTimeInstant \in TimeInstant)\}$$

$$TimeIntervalSet = \{tis : (tis = (interval)) \wedge (interval \subseteq TInterval) \wedge (card(interval) \geq 1) \wedge \exists (is_1, tis_1) \in ISTIS(tis = tis_1)\}$$

$$RealNumbersInterval = \{(lowerBound, upperBound) : (lowerBound \in double) \wedge (upperBound \in double)\}$$

$$RealNumbersIntervalSet = \{rnis : (rnis = (interval)) \wedge (interval \subseteq RealNumbersInterval) \wedge (card(interval) \geq 1) \exists (is_1, rnis_1) \in ISRNIS(rnis = rnis_1)\}$$

The set ***TimeInstance*** is not defined due to possible changes of the granularity from project to project and system to system. For example, in information systems, the granularity of ***TimeInstance*** must not go below the size of a milisecond, while in real-time systems the granularity of nanoseconds might be of interest. Formal specification would be only a definition of elements of set ***TimeInstance*** as tuples with a component for each attribute. The set ***double*** is a representation of the $\mathbb{R}$ set.

Finally, the specializations from the ***IntervalSet*** metaclass to the ***TimeIntervalSet*** and ***RealNumbersIntervalSet*** metaclasses are defined like follows.

$$ISTIS = \{(is, tis) : (is \in IntervalSet) \wedge (tis \in TimeIntervalSet) \wedge \nexists (is_1, tis_1) \in ISTIS (((is = is_1) \wedge (tis \neq tis_1)) \vee ((is \neq is_1) \wedge (tis = tis_1))) \wedge \nexists rnis \in RealNumbersIntervalSet((is, rnis) \in ISRNIS)\},$$

$$ISRNIS = \{(is, rnis) : (is \in IntervalSet) \wedge (tis \in TimeIntervalSet) \wedge \nexists (is_1, rnis_1) \in ISRNIS (((is = is_1) \wedge (rnis \neq rnis_1)) \vee ((is \neq is_1) \wedge (rnis = rnis_1))) \wedge \nexists tis \in TimeIntervalSet((is, tis) \in ISTIS)\},$$

$$IntervalSet = \{is : \exists tis \in TimeIntervalSet, rnsi \in RealNumbersIntervalSet (((is, tis) \in ISTIS) \vee ((is, rnis) \in RNIS))\}.$$

7.3. The Event and Context Part of the Linguistic Metamodel

The event and context part of this linguistic metamodel enables specification of events for which metrics are measured and computed. In specification of an event, the definition of its execution context is of crucial importance. As previously defined, an event in an application is defined with a modeling element execution and its execution context. Execution context is sequence of events before and after the event of interest, also called scenario, and state of the system when the event of interest occurs. Formally, this part of the metmodel is specified as

$$MM_{context} = MM_{tcontext} \cup MM_{rcontext}$$

In the transformational systems the response time dependency on their invocation context is well studied by Rohr et al. [2008], Ammons et al. [1997], and Hamou-Lhadj and Lethbridge [2004]. An example of the need for the definition of performance measurement and assessment of an event in a particular scenario can be found in the electronic items management application. In Figure 6.2 at the page 45 is presented a workflow of the application

in the case of obtaining video files. When a user obtains a movie, ***getItem*** method of an instance of ***ItemFacade*** is invoked twice, once for the trailer and once for the movie. If the mean response time of that method is measured and computed independent of the invocation context, it would be done for both invocations together. Because the size of the trailer is significantly smaller than the size of the movie, the value of mean response time can be misleading. If the performance problem is in obtaining movie files, it would be hard to notice due to significantly smaller value of the mean response time. The linguistic metamodel has to facilitate the isolation of the invocation of ***getItem*** for obtaining a video file. In order to perform this, the contexts of ***getItem*** method invocation, in which it is assessed, must be specified.

Specification of assessment event of interest can be done using the following part of the linguistic metamodel, presented in Figure 7.4

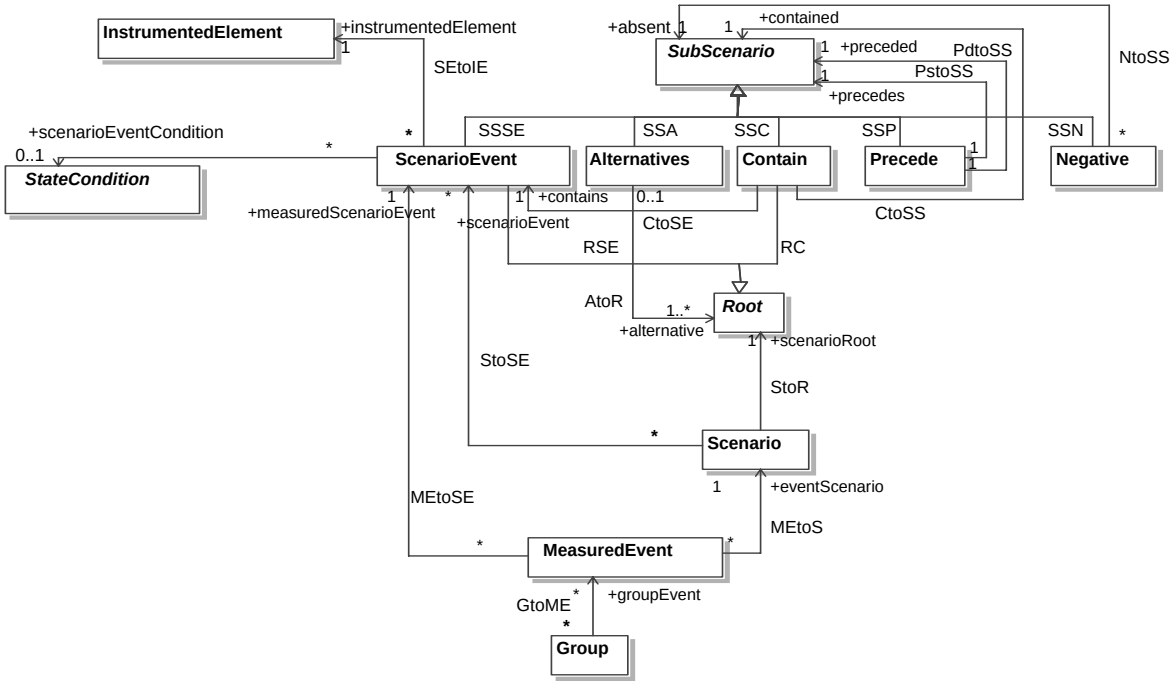


Figure 7.4.: Transformational context part of the linguistic metamodel

This metamodel part is formally defined as:

$$
\begin{aligned}
MM_{tcontext} = \{&Group, MeasuredEvent, Scenario, Root,\\
&InstrumentedElement, StateCondition,\\
&SubScenario, ScenarioEvent, Alternatives, Contain, Precede, Negative, int\\
&GtoME, MEtoSE, MEtoS, StoR, StoSE, AtoR, CtoSE, CtoSS,\\
&PdtoSE, PstoSE, NtoSE, SEtoIE, SEtoSC,\\
&RC, RSE, SSSE, SSA, SSC, SSP, SSN\}
\end{aligned}
$$

The ***int*** in the formal specification of the metamodel is the standard datatype for integers, and it is not presented in the Figure 7.4 because its existence is assumed.

The metaclass ***Group*** enables specification of set of events forming one measurement and assessment entity. Use cases for which SLA are defined can be designed in several scenarios and states. For example, if of interest for measurement and assessment is a throughput of a component. Then, all methods of that component have to be grouped and for this group the throughput computed. Or as another example, let us consider again the electronic item management application, explained in Chapter 6.1. Let us, more assume, that the service in SLA is defined for obtaining a music file. The obtaining of music file can be either an album track, explained as one of alternatives in Figure 6.3 at the page 46, or a music video, specified as one of options in Figure 6.2 at the page 45. The ***Group*** metaclass enables grouping these two use cases and computation of desired metrics for them. Formally it is defined in the following.

The relation ***GtoME*** is a formal model of the association between ***Group*** and ***MeasuredEvent***,

$$GtoME = \{(g, me) : (g \in Group) \wedge (me \in MeasuredEvent)\}$$

MeasuredEvent metaclass enables specification of execution context of an event and the event of interest for assessment. It contains two attributes ***eventScenario*** of type ***Scenario*** for specification of context of event of interest and ***measuredScenarioEvent*** of type ***ScenarioEvent*** for specification of the event of interest within previously specified context. Formally, this is defined as follows.

The association between ***MeasuredEvent*** and ***Scenario*** is formally defined as the relation ***MEtoS***:

$$\begin{aligned} MEtoS = \{(me, s) : (me \in MeasuredEvent) \wedge (s \in Scenario) \wedge \\ \nexists me_1, \in MeasuredEvent, s_1 \in Scenario \\ (me = me_1) \wedge (s \neq s_1) \wedge ((me_1, s_1) \in MEtoS) \wedge \\ \forall me_2 \in MeasuredEvent \exists s_2 \in Scenario((me_2, s_2) \in MEtoS)\} \end{aligned}$$

And the association between ***MeasuredEvent*** and ***ScenarioEvent*** is formally defined as the relation ***MEtoSE***.

$$\begin{aligned} MEtoSE = \{(me, se) : (me \in MeasuredEvent) \wedge (se \in ScenarioEvent) \wedge \\ \nexists me_1, \in MeasuredEvent, se_1 \in ScenarioEvent \\ (me = me_1) \wedge (se \neq se_1) \wedge ((me_1, se_1) \in MEtoSE) \wedge \\ \forall me_2 \in MeasuredEvent \exists se_2 \in ScenarioEvent((me_2, se_2) \in MEtoSE)\} \end{aligned}$$

The ***Scenario*** metaclass enables specification of the context in which desired event occurs. This metaclass corresponds to the MARTE UML Profile ***BehaviorScenario*** class. A context or a scenario is specified with a set of modeling elements whose executions form the scenario and relations between these events. The set of events is specified in the attribute ***scenarioEvent*** of type ***ScenarioEvent***. Their relations definition start with the attribute ***scenarioRoot*** of type ***Root***. The ***ScenarioEvent*** and the ***Root*** metaclasses are explained later in this section. Formal definition of the associations defining the attributes

of the ***Scenario*** metaclass is in the following.

The association between ***Scenario*** and ***Root*** is defined as relation ***StoR***

$$StoR = \{(s,r) : (s \in Scenario) \wedge (s \in Root) \wedge \\ \nexists s_1, \in Scenario, r_1 \in Root \\ (((s = s_1) \wedge (r \neq r_1)) \vee ((s \neq s_1) \wedge (r = r_1))) \wedge ((s_1, r_1) \in StoR) \wedge \\ \forall s_2 \in Scenario \exists r_2 \in Root((s_2, r_2) \in StoR)\}$$

And the association between ***Scenario*** and ***ScenarioEvent*** is defined as the relation ***StoSE***

$$StoSE = \{(s,se) : (s \in Scenario) \wedge (se \in ScenarioEvent) \wedge \\ \nexists s_1 \in Scenario, se_1 \in ScenarioEvent \\ (s = s_1) \wedge (se \neq se_1) \wedge ((s_1, se_1) \in StoSE) \wedge \\ \forall s_2 \in Scenario \exists se_2 \in ScenarioEvent((s_2, se_2) \in StoSE)\}$$

A scenario always starts with an invocation of a modeling element. For this reason, the ***Root*** metaclass is specialized into two metaclasses: (1) ***ScenarioEvent***, which models scenario that consists of only one invocation, and (2)***Contain***, for specification of scenarios with more than one invocation. Formally:

$$RSE = \{(r,se) : (r \in Root) \wedge (se \in ScenarioEvent) \wedge \\ \nexists (r_1, se_1) \in RSE \\ (((r = r_1) \wedge (se \neq se_1)) \vee ((r \neq r_1) \wedge (se = se_1))) \wedge \\ \nexists c \in Contain((r,c) \in RC)\},$$

$$RC = \{(r,c) : (r \in Root) \wedge (c \in Contain) \wedge \\ \nexists (r_1, c_1) \in RC \\ (((r = r_1) \wedge (c \neq c_1)) \vee ((r \neq r_1) \wedge (c = c_1))) \wedge \\ \nexists se \in ScenarioEvent((r, se) \in RSE)\},$$

$$Root = \{r : \exists se \in ScenarioEvent, c \in Contain \\ (((r, se) \in RSE) \vee ((r, c) \in RC))\}$$

Metaclasses ***Contain*** and ***ScenarioEvent*** are formally defined later in this section.

The ***Contain*** metaclass defines invocation composition of model elements, explained in Section 6.1. Invocation composition is the composition where the execution of one event starts after the beginning and ends before the end of the execution of the invoking event. For this reason, the ***Contains*** metaclass has two attributes ***contains*** and ***contained***. The attribute ***contains*** is the event which invokes and ***contained*** is the invoked event. The event which invokes has to be a modeling element execution, specified with an ***ScenarioElement*** instance, and the invoked event can be an instance of any metaclass specialized from the ***SubScenario*** metaclass.

The ***SubScenario*** metaclass enables specification and composition of basic scenario units. Scenario composition units are elements for transformational system specification defined

in Section 6.1. These units are ***command execution***, ***guarded command***, ***guarded command set***, and ***loop***. They are composed with invocation and sequential composition relations. In order to enable specification of relations and basic scenario units, ***SubScenario*** metaclass is specialized to five metaclasses ***Contain***, ***Precede***, ***Alternatives***, ***Negative*** and ***ScenarioEvent***.

Composite and simple commands measurements are specified with instances of the metaclass ***ScenarioEvent***.

For specification of the ***invocation relation***, as already mentioned, the metaclass ***Contain*** is used. The ***sequential composition*** between scenario elements is modeled using instances of the ***Precede*** metaclass. The ***Precede*** metaclass has two attributes, ***precedes*** and ***preceded***, both of type ***SubScenario***. The attribute ***precedes*** specifies an instance of ***SubScenario*** which begins and ends before the start of the program part modeled with an instance of ***SubScenario*** defined in ***preceded*** attribute.

Guarded command sets are specified with instances of the metaclass ***Alternatives***. Each guarded actions in a guarded command set is modeled as an alternative sub scenarios. Alternative sub scenarios are defined in the ***alternative*** attribute. The attribute ***alternative*** is of type ***Root*** because each alternative block in a sub scenario is a scenario itself and starts with a root.

Alternative execution sequences in guarded actions are specified with usage of an instance of the ***Negative*** metaclass. An example can be found in the case study, in Figure 6.2 at the page 45. Let us assume that response time of ***getItem*** method is of interest. To measure this execution, it has to be assured not only that ***getItem*** executes in the ***getVideoItem*** method execution, but also that option block did not execute. For this reason, scenario would be specified like ***getVideoItem*** method which contains a precede relation between negation of option block and ***getItem*** method, and the measured event would be the ***getItem*** invocation. In this way, ***getItem*** items which are contained in the option block would be removed from computing. Negation is necessary, because in the case when a scenario would be defined like ***getVideoItem*** which contains ***getItem***, and ***getItem*** marked as the measured event, ***getItem*** executions in the option block would be also integrated in computation, because its execution starts after and ends before the ***getVideoItem*** execution. The formal semantics of the language explains this in detail.

Loops are specified with an instance of the ***Contain*** metaclass. The relation between a loop and a sub scenario inside the loop is invocation, because loop starts and executions of elements inside are started and ended before the loop ends. Finally, ***ScenarioEvent*** is used for specification of invocation.

Formally, these metaclasses and their interrelations are defined as:

$$
\begin{aligned}
SSSE = \{(ss, se) : (ss \in SubScenario) \wedge (se \in ScenarioEvent) \wedge \\
\nexists (ss_1, se_1) \in SSSE \\
(((ss = ss_1) \wedge (se \neq se_1)) \vee ((ss \neq ss_1) \wedge (se = se_1))) \wedge \\
\nexists a \in Alternatives((ss, a) \in SSA) \wedge \\
\nexists c \in Contain((ss, c) \in SSC) \wedge \\
\nexists p \in Precede((ss, p) \in SSP) \wedge \\
\nexists n \in Negation((ss, n) \in SSN)\}
\end{aligned}
$$

$$
SSA = \{(ss, a) : (ss \in SubScenario) \wedge (a \in Alternatives) \wedge
$$

$$\nexists(ss_1, a_1) \in SSA$$
$$(((ss = ss_1) \wedge (a \neq a_1)) \vee ((ss \neq ss_1) \wedge (a = a_1))) \wedge$$
$$\nexists se \in ScenarioEvent((ss, se) \in SSSE) \wedge$$
$$\nexists c \in Contain((ss, c) \in SSC) \wedge$$
$$\nexists p \in Precede((ss, p) \in SSP) \wedge$$
$$\nexists n \in Negation((ss, n) \in SSN)\}$$

$$SSC = \{(ss, c) : (ss \in SubScenario) \wedge (a \in Contain) \wedge$$
$$\nexists(ss_1, c_1) \in SSC$$
$$(((ss = ss_1) \wedge (c \neq c_1)) \vee ((ss \neq ss_1) \wedge (c = c_1))) \wedge$$
$$\nexists se \in ScenarioEvent((ss, se) \in SSSE) \wedge$$
$$\nexists a \in Alternatives((ss, a) \in SSA) \wedge$$
$$\nexists p \in Precede((ss, p) \in SSP) \wedge$$
$$\nexists n \in Negation((ss, n) \in SSN)\}$$

$$SSP = \{(ss, p) : (ss \in SubScenario) \wedge (p \in Precede) \wedge$$
$$\nexists(ss_1, p_1) \in SSP$$
$$(((ss = ss_1) \wedge (p \neq p_1)) \vee ((ss \neq ss_1) \wedge (p = p_1))) \wedge$$
$$\nexists se \in ScenarioEvent((ss, se) \in SSSE) \wedge$$
$$\nexists a \in Alternatives((ss, a) \in SSA) \wedge$$
$$\nexists c \in Contain((ss, c) \in SSC) \wedge$$
$$\nexists n \in Negation((ss, n) \in SSN)\}$$

$$SSN = \{(ss, n) : (ss \in SubScenario) \wedge (n \in Negation) \wedge$$
$$\nexists(ss_1, n_1) \in SSN$$
$$(((ss = ss_1) \wedge (n \neq n_1)) \vee ((ss \neq ss_1) \wedge (n = n_1))) \wedge$$
$$\nexists se \in ScenarioEvent((ss, se) \in SSSE) \wedge$$
$$\nexists a \in Alternatives((ss, a) \in SSA) \wedge$$
$$\nexists c \in Contain((ss, c) \in SSC) \wedge$$
$$\nexists p \in Precede((ss, p) \in SSP)\}$$

$$SubScenario = \{ss :$$
$$\exists se \in ScenarioEvent, a \in Alternatives, c \in Contain, p \in Precede, n \in Negation$$
$$(((ss, se) \in SSSE) \vee ((ss, a) \in SSA) \vee ((ss, c) \in SSC) \vee$$
$$((ss, p) \in SSP) \vee ((ss, n) \in SSN))\}$$

$$ScenarioEvent = \{se : (se = (id)) \wedge (id \in int) \wedge$$
$$\exists(r_1, se_1) \in RSE(se = se_1) \wedge$$
$$\exists(ss_2, se_2) \in SSSE(se = se_2)\}$$

Instances of the ***ScenarioEvent*** metaclass have to be related to instances of the both metaclasses it specializes. Furthermore, the ***id*** attribute of this metaclass is identifier for the instances of the ***ScenarioEvent*** metaclasses. Herewith can be modeled several instances related to the same instrumented element and state context. This is needed for facilitating the well-formedeness rule 7.

$$Alternatives = \{a : \exists(ss_1, a_1) \in SSA(a = a_1)\}$$

$$Contain = \{c : \exists(ss_1, c_1) \in SSC(c = c_1) \wedge \exists(r_2, c_2) \in RC(c = c_2)\}$$

The Contain metaclass specializes Root and SubScenario metaclasses. For this reason it has previously defined rules on its elements.

$$Precede = \{p : \exists(ss_1, p_1) \in SSP(p = p_1)\}$$

$$Negation = \{n : \exists(ss_1, n_1) \in SSP(n = n_1)\}$$

The association between the metaclass ***Contain*** and ***ScenarioEvent*** which defines ***contains*** attribute of ***Contain*** class is defined with the relation ***CtoSE***:

$$CtoSE = \{(c, se) : (c \in Contains) \wedge (se \in ScenarioEvent) \wedge \nexists c_1 \in Contains, se_1 \in ScenarioEvent ((c = c_1) \wedge (se \neq se_1) \wedge ((c_1, se_1) \in CtoSE)) \wedge \forall c_2 \in Contains \exists se_2 \in ScenarioEvent((c_2, se_2) \in CtoSE)\}$$

The relation ***CtoSS*** formally defines the association between metaclasses ***Contain*** and ***SubScenario***. This relation specifies the attribute ***contained***:

$$CtoSS = \{(c, ss) : (c \in Contain) \wedge (ss \in SubScenario) \wedge \nexists c_1 \in Contain, ss_1 \in SubScenario ((c = c_1) \wedge (ss \neq ss_1) \wedge ((c_1, ss_1) \in CtoSS)) \wedge \forall c_2 \in Contains \exists ss_2 \in SubScenario((c_2, ss_2) \in CtoSS)\}$$

Associations which define attributes ***precedes*** and ***preceded*** of the metaclass ***Precede*** are defined with relations ***PstoSS*** and ***PdtoSS***, respectively:

$$PstoSS = \{(p, ss) : (c \in Precede) \wedge (ss \in SubScenario) \wedge \nexists p_1 \in Precede, ss_1 \in SubScenario ((p = p_1) \wedge (ss \neq ss_1) \wedge ((p_1, ss_1) \in PstoSS)) \wedge \forall p_2 \in Precede \exists ss_2 \in SubScenario((p_2, ss_2) \in PstoSS)\}$$

$$PdtoSS = \{(p, ss) : (p \in Precede) \wedge (ss \in SubScenario) \wedge \nexists p_1 \in Precede, ss_1 \in SubScenario ((p = p_1) \wedge (ss \neq ss_1) \wedge ((p_1, ss_1) \in PdtoSS)) \wedge \forall p_2 \in Precede \exists ss_2 \in SubScenario((p_2, ss_2) \in PdtoSS)\}$$

The formal definition of the association which defines the attribute ***alternative*** of the metaclass ***Alternatives*** is given with the relation ***AtoR***:

$$AtoR = \{(al, r) : (al \in Alternatives) \wedge (r \in Root) \wedge \nexists al_1 \in Alternatives, r_1 \in SubScenario$$

$$((al \neq al_1) \wedge (r = r_1) \wedge ((al_1, r_1) \in AtoR)) \wedge$$
$$\forall al_2 \in Alternatives \exists r_2 \in Root((al_2, r_2) \in AtoR)\}$$

Finally, the association which defines the attribute ***absent*** of the ***Negative*** metaclass is formally specified with the ***NtoSS*** relation:

$$NtoSS = \{(n, ss) : (n \in Negative) \wedge (ss \in SubScenario) \wedge$$
$$\nexists n_1 \in Negative, ss_1 \in SubScenario$$
$$((n = n_1) \wedge (ss \neq ss_1) \wedge ((n_1, ss_1) \in NtoSS)) \wedge$$
$$\forall n_2 \in Negative \exists ss_2 \in SubScenario((n_2, ss_2) \in NtoSS)\}$$

Associations between ***ScenarioEvent*** and ***InstrumentedElement*** and ***StateCondition*** metaclasses are defined later in this section.

In systems with interwoven transformational and reactive part, not only invocation sequence, but also the state of the system defines the context of an event. For example, in the case study, the UML State Diagram in Figure 6.5 specifies possible states of an ***ItemFacade*** class instance. An ***ItemFacade*** instance can be either in the state where compression is turned on or turned off. If an obtaining movie is again observed, it can be noticed that turning on compression can differently influence response times of obtaining a trailer and obtaining a movie. If the trailer's size is small, then the compression and decompression routines and the transmission of compressed data might take more time than a transmission of non-compressed data. Therefore, the response time can be increased. In the case of obtaining a movie, the response time can be lower, because the compression routine and transmission of compressed data is certainly lower than transmission of non-compressed data. Here the importance of the distinction between states and variables, argued in Section 6.1 at the page 42, in performance measurement and assessment becomes evident. States represent qualitative functional aspects of a system [Selic et al., 1994], and the dependency of the performance metrics on them is analyzed with observation of metrics in certain context in which those states take part. On the other hand, variables represent quantitative functional aspects of system [Selic et al., 1994], and performance metrics dependency of method invocations from variables is being approximated with statistical functions.

For these reasons, for each invocation the state of the system should be specified. The part of the linguistic metamodel that enables state specification with invocation is presented in the Figure 7.5.

This part of the metamodel is formally defined as:

$$MM_{rcontext} = \{ScenarioEvent, InstrumentedElement, StateCondition,$$
$$Binary, AND, OR, NOT, ConditionElement, ConditionRelation,$$
$$SEtoIE, SEtoSC, CEtoIE, BtoSlo, BtoSro, NtoSC,$$
$$SCB, SCN, SCCE, BA, BC\}$$

The metaclass ***ScenarioEvent*** enables specification of a modeling element execution in some state of a system. Like previously stated, an event of a scenario is defined by a scenario in which it executes, executing modeling element, and the state of the system in which that modeling element executes. While specification of a scenario is facilitated in the metamodel part in Figure 7.4, the metamodel part in Figure 7.5 enables the specification the state and the modeling element of an event. A modeling element, is defined in attribute ***instrumentedElement*** of type ***InstrumentedElement***. The state of the system, is specified in the

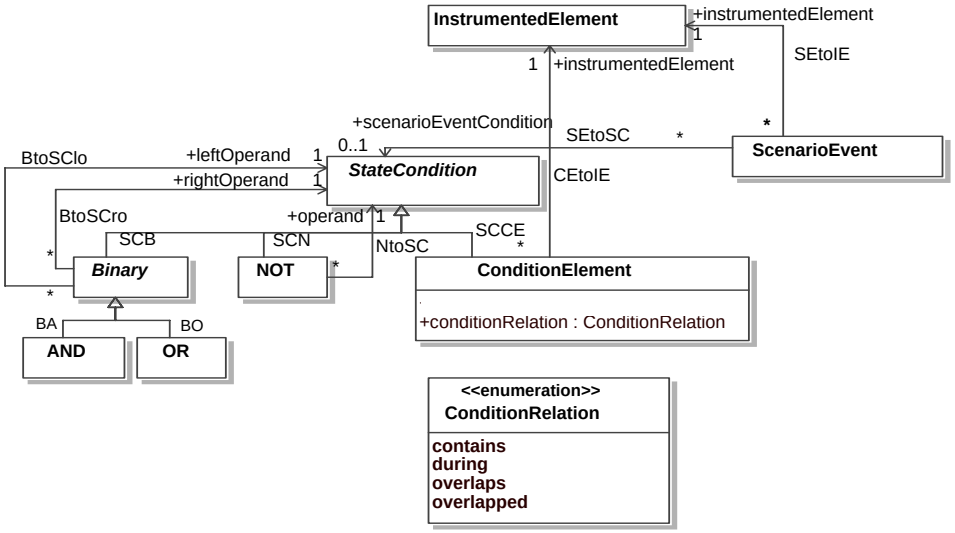


Figure 7.5.: The reactive context part of the metamodel

attribute ***scenarioEventCondition***. Formally this is defined in the following.

The association which defines ***instrumentedElement*** attribute of ***ScenarioEvent*** is formally specified as the relation ***SEtoIE***

$$\begin{aligned} SEtoIE = \{(se, ie) : (se \in ScenarioEvent) \wedge (ie \in InstrumentedElement) \wedge \\ \nexists se_1 \in ScenarioEvent, ie_1 \in InstrumentedElement \\ ((se = se_1) \wedge (ie \neq ie_1) \wedge ((se_1, ie_1) \in SEtoIE)) \wedge \\ \forall se_2 \in ScenarioEvent \exists ie_2 \in InstrumentedElement \\ ((se_2, ie_2) \in SEtoIE)\} \end{aligned}$$

The relation ***SEtoSC*** formally specifies the association between ***ScenarioEvent*** and ***StateCondition***. This relation defines the ***scenarioElementCondition*** attribute of the class ***ScenarioEvent***:

$$\begin{aligned} SEtoSC = \{(se, sc) : (se \in ScenarioEvent) \wedge (sc \in StateCondition) \wedge \\ \nexists se_1 \in ScenarioEvent, sc_1 \in StateCondition \\ (((se = se_1) \wedge (sc \neq sc_1) \wedge ((se_1, sc_1) \in SEtoSC)))\} \end{aligned}$$

Specification of the current state of the system is specified as a boolean algebra of active states. The boolean algebra consists of ***StateCondition*** metaclass and classes specialized from it ***Binary***, ***AND***, ***OR***, ***NOT***, and the ***ConditionElement***. Metaclasses ***AND***, ***OR***, and ***NOT***, represent Boolean operations with the same name.

The metaclass ***ConditionElement*** is the metaclass which facilitates specifications of variable in this boolean algebra, and it is used for specification of the state and its relation to the potential event of scenario. This is specified in two attributes ***instrumentedElement*** of type ***InstrumentedElement*** and ***conditionRelation*** of type ***ConditionRelation***.

ConditionRelation is the enumeration which contains possible relations between occurrence of a system state and occurrence of the execution of a modeling element specifying

event. The relations in the enumeration are based on relations between intervals explained in Appendix D. Having in mind the constraint/assumption of one thread for each scenario execution, defined in Section 6.2, the relations ***before*** and ***after*** are relations contained in scenario specification and therefore they are not needed in state context specification. ***Starts***, ***started-by***, ***finishes***, ***finished-by*** and ***equals*** are not part of relations because of the assumption that granularity of timing mechanism is large enough so that each execution of the part of application modeled with a modeling element starts and ends in different chronon. ***Meets*** and ***met-by***, are specific cases of ***before*** and ***after***, and for that reason they are not included in the final set of possible relations. Formally, this part of the model is defined as following:

$$SCB = \{(sc, b) : (sc \in StateCondition) \wedge (b \in Binary) \wedge \nexists(sc_1, b_1) \in SCB (((sc = sc_1) \wedge (b \neq b_1)) \vee ((sc \neq sc_1) \wedge (b = b_1))) \wedge \nexists n \in NOT, ce \in ConditionElement (((se, n) \in SCN) \wedge ((se, ce) \in SCCE))\},$$

$$SCN = \{(sc, n) : (sc \in StateCondition) \wedge (n \in NOT) \wedge \nexists(sc_1, n_1) \in SCN (((sc = sc_1) \wedge (n \neq n_1)) \vee ((sc \neq sc_1) \wedge (n = n_1))) \wedge \nexists b \in Binary, ce \in ConditionElement (((se, b) \in SCB) \wedge ((se, ce) \in SCCE))\},$$

$$SCCE = \{(sc, ce) : (sc \in StateCondition) \wedge (ce \in ConditionElement) \wedge \nexists(sc_1, ce_1) \in SCCE (((sc = sc_1) \wedge (ce \neq ce_1)) \vee ((sc \neq sc_1) \wedge (ce = ce_1))) \wedge \nexists n \in NOT, b \in Binary(((se, n) \in SCN) \wedge ((se, b) \in SCB))\},$$

$$StateCondition = \{sc : \forall sc \exists b \in Binary, n \in Not, ce \in ConditionElement (((sc, b) \in SCB) \vee ((sc, n) \in NOT) \vee ((sc, ce) \in ConditionEvent))\}$$

$$NOT = \{not : \exists(sc_1, not_1) \in SCN(not = not_1))\}$$

Metaclasses ***Binary*** and ***Condition*** element are specified later in this section.

$$BO = \{(b, o) : (b \in Binary) \wedge (o \in OR) \wedge \nexists(b_1, o_1) \in BO (((b = b_1) \wedge (o \neq o_1)) \vee ((b \neq b_1) \wedge (o = o_1))) \wedge \nexists a \in AND((b, a) \in BA)\},$$

$$BA = \{(b, a) : (b \in Binary) \wedge (a \in AND) \wedge \nexists(b_1, a_1) \in BA (((b = b_1) \wedge (a \neq a_1)) \vee ((b \neq b_1) \wedge (a = a_1))) \wedge \nexists o \in OR((b, o) \in BO)\},$$

$$Binary = \{b : \forall b \exists a \in AND, o \in OR(((b, a) \in BA) \vee ((b, or) \in OR)) \wedge$$

$$\nexists(cs_1, b_1) \in SCB$$
$$(((sc = sc_1) \wedge (b \neq b_1)) \vee ((sc \neq sc_1) \wedge (b = b_1)))\}$$
$$(((b, a) \in BA) \vee ((b, o) \in BO))\}$$

$$AND = \{and : \exists(b_1, and_1 \in BA(and = and_1)\}$$

$$OR = \{or : \exists(b_1, or_1 \in BA(or = or_1)\}$$

The formal specification of the associations between ***Binary*** and ***StateCondition*** metaclasses which define attributes ***leftOperand*** and ***rightOperand*** of the ***Binary*** metaclass are ***BtoSClo*** and ***BtoSCro*** relations, respectively:

$$BtoSClo = \{(b, sc) : (\in Binary) \wedge (sc \in StateCondition) \wedge$$
$$\nexists b_1 \in Binary, sc_1 \in StateCondition$$
$$(((b = b_1) \wedge (sc \neq sc_1) \wedge ((b_1, sc_1) \in BtoSClo)) \wedge$$
$$\forall b_2 \in Binary \exists sc_2 \in StateCondition((b_2, sc_2) \in BtoSClo)\}$$

$$BtoSCro = \{(b, sc) : (b \in Binary) \wedge (sc \in StateCondition) \wedge$$
$$\nexists b_1 \in Binary, sc_1 \in StateCondition$$
$$(((b = b_1) \wedge (sc \neq sc_1) \wedge ((b_1, sc_1) \in BtoSCro)) \wedge$$
$$\forall b_2 \in Binary \exists sc_2 \in StateCondition((b_2, sc_2) \in BtoSCro)\}$$

The final operation of this boolean algebra is negation. It is defined with the metaclass ***NOT***. The association which defines ***operand*** of this metaclass is specified with the relation ***NtoSC***:

$$NtoSC = \{(n, sc) : (n \in NOT) \wedge (sc \in StateCondition) \wedge$$
$$\nexists n_1 \in NOT, sc_1 \in StateCondition$$
$$(((n = n_1) \wedge (sc \neq sc_1) \wedge ((n_1, sc_1) \in NtoSC)) \wedge$$
$$\forall n_2 \in NOT \exists sc_2 \in StateCondition((n_2, sc_2) \in NtoSC)\}$$

The association between ***ConditionElement*** and ***InstrumentedElement*** metaclasses which defines ***instrumentedElement*** attribute of the metaclass ***ConditionElement*** is formally specified with relation ***CEtoIE***:

$$CEtoIE = \{(ce, ie) : (ce \in ConditionElement) \wedge (ie \in InstrumentedElement) \wedge$$
$$\nexists ce_1 \in ConditionElement, ie_1 \in InstrumentedElement$$
$$(((ce = ce_1) \wedge (ie \neq ie_1) \wedge ((ce_1, ie_1) \in CEtoIE)) \wedge$$
$$\forall ce_2 \in ConditionElement \exists ie_2 \in InstrumentedElement$$
$$((ce_2, ie_2) \in CEtoIE)\}$$

The ***ConditionRelation*** enumeration is defined as:

$$ConditionRelation = \{overlaps, overlapped, contains, during\}$$

Finally, the ***ConditionElement*** metaclass is defined as:

$$ConditionElement = \{ce : (ce = (cr)) \wedge (cr \in ConditionRelation) \wedge \\ \nexists (sc_1, ce_1) \in SCCE \\ (((sc = sc_1) \wedge (ce \neq ce_1)) \vee ((sc \neq sc_1) \wedge (ce = ce_1)))\}$$

7.4. The Static Semantics of the Metamodel

Engineering a language, as explained in Section 2.2, besides the definition of abstract and concrete syntax must contain the rules for well-formed language constructs. The abstract and the major elements of the concrete syntax have been specified in previous sections. This section contains the well-formedness rules. These well-formedness rules either cannot be adequately captured by a the abstract syntax, because there is no relation between elements of the metamodel and the granularity of timing mechanism, or their specification would be very complicated. For this reason, the well-formedness rules are defined in common language.

1. **The intervals in the interval set must be consecutive**
   Intervals in the interval set are used either for the grouping of events according to calender time of their occurrence, or grouping them for the distribution analysis. Both of these groupings require consecutive intervals,, i.e., set of intervals in which the end of one interval and the start of the next one are in two successive chronons.

2. **The value of the *intervalSet* attribute of an instance of class *Distribution* in the duration distribution analysis must be of type *TimeIntervalSet***
   Duration analysis is an analysis of time intervals, and time intervals can only be classified for the distribution analysis with some time interval values.

3. **The value of the *intervalSet* attribute in an instance of the *Distribution* metaclass in the composite assessment must be of type *RealNumbersIntervalSet* when the computed metric is *OcurrenceRate* or *OccurrencePercentage***
   Values of occurrence rate and occurrence percentage are real numbers, and any statistical analysis of them is also a real number. This implies that, for structuring values of occurrence rate and occurrence percentage for an distribution histogram computation must be based on the values of intervals of real numbers.

4. **The value of the *intervalSet* in the composite assessment must be of type *TimeIntervalSet* when the computed metric is duration**
   This rule is opposite to the previous well-formedness rule. A value of a duration and its statistical analysis are time intervals. Therefore, for the distribution histogram computation these values are grouped in time intervals they belong.

5. **A distribution assessment cannot be further statistically analyzed**
   The ***previousLevelAssessment*** attribute of an instance of ***CompositeAssessment*** cannot be a distribution assessment. In the distribution assessment the outcome of the computation is a set of distribution diagrams for each time interval specified in the ***timeInterval*** attribute of the ***Assessment*** metaclass. These set of values can not be further statistically analyzed.

6. **An event of interest in the specification of a measured event has to be a part of the scenario**
   The metaclass ***MeasuredEvent*** is used for specification of a measured event within some certain scenario. The scenario is specified with an instance of the ***Scenario*** metaclass and, therefore, the measured event has to be part of that scenario.

7. **Each sub scenario of a scenario must not share instances of *ScenarioEvent* with another sub scenario, although there is a need for a scenario event having same instrumented element and the reactive context**
   This well-formedenes rule is essential for valid computation of metrics. The need for this rule is explained in the Chapter 8.

8. **The *root* attribute of an instance of the *Scenario* metaclass must contain an event or a contain relation of that scenario**
   The ***root*** attribute specifies the beginning of the transformational context of an event. The transformational context can start in two ways. In the case when the transformational context is some scenario with more than one executing scenario events, then the scenario starts with an instance of the ***Contain*** metaclass. In the opposite case, when the scenario is only one method invocation, the ***root*** attribute contains that event.

9. **The *absent* attribute of the *Negative* metaclass can only be an instance of the *ScenarioEvent* and *Contain* metaclasses.**
   A negative sub scenario can only be a part of the optional block. An optional block is a composite block and can be treated either as a scenario event or a contain interrelation between the optional block and inside specified application logic.

8. Chapter
The Metamodel Formal Semantics

The metamodel defined in the previous chapter enables specification of measurements and metrics computation for particular events. It is in natural language and set theory explained what each metaclass is for. Many of current languages are defined in such a way. For this reason, many of them are misused [Selic, 2003]. This section gives the formal semantics of the metamodel for measurement and assessment. Formal semantics are given with mappings of modeling elements to Libkin's algebra. Libkin's algebra is a ***relational algebra*** with an ***open set of aggregate functions***. Furthermore, this formal semantics is used as the transformation design. The implementation of mappings is done according to the formal semantics. This chapter is structured as follows. Section 8.1 explains prerequisites of the algebra/RDBMS required for this mapping and initialization of the RDBMS. Next, Section 8.2 gives the formal semantics of the context part of the metamodel. The final section of this chapter, Section 8.3 gives formal semantics to assessment and metrics part of the metamodel.

8.1. The Relational Database Management System Prerequisites and Initialization

In order to be used for the metrics computation, as described in Chapter 5, the RDBMS has to satisfy some prerequisites and has to be initialized. Those prerequisites are specified in Subsection 8.1.1. The initialization is specified in Subsection 8.1.2.

8.1.1. Prerequisites

The ***MoDePeMART*** approach for performance measurement and assessment, as explained in Chapter 5, suggests the usage of RDBMS. In order to be used, the RDBMS has to satisfy two prerequisites.

The first prerequisite the RDBMS has to satisfy is that each statistical function specified in the enumeration ***Statistics*** of the metamodel has to have a corresponding aggregate function in the database management system. This set of statistical functions can vary from project to project. Therefore, for some specific project, some specific statistical functions can be added to this set. In the case that there is no corresponding aggregate function in the RDBMS, this prerequisite orders an implementation of that function. Most of the current relational database management systems support adding new aggregate functions.

The second prerequisite of the used RDBMS is that it must contain a counting aggregate function, independently of the functions in the ***Statistics*** set. This function is needed for density and cumulative histogram computations. This assumption is realistic because COUNT is one of the standard SQL functions [Atzeni et al., 1999].

Formally, the RDBMS is modeled with Libkin's algebra, explained in Chapter 4. The first prerequisite is defined as the existence of an injective function between the ***Statistics*** set and the set of aggregate functions in Libkin's algebra, Θ. The second prerequisite is trivial. Formally, this is defined as:

$\exists agg : Statistics \rightarrow \Theta, \forall s_1, s_2 \in Statistics((s_1 \neq s_2) \Leftrightarrow (agg(s_1) \neq agg(s_2)))$ and
$\exists c \in \Theta$, where c is aggregate function which corresponds to standard COUNT aggregate function.

The preconditions of algebra correspond to the requirements of the used database management system. The initialization is not the requirement of the RDBMS, but actually requirements of the database used for performance data storage and metrics computation. It consists of creation and filling the database tables needed for metrics computation, as explained in Chapter 5.

8.1.2. Initialization

A RDBMS satisfying prerequisites described in the previous section has to be initialized in order to facilitate storage and metrics computation. The initialization yields a place for storage of measured data. The relation for data storage corresponds to the formal specification of execution in Section 6.2, at the page 55. This initialization is formally specified in the following.

Let $mToDom : M \rightarrow Dom$ be a function which maps *elements of a model* to the the elements of the ***Dom*** set of Libkin's algebra. Let $threadToDom : Th \rightarrow Dom$ be a function which maps the set of threads/processes/sessions to the ***Dom*** set. Let ***Ran(mToDom)*** and ***Ran(threadToDom)*** be ranges of functions ***mToDom*** and ***threadToDom***, respectively. The relation in which the execution data of a software system is stored is defined as:

$$\exists executionTrace \in SC(executionTrace : bbnn) \wedge \\ (executionTrace \subseteq Ran(mToDom) \times Ran(threadToDom) \times Num \times Num)).$$

Each software system operation takes place in a time interval. This information is also needed for metrics computation. It is particularly needed when computing performance metrics for the time interval of whole operation. For example, let us assume that the throughput of a component for the whole experiment time is of interest. This metric is, then, measured and computed for the whole experiment time interval. The computation takes the complete number of invocations and divides them by the duration of the experiment time interval. Furthermore, this information can be used to remove events from the training period of software execution. The training period is the period of software system initialization. Invocations of the training period are not considered in metrics computation and assessment. For this reasons, the ***SC*** in Libkin's Algebra has to have defined a relation containing only start and end experiment time. Accordingly, used database has to have a table with only one record. This record consists of only the start and the end time of the experiment. Formally:

$$\exists experimentTime((experimentTime \in SC) \wedge (experimentTime : nn))$$

Finally, as mentioned in Section 7.2, interval sets are used for grouping data for some particular computation. In the database this is implemented by having a table for each interval set. These tables contain records with values corresponding to values in interval sets. Later, these tables are used for structuring information for computation and assessment.

This initialization is formally specified as an existence of mapping of the metamodel MM_{pema} set ***IntervalSet*** to the ***SC*** database schema of the Libkin's algebra. This mapping maps instances of metaclasses specialized from the metaclass ***IntervalSet*** to the corresponding tables in the ***SC*** database schema. Formal definition is in the following.

Let ***timeInstantToNum*** be a function which maps ***TimeInstant*** *set of the metamodel* MM_{pema} to the ***Num*** *set of the Libkin's algebra*, $timeInstantToNum : TimeInstant \rightarrow Num$. And let ***dToNum*** be a function which maps the domain ***double*** of the metamodel MM_{pema} to the domain ***Num*** of Libkin's algebra, $dToNum : double \rightarrow Num$. Then, this initialization is defined as:

$$\forall tis \in TimeIntervalSet \exists ttis \in SC((ttis : nn) \wedge (tis = (tinterval)) \wedge$$
$$(tinterval \subseteq TInterval) \wedge \forall tin \in tinterval \exists (tsti, teti) \in ttis((tin = (sti, eti)) \wedge$$
$$(sti, eti \in TimeInstant)$$
$$\wedge (tsti = timeInstantToNum(sti)) \wedge (teti = timeInstantToNum(eti))))$$

The previous part of the definition specifies the characteristics of tables in ***SC*** which correspond to time interval sets of the metamodel. The definition of characteristics of tables corresponding to real numbers interval sets is similar and follows.

$$\forall ris \in RealNumbersIntervalSet \exists tris \in SC((tris : nn) \wedge (ris = (rinterval)) \wedge$$
$$(rinterval \subseteq RealNumbersInterval) \wedge \forall rin \in rinterval \exists (tlb, tub) \in tris((rin = (lb, ub)) \wedge$$
$$(rb, lb \in double) \wedge (tlb = dToNum(ub)) \wedge (tub = dToNum(eti))))$$

The mapping ***rToNum*** of the metamodel ***double*** domain to the ***Num*** domain of Libkin's algebra is a mapping between different representations of the set $\mathbb{R}$, since ***Num*** represents the domain of real numbers in Libkin's algebra. Usage of ***Num*** domain for representation of time is possible because the discrete time model is ***isomorphic*** with the ***set of natural numbers*** $\mathbb{N}$, and the set of natural numbers is subset of $\mathbb{R}$. Furthermore, isomorphism is a transitive relation, and therefore, each subset of $\mathbb{R}$ which is isomorphic with set $\mathbb{N}$, is isomorphic with set ***TimeInstant***. That implies that ***TimeInstant*** can be mapped to the subset of $\mathbb{R}$ with finer granularity then $\mathbb{N}$, as long as this subset is isomorphic to $\mathbb{N}$.

The final part of this initialization definition is the specification of the the ***isRel*** mapping function, explained previously in the text:

$$\exists iSRel : IntervalSet \rightarrow SC((is \in IntervalSet) \wedge$$
$$((((is, tis) \in ISTIS) \wedge (ttis = iSRel(is))) \vee (((is, ris) \in ISRNIS) \wedge (tris = iSRel(is)))))$$

8.2. The Event and Context Metamodel Part Formal Semantics

The complete formal semantics of the metamodel presented in Chapter 7 defines a set of transformations to Libkin's algebra queries. These queries yield from the previously defined database schema ***SC***, relation having the desired performance assessment. This process has two steps. The first step is the selection of events specified in the event and context metamodel part. These events are, after the selection, subject of the metrics computation. The metrics computation is performed according to the assessment and metrics specification and it is specified in the next section.

Subject of this section is a set of mappings defining queries whose outcomes are relations representing executions of scenarios in specified contexts and isolating measured events out of them. The type of this relation is **bnn** and contains the event of interest and times of its start and end. This relation is the outcome of the function ***grTr*** explained at the end of Subsection 8.2.2. Such relation is a union of relations having executions of specified measured events. The sets of mappings for selecting measured events are specified in Subsections 8.2.1 and 8.2.2.

Figure 8.1 presents an example of abstract syntax tree used for the explanation of these mappings. It contains one use case of the sequence diagram in Figure 6.2, at the page 45. The use case it contains is obtaining a music video from the database when the compression is on.

The specification of obtaining a music video file is a group containing only one measured event. The scenario of the measured event is a sequence which occurs with an invocation of the ***getVideoItem*** operation of the ***VideoItemFacade*** class. This sequence contains only one invocation of the ***getItem*** method. It is an invocation for obtaining a music video. In order to distinct this use case from the use case when a video file and a trailer are obtained, it must be specified that there is no execution of the body of the optional block which precedes the ***getItem*** execution.

The abstract syntax tree presented in Figure 8.1 is for visualization. It is based on UML Diagrams used in Chapter 7 for the visualization of the metamodel. In order to understand the formal definition of mappings, it is needed to understand how the actual abstract syntax tree looks like. The actual abstract syntax tree is in correspondence with the metamodel formal specification in the form of sets. According to the formal specification, what is in Figure 8.1 presented as instances of ***ScenarioEvent***, ***Contain***, ***Precede***, ***Negative*** and ***ConditionEvent*** metaclasses actually consist of several related set elements. An instances of ***ScenarioEvent*** *contains elements of* the ***ScenarioEvent***, ***Root***, and ***SubScenario*** sets ***forming doubles*** in the ***SSSE*** and ***RSE*** metamodel relations. These related elements are results of inheritances from superclasses. The basic principle of defining inheritance is specified in Section 7.1,p. 58. Accordingly, an instance of ***Contain*** metaclass contain elements of ***Contain***, ***Root***, and ***SubScenario*** sets related with the ***RC*** and ***SSR*** metamodel relations. Furthermore, instances of associations like ***StoR***, ***CtoSS*** for example, relate elements of ***Scenario*** and ***Root***, and ***Contain*** and ***SubScenario***, respectively, because they are defined as relations between these two metaclasses. Instances of ***Precede***, and ***Negative*** have elements of ***Precede*** and ***Negative*** sets, and additionally elements of only ***SubScenario***. The instance of ***ConditionEvent*** consists of elements of the ***ConditionEvent***

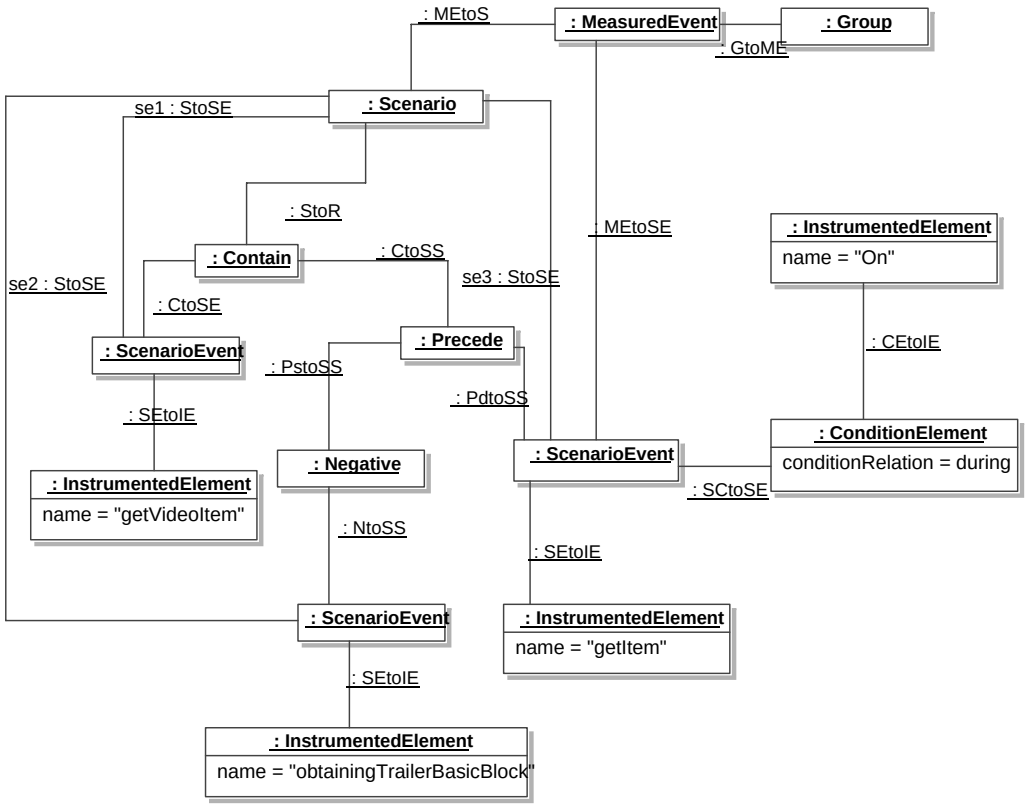


Figure 8.1.: An abstract syntax tree of the event and context specification for obtaining a music video in Figure 6.2

and ***StateCondition*** set.

8.2.1. The Reactive Context Metamodel Formal Semantics

The mappings of the reactive context metamodel to Libkin's algebra yield for each specified scenario event a relation containing executions of its instrumented element which satisfies the state condition. In the example in Figure 8.1 these relations are made for all three scenario events. They are made by the ***evTrace*** mapping from the ***executionTrace***, specified in subsection 8.1.2. The ***evTrace*** function takes as argument $se \in ScenarioEvent$ and produces the desired relation in the following way.

$$evTrace(se) = \begin{cases} vEvent(ie), \\ \quad \text{if } \exists(sev_1, ie) \in SEtoIE \not\exists (sev_2, sc) \in SEtoSC((sev_1 = se) \wedge \\ \quad (sev_2 = se)) \\ \\ cEvTr(se, sc), \\ \quad \text{if } \exists(sev_1, ie) \in SEtoIE, (sev_2, sc) \in SEtoSC((sev_1 = se) \wedge (sev_2 = se)) \end{cases}$$

The function recognizes two cases. The upper case is the case when there is no reactive

context specified. Here in the resulting relation are all executions of the instrumented element in the specified experiment time window. Such scenario events in the Figure 8.1 example are scenario events with ***getVideoItem*** and ***obtainingTrailerBasicBlock*** as instrumented elements. For them the computation is achieved with the function ***vEvTr*** which takes as argument $ie \in InstrumentedElement$ and is defined as follows.

$$vEvent(ie) = \pi_{1,2,3,4}(\sigma[>]_{3,5}(\sigma[<]_{3,6}(\sigma_{1=mToDom(ie)}(executionTrace) \times experimentTime))).$$

The ***vEvent*** function first makes a Cartesian product of the relation in which are executions of a specified modeling element, and the ***experimentTime*** relation. After, only those executions of instrumented element are selected which execute in the time experiment time interval. This is accomplished with two selection operations. The first selection selects only those modeling element occurrences which start after the beginning. The second selects those ending before the end of the experiment time. Finally, with the projection, values of experiment time interval are removed from the relation.

The bottom case of the function ***evTrace*** is when a reactive context for the scenario event is specified. The reactive context is specified with the Boolean algebra depicted in Figure 7.5, p. 73. In Figure 8.1 example, this is the case of scenario event with the ***getItem*** instrumented element. The function ***cEvTr*** yields relations of scenario events with reactive context. The ***cEvTr*** mapping takes as arguments $se \in ScenarioEvent$ and $sc \in ScenarioCondition$, and it is specified as:

$$cEvTr(se, sc) = \begin{cases} cEvTr(se, lo) \cap cEvTr(se, ro), \\ \quad \text{if } \exists b \in Binary, a \in AND, lo, ro \in ScenarioCondition(((sc, b) \in SCB) \\ \quad \wedge((b, a) \in AND) \wedge ((b, lo) \in BtoSClo) \wedge ((b, ro) \in BtoSCro)) \\ \\ cEvTr(se, lo) \cup cEvTr(se, ro), \\ \quad \text{if } \exists b \in Binary, a \in AND, lo, ro \in ScenarioCondition(((sc, b) \in SCB) \\ \quad \wedge((b, a) \in AND) \wedge ((b, lo) \in BtoSClo) \wedge ((b, ro) \in BtoSCro)) \\ \\ \pi_{1,2,3,4}(\sigma[=]_{2,5}(vEvent(ie) \times cEvTr(se, nsc))) \\ -cEvTr(se, nsc) \\ \quad \text{if } \exists n \in NOT, nsc \in ScenarioCondition, ie \in InstrumentedElement \\ \quad (((sc, n) \in SCN) \wedge ((n, nsc) \in NtoSC) \wedge ((se, ie) \in SEtoIE)) \\ \\ oneCondEvTr(se, ce, ie), \\ \quad \text{if } \exists ce \in ConditionElement(((sc, ce) \in SCCE) \wedge ((ce, ie) \in CEtoIE)) \end{cases}$$

The simplest case of this function is the bottom case. This is the case when the boolean expression consists of only one condition element. An example of this case in Figure 8.1 is the scenario event with the ***getItem*** instrumented element. Here, the function ***cEvTr*** only uses the function ***oneCondEvTr***, specified in the following. Let $se \in ScenarioEvent$,$ce \in$

$ConditionElement$, and $ie \in InstrumentedElement$. The ***oneCondEvTr*** function is then:

$$oneCondEvTr(se, ce, ie) = \begin{cases} \pi_{1,2,3,4}(\sigma[<]_{3,7}(\sigma[>]_{4,8}(\sigma[=]_{2,6}(vEvent(ie_{se}) \times vEvent(ie))))), \\ \quad \text{if } ce = (cr) \wedge (cr = contains) \wedge ((se_{se}, ie_{se}) \in SEtoIE) \wedge \\ \quad (se_{se} = se) \\ \\ \pi_{1,2,3,4}(\sigma[>]_{3,7}(\sigma[<]_{4,8}(\sigma[=]_{2,6}(vEvent(ie_{se}) \times vEvent(ie))))), \\ \quad \text{if } ce = (cr) \wedge (cr = during) \wedge ((se_{se}, ie_{se}) \in SEtoIE) \wedge \\ \quad (se_{se} = se) \\ \\ \pi_{1,2,3,4}(\sigma[<]_{3,7}(\sigma[<]_{4,8}(\sigma[=]_{2,6}(vEvent(ie_{se}) \times vEvent(ie))))), \\ \quad \text{if } ce = (cr) \wedge (cr = overlaps) \wedge ((se_{se}, ie_{se}) \in SEtoIE) \wedge \\ \quad (se_{se} = se) \\ \\ \pi_{1,2,3,4}(\sigma[>]_{3,7}(\sigma[>]_{4,8}(\sigma[=]_{2,6}(vEvent(ie_{se}) \times vEvent(ie))))), \\ \quad \text{if } ce = (cr) \wedge (cr = overlapped) \wedge ((se_{se}, ie_{se}) \in SEtoIE) \wedge \\ \quad (se_{se} = se) \end{cases}$$

The function ***oneCondEvTr*** produces the relation having only those executions of the specified event which are in the defined ***ConditionRelation*** relation with the specified state. For the case of scenario event with ***getItem*** as instrumented element this function is depicted in Figure 8.2. It does this in the similar way as the function ***vEvent***. First is performed a Cartesian product of denoted state experiment time and specified modeling element execution relations. Subsequently, a selection is performed to isolate only executions in which the state and the modeling element execute in the same thread of control. This is in correspondence with the assumptions of the execution of the system, defined in Section 6.1, p. 55. Thereafter, depending on the specified interrelation between these two modeling elements, appropriate comparison of their execution time intervals are made and appropriate selection carried out. At the end is performed a projection on the modeling element, thread and modeling element execution time interval.

The other three cases of the function ***cEvTr*** are the cases when the reactive context is specified with more than one condition element. Such specifications are more complex Figure 7.5 Boolean algebra specifications. Here is for each condition element computed a relation with the function ***oneCondEvTr***, and the Figure 7.5 Boolean algebra expressions are simply mapped to the corresponding Libkin's algebra boolean expressions. Boolean operations ***OR*** and ***AND*** of that algebra map to the set operations $\cup$, and $\cap$, respectively. Accordingly, the function is defined with recursive invocations and corresponding boolean operation of results. Negation, ***NOT*** expression mapping is defined in two steps. First is computed the set of all element executions satisfying the negated state. Then is calculated the complement of this set in the set of all executions of that modeling element. The final relation contains all executions not satisfying the specified condition.

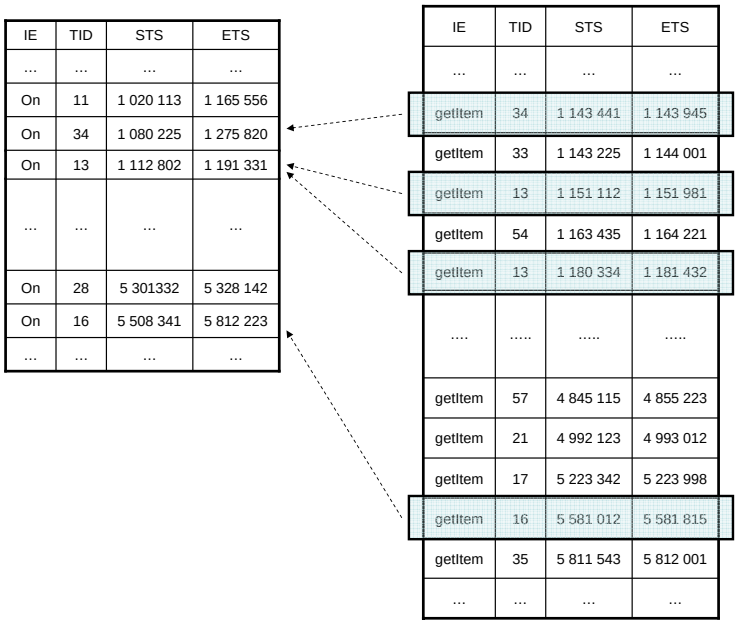


Figure 8.2.: The result of the ***oneCondEvTr*** mapping of the scenario event with instrumented element ***getItem***. The resulting relation contains marked rows of the left relation which are executions of ***getItem*** while the compression was on.

8.2.2. The Transformational Context Formal Semantics

The previously explained reactive context mappings produce relations containing scenario events satisfying the specified reactive context. These relations are used by transformational context mappings in order to produce relations representing specified scenarios, and selecting the measured events out of them.

The function whose outcome is the relation containing executions of the measured event is the function ***msEvTrace***. The outcome of this function is a relation containing three columns, instrumented element identifier, start time of the execution and end time of the execution. The executions of that instrumented element are in the specified transformational and reactive context. This function is defined in the following.

Let $me \in MeasuredEvent$. And let ***scOrdNum*** be an ordinal function mapping a particular scenario and a particular scenario event to the set $\mathbb{N}$, $scOrdNum : Scenario \times ScenarioEvent \rightarrow \mathbb{N}$, defined in the Appendix C. Let ***scTrace*** be a function which computes a relation containing executions of the specified scenario. The result of ***scTrace*** is a subset of the relation illustrated in Figure 8.3, at the page 87. The function ***scTrace*** is defined

later in this section. Then ***msEvTrace*** function is:

$$msEvTrace(me) = \begin{cases} \pi_{1,3,4}(evTrace(se)) \\ \quad \text{if } \exists se \in ScenarioEvent, sc_{me} \in Scenario, scEv \subseteq StoSE \\ \quad \forall (sc, se) \in scEv(((me, sc_{me}) \in MEtoS) \wedge (sc = sc_{me}) \wedge \\ \quad ((me, se_{me}) \in MEtoSE) \wedge (card(scEv) = 1)) \\ \\ \pi_{3*scOrdNum(me,sc)-2,3*scOrdNum(me,sc)-1,3*scOrdNum(me,sc)}(scTrace(sc, r)) \\ \quad \text{if } \exists se \in ScenarioEvent, sc_{me} \in Scenario, scEv \subseteq StoSE \\ \quad \exists r \in Root \forall (sc, se) \in scEv(((me, sc_{me}) \in MEtoS) \wedge \\ \quad ((me, se_{me}) \in MEtoSE) \wedge (sc = sc_{me})) \end{cases}$$

The function ***msEvTrace*** recognizes two cases. The upper case is when the measured event is the event of a scenario consisting of a single event. In this case, the resulting relation contains event identifier, and its start and end times taken from the result of the ***evTrace*** function, defined in the previous subsection.

The second case is the case when the measured event is a part of some defined scenario. In this case the result is obtained by execution of a projection on columns of the ***scTrace*** resulting relation. The ***scTrace*** takes as arguments a scenario and its root. The resulting relation consists of tuples representing a certain occurrence of the specified scenario in software execution. In that relation, each scenario event is represented with three columns. These three columns contain instrumented element identification, and its start and end time of execution. The projection is performed on tuples representing the measured event of that scenario. The ***scTrace*** function and mappings it uses for producing the resulting relation are given in the rest of this subsection.

The starting mapping for producing ***scTrace*** resulting relation is the function ***scRelation***. Formally, this mapping is defined as follows.

Let $sc \in Scenario$. Let, ***n*** be the number of the ***ScenarioEvent*** metaclass instances related in the ***StoSE*** relation to ***sc***. And let ***ordNumSEv*** be a function which maps a scenario and an ordinal number of a scenario event to that scenario event, $odrNumSEv : Scenario \times \mathbb{N} \rightarrow ScenarioEvent$, defined in Appendix C. Then:

$$\begin{aligned} scRelation(sc) = \pi_{1,3,4,5,7,8,\ldots,n*4-3,n*4-1,n*4}(\\ \sigma[=]_{2,6}(\sigma[=]_{6,10}(\ldots(\sigma[=]_{n*4-2,n*4+1}(evTrace(ordNumSEv(sc,1)) \times \\ evTrace(ordNumSEv(sc,2)) \times \ldots \times evTrace(ordNumSEv(sc,n))))))). \end{aligned}$$

The resulting relation of ***scRelation*** is created with a Cartesian product of results of ***evTrace*** for each ***ScenarioEvent*** of the specified scenario. From this Cartesian product are selected only those executions of scenario events' instrumented element executing in the same thread/process/session. Because the thread/process/session identifier is the same in the tuple, columns denotating them are removed from the resulting relation with a projection. Finally, in the resulting relation is each tuple a potential scenario sequence, where each scenario event is specified with instrumented element identifier, start, and end time stamp of its execution. The resulting relation of the ***scRelation*** function for the Figure 8.1 example is given in Figure 8.3.

IE1	STS1	ETS1	IE2	STS2	ETS2	IE3	STS3	ETS3
...	...	...	...	...	...	...	...	...
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 143 441	1 143 945
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 151 112	1 151 981
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 180 334	1 181 432
...	...	...	...	...	...	...	...	...
getVideoItem	5 580 882	5 582 013	obtainingTrailerBasicBlock	1 179 112	1 180 123	getItem	5 581 012	5 581 815
getVideoItem	5 580 882	5 582 013	obtainingTrailerBasicBlock	1 229 109	1 229 803	getItem	1 143 441	1 143 945
...	...	...	...	...	...	...	...	...

Figure 8.3.: Example of the result of ***scRelation*** for the Figure 8.1 example.

The result of the function ***scRelation*** is used for defining relation which represent executions of the specified scenario. Selection leaving in final relations only those tuples of the ***scRelation*** result in which executions of scenario events satisfy correlations defined in scenario is specified in the function ***scTrace***. This function uses several sub functions and in the following are defined these sub functions.

One of the functions used for resolving the transformational context is ***fEv***. This function defines the first starting event in one sub scenario $ss \in SubScenario$. It is defined as:

$$fEv(ss) = \begin{cases} ss_{se}, \\ \quad \text{if } \exists c \in Contain, se \in ScenarioEvent, ss_{se} \in SubScenario \\ \quad (((ss, c) \in SSC) \wedge ((c, se) \in CtoSE) \wedge ((ss_{se}, se) \in SSSE)) \\ \\ fEv(ss_{ps}), \\ \quad \text{if } \exists p \in Precede, ss_{ps} \in SubScenario \nexists n \in Negative \\ \quad (((ss, p) \in SSP) \wedge ((p, ss_{ps}) \in PstoSS) \wedge ((ss_{ps}, n) \in SSN)) \\ \\ fEv(ss_{pd}), \\ \quad \text{if } \exists p \in Precede, ss_{ps}, ss_{pd} \in SubScenario, n \in Negative(((ss, p) \in SSP) \wedge \\ \quad ((p, ss_{ps}) \in PstoSS) \wedge ((ss_{ps}, n) \in SSN) \wedge ((p, ss_{pd}) \in PdtoSS)) \\ \\ ss, \\ \quad \text{if } \exists se \in ScenarioEvent), a \in Alternatives(((sc, se) \in SSSE) \vee \\ \quad ((sc, a) \in SSA)) \end{cases}$$

The ***fEv*** function is a recursive function which recognizes, four different cases:

1. ***ss*** is a ***Contain*** sub scenario. In this case the result is the event which contains the execution of contained scenario. In the example in Figure 8.1 the scenario event having ***getVideoItem*** as instrumented element is the first event of the whole scenario.

2. ***ss*** is a ***Preceede*** subscenario and the preceding sub scenario is not an instance of ***Negation***. Here, the first event is the first event of the preceding scenario.

3. ***ss*** is a ***Preceede*** sub scenario and the preceding sub scenario is a ***Negation*** sub scenario. In this situation, the preceding sub scenario does not exist. Therefore, the result is the first event of the preceded sub scenario. Such case can be seen in Figure 8.1. Here the first event is the scenario event having ***getItem*** as the instrumented element.

4. ***ss*** is either a ***ScenarioEvent*** or ***Alternatives*** subscenario. Now, the ***ss*** value is the result. This information is later used in ***scTrace*** function for complex selections which an alternating sub scenario requires.

Besides the ***fEv*** function there is also a need for a function resulting in an last ending event, entitled ***lEv***. Let $ss \in SubScenario$. This function is defined as:

$$lEv(ss) = \begin{cases} ss_{se}, \\ \quad \text{if } \exists c \in Contain, se \in ScenarioEvent, ss_{se} \in ScenarioEvent \\ \quad (((ss, c) \in SSC) \wedge ((c, se) \in CtoSE) \wedge ((ss_{se}, se) \in SSSE)) \\ \\ lEv(ss_{pd}), \\ \quad \text{if } \exists p \in Precede, ss_{pd} \in SubScenario \nexists n \in Negative \\ \quad (((ss, p) \in SSP) \wedge ((p, ss_{pd}) \in PdtoSS) \wedge ((ss_{pd}, n) \in SSN)) \\ \\ lEv(ss_{ps}), \\ \quad \text{if } \exists p \in Precede, ss_{ps}, ss_{pd} \in SubScenario, n \in Negative(((ss, p) \in SSP) \wedge \\ \quad ((p, ss_{ps}) \in PdtoSS) \wedge ((sd_{ps}, n) \in SSN) \wedge ((p, ss_{ps}) \in PstoSS)) \\ \\ ss, \\ \quad \text{if } \exists se \in ScenarioEvent), a \in Alternatives(((sc, se) \in SSSE) \vee \\ \quad ((sc, a) \in SSA)) \end{cases}$$

Similarly to the ***fEv***, function ***lEv*** recognizes four cases:

1. ***ss*** is a ***Contain*** sub scenario. Here, the last ending event of scenario is also the event which contains the sub scenario. In the Figure 8.1 the last ending event for the ***Contain*** metaclass instance is the ***getVideoItem*** operation.

2. ***ss*** is a ***Precede*** sub scenario and preceded sub scenario is not a ***Negation***. Now, the result is the last event of the preceded sub scenario.

3. ***ss*** is a ***Precede*** sub scenario and preceded sub scenario is a ***Negation*** sub scenario. In this case, the result is the last event of the preceding sub scenario, because there is no preceded sub scenario. In the Figure 8.1 the outcome of this function, when visiting the instance of Precede is the sub scenario having ***getItem*** as the instrumented element.

4. ***ss*** is a ***ScenarioEvent*** sub scenario or ***Alternatives*** sub scenario. Similarly and for the same reasons as in ***fEv*** function, here is ***ss*** value also the result.

For the definition of ***scTrace*** let $sc \in Scenario$, and $ss \in SubScenario$. Let $altSubSc : Alternatives \times \mathbb{N} \rightarrow ScenarioEvent$ and already mentioned ***scOrdNum*** be functions defined in Appendix C. And let ***contSel***, and ***prSel*** be later defined functions selecting tuples from the function ***scRelation*** resulting relation which satisfy contain and precede interrelations for sub scenarios, respectively. Then:

$$
scTrace(sc, ss) = \begin{cases}
contSel(sc, se_{cs}, ss_{cd}) \\
\quad \text{if } \exists c \in Contain, se_{cs} \in ScenarioEvent, ss_{sc} \in SubScenario \\
\quad (((ss, c) \in SSC) \wedge ((c, se_{cs}) \in ScenarioEvent) \wedge ((c, ss_{cs}) \in CtoSS)) \\
\\
prSel(sc, ss) \\
\quad \text{if } \exists p \in Precede((ss, p) \in SSP) \\
\\
scTrace(sc, altSubSc(a, 1)) \cap ... \cap scTrace(sc, altSubSc(a, n)) \\
\quad \text{if } \exists a \in Alternatives, alt \subseteq AtoR \forall (a_{alt}, r) \in alt \\
\quad (((ss, a) \in SSA) \wedge (a_{alt} = a) \wedge (n = card(alt))) \\
\\
\pi_{1,2,..,n}(\sigma[=]_{3*scOrdNum(sc,cNeg(ss))-1,3*n+1} \\
(\sigma[=]_{3*scOrdNum(sc,cNeg(ss))-1,n+2}(scRelation(sc) \times (evTrace(cNeg(ss))- \\
\pi_{3*scOrdNum(sc,cNeg(ss))-2,3*scOrdNum(sc,cNeg(ss))-1,3*scOrdNum(cNeg(ss))} \\
(contSel(sc, c_{se}, ss_{ab})))))) \\
\quad \text{if } \exists n \in Negative, ss_{ab} \in SubScenario, c \in Contain, se \in ScenarioEvent \\
\quad (((ss, n) \in SSN) \wedge ((n, ss_{ab}) \in NtoSS) \wedge ((cNeg(ss), c) \in SSC) \wedge \\
\quad ((c, c_{se}) \in CtoSE)) \\
\\
scRelation(sc) \\
\quad \text{if } \exists se \in ScenarioEvent((ss, se) \in SSSE)
\end{cases}
$$

From the definition of this function can be seen that it is a recursive function used for traversing the abstract syntax three. Nodes in syntax tree are sub scenarios. When invoked for a certain node, the function ***scTrace*** also returns a subset of the ***scRelation*** resulting relation. In such a case the ***scTrace*** relation returns a relation in which values in columns corresponding to scenario events of that sub scenario satisfy correlations defined by that sub scenario.

The final result of the ***scTrace*** for a certain scenario is the intersection of ***scTrace*** for all sub scenarios. This means that in the result are tuples in which values of components corresponding to scenario events satisfy all specified interrelations. The selections are explained in the following.

When the visited node of model syntax tree is an instance of ***Contains*** metaclass, the result of ***scTrace*** function is the result of function ***contSel***. Function ***contSel*** maps the sub scenario containing event, and the contained sub scenario to appropriate tuples in ***scRelation*** of the modeled scenario. This funcion is specified in the following.

Let $sc \in Scenario$, $cts \in ScenarioEvent$, and $ctd \in SubScenario$. And let ***sAf*** and ***eBe*** be functions which take a scenario, an event and a sub scenario of that scenario as arguments, and select tuples from the scenario's ***scRelation*** outcome, in which the sub scenario starts after and ends before the specified event, respectively. And let ***scOrdNum*** be a function taking a scenario and one of its scenario events and maps them to that numbers ordinal number in the scenario, $scOrdNum : Scenario \times ScenarioEvent \rightarrow \mathbb{N}$, defined in Appendix C. Then ***contSel*** function is defined as:

$$contSel(sc, cs, cd) = \begin{cases} ((sA(sc, cs, cd)) \cap (eBe(sc, cs, cd))) \\ \quad \text{if } \exists se_{cd} \in ScenarioEvent((cd, se_{cd}) \in SSSE) \\ \\ scTrace(sc, cd) \\ \quad \text{if } \exists n_{cd} \in Negation((cd, n_{cd}) \in SSN) \\ \\ ((sAf(sc, cs, cd)) \cap (eBe(sc, cs, cd))) \cap scTrace(sc, cd) \\ \quad \exists c_{cd} \in Contain, p_{cd} \in Precede, \nexists a_{fe} \in Alternatives \\ \quad (((cd, c_{cd}) \in SSC) \vee (((cd, p_{cd}) \in SSP) \wedge \\ \quad (fEv(cd) = lEv(cd) = a_{fe}))) \\ \\ ((sAf(sc, cs, altSubSc(a_f, 1)) \cap eBe(sc, cs, altSubSc(a_f, 1))) \cup .. \\ .. \cup (sAf(sc, cs, altSubSc(a_f, n))) \cap eBe(sc, cs, altSubSc(a_f, n))) \\ \cap scTrace(sc, cd) \\ \quad \text{if } \exists a_f \in Alternatives, alt \subseteq AtoR \forall (a, ass) \in alt \\ \quad (((fE(cd), a_f) \in SSA) \wedge (a = a_{cd}) \wedge (fE(cd) = lE(cd)) \\ \quad \wedge (n = card(alt))) \end{cases}$$

Function ***contSel*** recognizes four cases. The first case is when the contained sub scenario is only one event, and that is an execution of a modeling element in a defined state. Here, the resulting relation is intersection of two relations. In one relation the execution of this modeling elements stats after the event which contains sub scenario. In the other the execution of the modeling element ends before the event which contains the sub scenario. The second case is the case when the contained sub scenario is the negation. In this case, the result of this function is the invocation of ***scTrace*** for that negation, because there is no other interrelation to be satisfied.

The third case of this function is, when the sub scenario is either a contain or a precede interrelation whose first event is not an alternative. Then, a further selection is needed besides selecting relations in which sub scenario starts after and ends before the execution of the sub scenario containing event. The selection is done with continuing selection with

traversing the model syntax tree. This case can be found in Figure 8.1 example, and it is used for depicting the transformation.

Previously described functions***sAf*** and ***eBe*** are defined in the following.

Let $sc \in Scenario$, $cts \in ScenarioEvent$, and $ctd \in SubScenario$. And let ***plpfSel*** be a function which takes two sub scenarios and selects only those relations in which one sub scenario precedes another one. And let ***fEvAltP*** be a function which takes as arguments an alteration and a sub scenario and returns the first sub sub scenario of that sub scenario it precedes. The ***plpfSel*** and ***fEvAltP*** functions are described later in the section.Then, the ***sAf*** function is defined as:

$$sAf(sc,cs,cd) = \begin{cases} \sigma[<]_{3*scOrdNum(sc,cs)-1,3*scOrdNum(sc,se_{cd})-1}(scRelation(sc)) \\ \quad \text{if } \exists se_{cd} \in ScenarioEvent((fEv(cd),se_{cd}) \in SSSE) \\ \\ ((sAf(sc,cs,altSubSc(a_c,1)) \cap (plpfSel(sc,altSubSc(a_c,1)),ss_{ap})) \cup .. \\ \cup((sAf(sc,cs,altSubSc(a_c,n)) \cap (plpfSel(sc,altSubSc(a_c,n)),ss_{ap})))) \\ \quad \text{if } \exists a_c \in Alternatives, alt \subseteq AtoR \forall (a,ass) \in alt \\ \quad (((fEv(cd),a_{cd}) \in SSA) \wedge (a = a_c) \wedge (n = card(alt)) \\ \quad \wedge (ss_{ap} = fAltEvP(a_c,cd))) \end{cases}$$

The function ***sAf*** recognizes two cases. In the following, the attention is payed only to the upper case, while the explanation of the bottom case is later in this section.

The upper case of the function ***sAf*** first finds the columns in relation corresponding to the first event of the contained sub scenario. Then, selected are only those having the start time is larger than the start time of the containing event. This is illustrated with an example in Figure 8.4.

IE1	STS1	ETS1	IE2	STS2	ETS2	IE3	STS3	ETS3
...	...	...	...	...	...	...	...	...
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 143 441	1 143 945
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 151 112	1 151 981
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 180 334	1 181 432
...	...	...	...	...	...	...	...	...
getVideoItem	5 580 882	5 582 013	obtainingTrailerBasicBlock	1 179 112	1 180 123	getItem	5 581 012	5 581 815
getVideoItem	5 580 882	5 582 013	obtainingTrailerBasicBlock	1 229 109	1 229 803	getItem	1 143 441	1 143 945
...	...	...	...	...	...	...	...	...

Figure 8.4.: Example of the result of the ***sAf*** mapping. The illustrated selection is for the instance of ***Contain*** in Figure 8.1. In all selected tuples is STS1>STS3.

The containing event in the ***Contain*** metaclass instance is the scenario event with ***getVideoItem*** as the instrumented element. According to the ***sAf*** function definition, the resulting tuples of this function have to have smaller start time of the executions of ***getVideoItem*** operation then the first event of the contained sub scenario. The first event is obtained with the function ***fEv***, and that is the ***getItem*** operation execution. Finally, the

resulting relation selects only those tuples in which ***getVideoItem*** execution starts before the ***getItem*** execution.

The previously mentioned function ***eBe***, takes a scenario, its containing event and the contained sub scenario as arguments and selects tuples in which the last event of contained sub scenario ends before the end of the containing event. It is similar to the ***sAf*** function. Let $sc \in Scenario$, $cts \in ScenarioEvent$, and $ctd \in SubScenario$. And let ***plpfSel*** be a function which takes two sub scenarios and selects tuples in which one sub scenario precedes another one. And let ***lEvAltP*** be a function which takes as arguments an alteration and a sub scenario and returns the first preceding sub sub scenario of that sub scenario. The ***plpfSel*** and ***lEvAltP*** functions are defined later in this section. Then, ***eBe*** is:

$$eBe(sc, cs, cd) = \begin{cases} \sigma[>]_{3*scOrdNum(sc,cs),3*scOrdNum(sc,se_{cd})}(scRelation(sc)) \\ \quad \text{if } \exists se_{cd} \in ScenarioEvent((lEv(cd), se_{cd}) \in SSSE) \\ \\ ((eBe(sc, altSubSc(a_c, 1), cs)) \cap (plpfSel(sc, ss_{ap}, altSubSc(a_c, 1)))) \cup .. \\ \cup((eBe(sc, altSubSc(a_c, n), cs) \cap (plpfSel(sc, ss_{ap}, altSubSc(a_c, n)))))) \\ \quad \text{if } \exists a_c \in Alternatives, alt \subseteq AtoR \forall (a, ass) \in alt \\ \quad (((lEv(cd), a_{cd}) \in SSA) \wedge (a = a_c) \wedge (n = card(alt)) \\ \quad \wedge (ss_{ap} = lAltEvP(a_c, cd))) \end{cases}$$

Here is also only the upper case discussed, while the bottom is discussed later in this section. The upper case is illustrated in Figure 8.5.

IE1	STS1	ETS1	IE2	STS2	ETS2	IE3	STS3	ETS3
...	...	...	...	...	...	...	...	...
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 143 441	1 143 945
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 151 112	1 151 981
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 180 334	1 181 432
...	...	...	...	...	...	...	...	...
getVideoItem	5 580 882	5 582 013	obtainingTrailerBasicBlock	1 179 112	1 180 123	getItem	5 581 012	5 581 815
getVideoItem	5 580 882	5 582 013	obtainingTrailerBasicBlock	1 229 109	1 229 803	getItem	1 143 441	1 143 945
...	...	...	...	...	...	...	...	...

Figure 8.5.: Example of the result of the ***eBe*** mapping. The illustrated selection is for the ***Contain*** instance in Figure 8.1. In the selected tuples is ETS1 > ETS3.

The upper case of the function ***eBe*** first finds the columns in the relation corresponding to the last event of the contained sub scenario. Then, only those are selected in which end time is smaller than the end time of the containing event.

Having previous definitions in mind, the intersection of results of ***sAf*** and ***eBe*** when visiting the instance of ***Contain*** metaclass in Figure 8.1 gives the relation satisfying both conditions. Such intersection result for the examples in Figure 8.4 and Figure 8.5 are rows marked in both figures. In order to have a relation which satisfies both, the previously mentioned contain interrelation, and the interrelations specified in the contained sub scenario,

the result is further intersected with the result of ***scTrace*** for the contained sub scenario. In this case the ***scTrace*** invokes the ***prSel*** mapping with the scenario, preceded, and preceding sub scenario as the arguments.

The definition of ***prSel*** is given in the following. Let ***plpfSel*** be a function selecting tuples in which the last event of the preceding sub scenario ends before the first event of preceded, defined later in this section. For the brevity in the definition, in each case is $sc \in Scenario, p \in Precede, ss, ss_{ps}, ss_{pd} \in SubScenario, se_{ps}, se_{pd} \in ScenarioEvent, n, n_{ps}, n_{pd} \in Negative$.

$$
prSel(sc, ss) = \begin{cases}
(plpfSel(sc, ss_{ps}, ss_{pd})) \\
\quad \text{if } \exists p, ss_{ps}, ss_{pd}, se_{ps}, se_{pd}(((ss, p) \in SSP) \wedge ((p, ss_{ps}) \in PstoSS) \wedge \\
\quad ((p, ss_{pd}) \in PdtoSS) \wedge ((ss_{ps}, se_{ps}) \in SSSE) \wedge ((ss_{pd}, se_{pd}) \in SSSE)) \\
\\
(plpfSel(sc, ss_{ps}, ss_{pd})) \cap scTrace(sc, ss_{ps}) \\
\quad \text{if } \exists p, ss_{ps}, ss_{pd}, se_{pd} \nexists se_{ps}, n(((ss, p) \in SSP) \wedge ((p, ss_{ps}) \in PstoSS) \wedge \\
\quad ((p, ss_{pd}) \in PdtoSS) \wedge ((ss_{ps}, se_{ps}) \in SSSE) \wedge \\
\quad ((ss_{pd}, se_{pd}) \in SSSE) \wedge ((ss_{ps}, n) \in SSN)) \\
\\
(plpfSel(sc, ss_{ps}, ss_{pd})) \cap scTrace(sc, ss_{pd}) \\
\quad \text{if } \exists p, ss_{ps}, ss_{pd}, se_{ps} \nexists se_{pd}(((ss, p) \in SSP) \wedge ((p, ss_{ps}) \in PstoSS) \wedge \\
\quad ((p, ss_{pd}) \in PdtoSS) \wedge ((ss_{ps}, se_{ps}) \in SSSE) \wedge \\
\quad ((ss_{pd}, se_{pd}) \in SSSE) \wedge ((ss_{pd}, n) \in SSN)) \\
\\
(plpfSel(sc, ps, pd)) \cap scTrace(sc, ps) \cap scTrace(sc, pd) \\
\quad \text{if } \exists p, ss_{ps}, ss_{pd} \nexists se_{pd}, se_{ps}, n_{ps}, n_{pd}(((ss, p) \in SSP) \wedge \\
\quad ((p, ss_{ps}) \in PstoSS) \wedge ((p, ss_{pd}) \in PdtoSS) \wedge ((ss_{ps}, se_{ps}) \in SSSE) \wedge \\
\quad ((ss_{pd}, se_{pd}) \in SSSE) \wedge ((ss_{ps}, n_{ps}) \in SSN) \wedge ((ss_{ps}, n_{pd}) \in SSN)) \\
\\
scTrace(sc, ps) \cap scTrace(sc, pd) \\
\quad \text{if } \exists p, ss_{ps}, ss_{pd}, n_{ps}, n_{pd}(((ss, p) \in SSP) \wedge ((p, ss_{ps}) \in PstoSS) \wedge \\
\quad ((p, ss_{pd}) \in PdtoSS) \wedge (((ss_{ps}, n_{ps}) \in SSN) \vee ((ss_{pd}, n_{pd}) \in SSN)))
\end{cases}
$$

Function ***prSel*** takes a scenario and a precede sub scenario of that scenario as arguments. The outcome of this function is the subset of ***scRelation*** result for that scenario in which preceded sub scenario executed after the preceding sub scenario. Depending on the kinds of sub scenarios it continues the traversal of sub scenarios. In the following is of interest the bottom case, because this is a case from the previous example. The remaining cases of this function are described later in this section.

The fifth case from the top of the function ***prSel*** is the case when one of the sub sub scenarios is an instance of ***Negation*** metaclass. Here exist no interrelations between the

preceded and the preceding sub sub scenario because one of them or even both do not execute. Therefore, the function only intersects relations satisfying the sub scenarios interrelations. For the Figure 8.1 example this function it produces the intersection of ***scTrace*** results for a ***Negation*** sub scenario and a ***ScenarioEvent*** sub scenario.

The invocation of the ***scTrace*** for an instance of a ***ScenarioEvent*** is the bottom case of the ***scTrace***. This case produces neutral result because there is no specified scenario event interrelation. Since the main operation in composing relation satisfying interrelations between events of different sub scenarios is intersection, the produced result is the result of ***scRelation*** for that scenario. The result of ***scRelation*** is neutral, because its intersection with some of its subset is that subset.

The resulting relation of the ***scTrace*** in the case of a ***Negative*** sub scenario contains those tuples in which the ***getVideoItem*** execution does not contain the negative sub scenario. For the Figure 8.1 example is this case illustrated in Figure 8.6.

IE1	STS1	ETS1	IE2	STS2	ETS2	IE3	STS3	ETS3
...	...	...	...	...	...	...	...	...
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 143 212	1 143 328	getItem	1 143 441	1 143 945
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 143 441	1 143 945
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 151 112	1 151 981
getVideoItem	1 143 200	1 144 141	obtainingTrailerBasicBlock	1 149 812	1 151 103	getItem	1 180 334	1 181 432
...	...	...	...	...	...	...	...	...
getVideoItem	5 580 882	5 582 013	obtainingTrailerBasicBlock	1 179 112	1 180 123	getItem	5 581 012	5 581 815
getVideoItem	5 580 882	5 582 013	obtainingTrailerBasicBlock	1 229 109	1 229 803	getItem	1 143 441	1 143 945
...	...	...	...	...	...	...	...	...

Figure 8.6.: The result of the ***scTrace*** for the negative sub scenario of Figure 8.1. The result is used in the function ***prSel*** for the ***Precede*** instance. The resulting relation contains all tuples except for those in which STS1<STS2 and ETS1>ETS2.

This process is performed in three steps. First are identified tuples of the ***scRelation*** resulting relation whose values in columns corresponding to the closest negation containing element satisfy the contain interrelation with the values in columns of the negated sub scenario. The closest negation containing event is the root of the smallest abstract syntax sub tree starting with an instance of the ***Contain*** metaclass and having the negated sub scenario in it. In our example this is the ***Contain*** instance having ***getVideoItem*** as the containing event. In order to specify the negation containing event the mapping uses function ***cNeg*** specified in the following. Let $ss_{neg} \in SubScenario$. Then:

$$cNeg(ss_{neg}) = \begin{cases} cNeg(ss_p) \\ \quad \text{if } \exists ss_p \in SubScenario, p \in Precede(((ss_p, p) \in SSP) \wedge \\ \quad (((p, ss_{neg}) \in PdtoSS) \vee ((p, ss_{neg}) \in PstoSS)) \\ \\ ss_c \\ \quad \text{if } \exists ss_c \in SubScenario, c \in Contain(((ss_c, c) \in SSC) \wedge \\ \quad ((c, ss_{neg}) \in CtoSS)) \end{cases}$$

This function is a recursive function. In the upper case, the negative sub scenario is either a preceding or a preceded sub scenario of an instance of the ***Precede*** metaclass. In this case the function is recursively invoked for this ***Precede*** instance. In the case that the negated sub scenario is a contained sub scenario of an instance of the ***Contain*** metaclass, the result of the function is this instance of the ***Contain*** metaclass.

Then, these values are taken out from from the ***evTrace*** of that root. Finally, the outcome is used for selecting only those tuples of the ***scRelation*** which do not satisfy contains relation with the negated sub scenario.

Finally, the remaining case of the function ***contSel*** is explained on the second example, depicted in Figure 6.3. Let it be assumed that there is an interest of analyzing probabilities of execution of different alternatives. Then scenario would be specified as depicted in Figure 8.7.

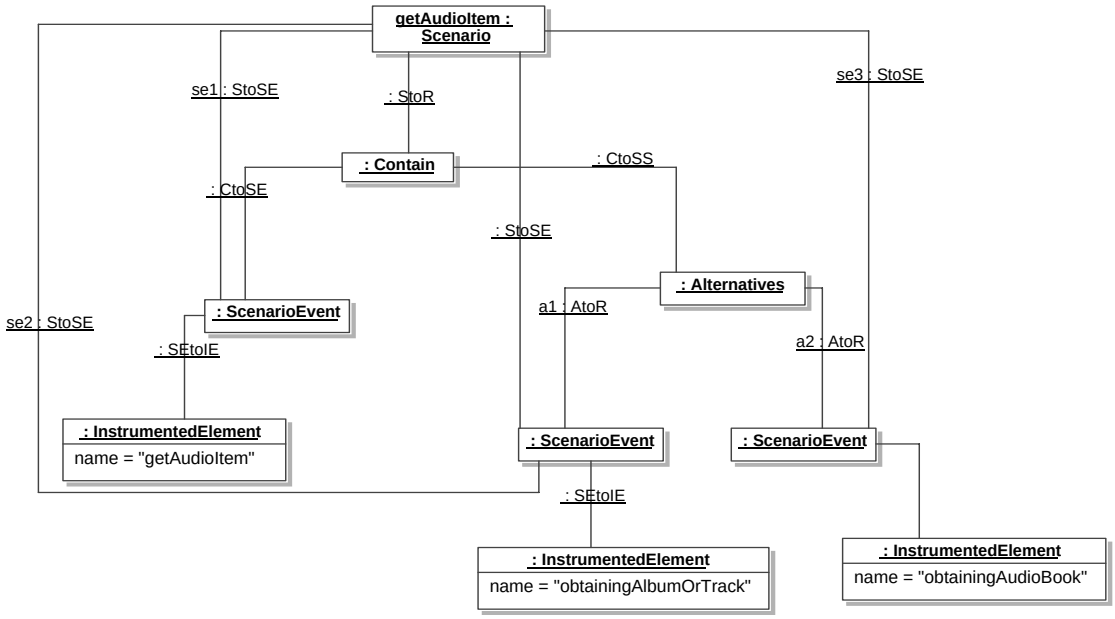


Figure 8.7.: The abstract syntax tree of a ***getAudioItem*** operation part specification.

The instance of ***Contain*** metaclass has an instance of ***ScenarioEvent*** pointing with the ***instrumentedEvent*** attribute to ***getAudioItem*** method. In the contained attribute is an instance of ***Alternatives*** with two instances of ***ScenarioEvent*** metaclass. The instances of ***ScenarioEvent*** metaclass have in ***instrumentedElement*** attribute blocks of the design model first alternative, the alternative for obtaining music albums and tracks. The second instance points with the ***instrumentedElement*** attribute to the alternative for obtaining audio book.

Accordingly, the ***scRelation*** function for this scenario has nine columns. Each scenario event is represented with three columns in the relation. The first three columns contain information about ***getAudioItem*** method executions, while the second and the third triple are dedicated to the blocks of the alternatives. For this reason, the outcome of the ***scTrace*** function for this scenario has the same number of columns. An example of obtained relation from the ***scTrace*** function for this scenario is depicted in Figure 8.8.

In the resulting relation each tuple represents an execution of one of alternatives. For each tuple in relation is valid that (STS1<STS2 AND ETS1>ETS2) OR (STS1<STS3 AND

IE1	STS1	ETS1	IE2	STS2	ETS2	IE3	STS3	ETS3
...	...	...	...	...	...	...	...	...
getAudioItem	2 243 200	2 245 120	obtainingAlbumOrTrack	2 243 210	2 245 102	obtainingAudioBook	3 243 210	3 245 120
getAudioItem	2 243 200	2 245 120	obtainingAlbumOrTrack	2 243 210	2 245 102	obtainingAudioBook	3 758 771	3 788 821
getAudioItem	2 785 750	2 788 830	obtainingAlbumOrTrack	2 758 771	2 788 821	obtainingAudioBook	3 243 210	3 245 120
getAudioItem	2 785 750	2 788 830	obtainingAlbumOrTrack	2 758 771	2 788 821	obtainingAudioBook	3 758 771	3 788 821
...	...	...	...	...	...	...	...	...
getAudioItem	3 243 200	3 245 120	obtainingAlbumOrTrack	2 243 210	2 245 102	obtainingAudioBook	3 243 210	3 245 120
getAudioItem	3 243 200	3 245 120	obtainingAlbumOrTrack	2 758 771	2 788 821	obtainingAudioBook	3 243 210	3 245 120
...	...	...	...	...	...	...	...	...
getAudioItem	3 785 750	3 788 830	obtainingAlbumOrTrack	2 243 210	2 245 102	obtainingAudioBook	3 758 771	3 788 821
getAudioItem	3 785 750	3 788 830	obtainingAlbumOrTrack	2 758 771	2 788 821	obtainingAudioBook	3 758 771	3 788 821
...	...	...	...	...	...	...	...	...

Figure 8.8.: Example of the outcome of the function ***scTrace*** for the Figure 8.7 scenario

ETS1>ETS3). One execution scenario can be repeated in several relations, because of different values of fields that correspond to columns of other alternative. However, all columns of one alternative satisfy specified interrelations. This is achieved with the case ***scTrace*** for alternatives. This case intersects relations satisfying interrelations of one scenario. The intersection produces relations satisfying all sub scenarios. For example, in Figure 8.8, columns STS2 and ETS2 correspond to one, and columns STS3 and ETS3 to the second alternative. This repetition of columns does not have the effect on the result because when selecting the measured event in the function ***msEvTrace*** by performing projection on its columns in the relation, the repeating values are removed.

In the following, two examples are presented in order to be used for the explanation of the remaining cases of the function ***prSel***. Let us assume that there is an additional event before the alternative from the previous case. That event is opening the connection with the database, ***openDBConn***. The abstract syntax for this example is shown in Figure 8.9.

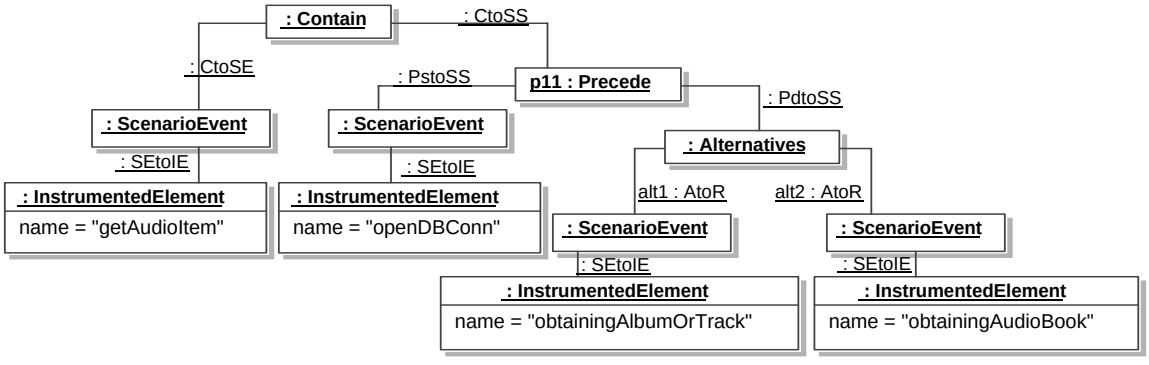


Figure 8.9.: An extended Figure 8.7 example. The opening of a database connection precedes each alternative

The main difference of Figure 8.9 comparing to Figure 8.7 is that the ***getAudioItem*** contains a ***Precede*** instance between having ***openDBConn*** as the preceding and the alternatives as the preceded sub scenario.

The second example adds one more event to the scenario. That is the ***closeDBConn*** event which closes the connection with the database. An abstract syntax for this example is given in Figure 8.10.

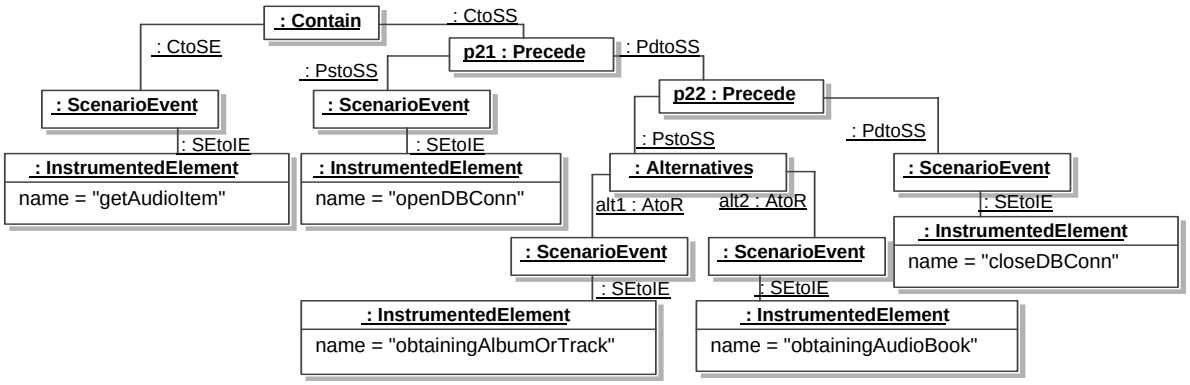


Figure 8.10.: Second extended Figure 8.7 example. The opening of a database connection precedes each alternatives and the closing follows

The major difference comparing to the Figure 8.9 is that the opening of the database connection precedes another instance of the ***Precede*** metaclass. It precedes the precede interrelation between the alternatives and the closing the connection with the database.

The remaining cases of the ***sAf***, ***eBe***, and ***prSel*** mappings use functions ***lEvAltP*** and ***fEvAltP***. These functions are explained in the following.

The function ***lEvAltP*** takes as arguments an alternative sub scenario and a sub scenario and returns the first sub sub scenario of the sub scenario it precedes. For example, in Figure 8.9 for the instance of alternative and the ***p11*** instance of ***Precede*** the function returns the opening database scenario event. This function is formally defined in the following.

Let $ss_{alt}, ss_{pd} \in SubScenario$. Then:

$$lEvAltP(ss_{alt}, ss_{ps}) = \begin{cases} fEv(ss_{psps}) \\ \quad \text{if } \exists p \in Precede, ss_{psps}, ss_a \in SubScenario(((ss_{ps}, p) \in SSP) \wedge \\ \quad ((p, ss_a) \in PdtoSS) \wedge (fEv(ss_a) = lEv(ss_a) = ss_{alt}) \wedge \\ \quad ((p, ss_{psps}) \in PstoSS)) \\ \\ lEvAltP(ss_{alt}, ss_{psps}) \\ \quad \text{if } \exists p \in Precede, ss_{psps} \in SubScenario(((ss_{ps}, p) \in SSP) \wedge \\ \quad ((p, ss_{alt}) \notin PdtoSS)) \wedge ((p, ss_{ps}) \in PstoSS)) \end{cases}$$

The function ***fEvAltP*** is opposite to the function ***lEvAltP***. It returns for a sub scenario and an alternative the first sub sub scenario that precedes that alternative. For example, in Figure 8.10, for the instance of ***Alternatives*** and ***p21*** or ***p22*** it returns the closing database event. This function is defined as follows.

Let $ss_{alt}, ss_{pd} \in SubScenario$. Then:

$$fEvAltP(ss_{alt}, ss_{pd}) = \begin{cases} lEv(ss_{pdpd}) \\ \quad \text{if } \exists p \in Precede, ss_{pdpd}, ss_a \in SubScenario(((ss_{pd}, p) \in SSP) \wedge \\ \quad ((p, ss_a) \in PstoSS) \wedge (fEv(ss_a) = lEv(ss_a) = ss_{alt}) \wedge \\ \quad ((p, ss_{pdpd}) \in PdtoSS)) \\ \\ fEvAltP(ss_{alt}, ss_{pdpd}) \\ \quad \text{if } \exists p \in Precede, ss_{pdpd} \in SubScenario(((ss_{pd}, p) \in SSP) \wedge \\ \quad ((p, ss_{alt}) \in PstoSS) \wedge ((p, ss_{pdpd}) \in PdtoSS)) \end{cases}$$

A traversal of Figure 8.9 example starts with the ***scTrace*** mapping for the top instance of the ***Contain*** metaclass. The ***scTrace*** then enters the third from the top case of the ***contSel*** mapping. The third case uses the result of ***sAf*** for the ***getAudioItem*** and ***openDBConn*** events, ***eBe*** for the ***getAudioItem*** and the instance of ***Alternatives***. For the purpose of illustration let us assume that the scenario relation contains STS1 and ETS1, STS2 and ETS2, STS3 and ETS3, and STS4 and ETS4, corresponding to ***getAudioItem***, ***openDBConn***, ***obtainingAlbumOrTrack***, ***obtainingAudioBook***, respectively. The function ***sAf*** for the ***getAudioItem*** and ***openDBConn*** returns tuples satisfying STS1<STS2. This relation is intersected with the result of ***eBe*** for ***getAudioItem*** and the instance of ***Alternatives***. This is the bottom case of the ***eBe*** function which for this case returns relation satisfying ((ETS1<ETS3) AND (ETS2<ETS3)) OR (((ETS1<ETS4) AND (ETS2<ETS4))). The intersected result is the relation (STS1<STS2) AND (((ETS2<STS3)AND(ETS3<ETS1)) OR (((ETS2<STS4) AND (ETS1<ETS4)))). It means that ***getAudioItem*** contains union of relations having ***openDBConn*** which precedes ***obtainingAlbumOrTrack*** or ***obtainingAudioBook***. It is actually the relation containing executions of the scenario. Nevertheless, the function continues the traversal by invoking the ***prSel*** function for the ***p11*** instance.

The ***prSel*** function for the Figure 8.9 ***p11*** instance, is the third case of this function. It uses the function ***plpfSel***. The ***plpfSel*** function is defined in the following. In order to avoid repetitions, it is assumed that $se_{ps}, se_{pd} \in ScenarioEvent, a_{ps}, a_{pd} \in Alternatives, alt, alt_{ps}, alt_{pd} \subseteq AtoR$ in the following definition. And let $sc \in Scenario$,

$ps, pd \in SubScenario$.

$$plpfSel(sc, ps, pd) = \begin{cases} \sigma[<]_{3*ordNumSEv(sc,se_{ps}),3*ordNumSEv(scenario,sepd)}(scRelation(sc)) \\ \quad \text{if } \exists se_{ps}, se_{pd}(((lEv(ps), se_{ps}) \in SSSE) \wedge \\ \quad ((fEv(pd), se_{pd}) \in SSSE)) \\ \\ plpfSelSEA(sc, ps, pd) \\ \quad \text{if } \exists a_{pd}, alt \ \nexists a_{ps} \forall (a, ass) \in alt(((ps, a_{ps}) \in SSA) \wedge \\ \quad ((pd, a_{pd}) \in SSA) \wedge (a = fEv(pd)) \wedge (n = card(alt))) \\ \\ plpfSelASE(sc, ps, pd) \\ \quad \text{if } \exists a_{ps}, alt \forall (a, ass) \in alt(((ps, a_{ps}) \in SSA) \wedge \\ \quad ((pd, a_{pd}) \in SSA) \wedge (a = lEv(ps)) \end{cases}$$

The function recognizes three different cases. The top case is when the last event of the ***preceding*** sub scenario and the first event of the ***preceded*** sub scenarios are scenario events. Here, the result is a subset of the ***scRelation*** having values in column corresponding to the end of the last event of the first sub scenario smaller than values in the column corresponding to the start of the first event of the second.

The second case is when the last event of the preceding scenario is a scenario event and the first event of the preceded scenario is an instance of ***Alternatives***. This is the case is entered when visiting the Figure 8.9 ***p11*** instance with ***scTrace***. Here, the ***scTrace*** mapping uses the second case from the top of the ***prSel*** mapping with ***openDBConn*** and the ***Alternatives***. The ***prSel*** mapping uses ***plpfSelSEA*** function.

The ***plpfSelSEA*** function is defined in the following. Let $sc \in Scenario, ps, pd \in SubScenario, a_{pd} \in Alternatives$, and $alt \subseteq AtoR$. Then:

$$plpfSelSEA(sc, ps, pd) = \begin{cases} ((plpfSel(sc, ps, altSubSc(a_{pd}, 1))) \cup ... \cup \\ (plfpSel(sc, ps, altSubSc(a_{pd}, n)))) \\ \quad \text{if } \exists a_{pd}, alt \forall (a, ass) \in alt((fEv(pd) = lEv(pd)) \wedge \\ \quad (a_{pd} = fEv(pd)) \wedge (a = a_{pd}) \wedge (n = card(alt))) \\ \\ ((plpfSel(sc, ps, altSubSc(a_{pd}, 1)) \cap \\ (plpfSel(sc, altSubSc(a_{pd}, 1), ss_{altpd}))) \cup ... \cup \\ (plfpSel(sc, ps, altSubSc(a_{pd}, n)) \cap \\ plpfSel(sc, altSubSc(a_{pd}, n), ss_{altpd}))) \\ \quad \text{if } \exists ss_{altpd} \in SubScenario, a_{pd}, alt \forall (a, ass) \in alt \\ \quad ((fEv(pd) \neq lEv(pd)) \wedge (a_{pd} = fEv(pd)) \wedge (a = a_{pd}) \\ \quad (n = card(alt)) \wedge (ss_{altpd} = fEvAltP(a_{pd}, pd))) \end{cases}$$

For the ***p11*** instance the function uses the upper case, where ***ps*** is the ***openDBConn*** event and ***pd*** is the instance of ***Alternatives***. With respect to the scenario relation this function produces a condition that ((ETS2<STS3) OR (ETS2<STS4)). This condition is actually already inside the previous condition of this example and has no impact on the final result. Furthermore, the remaining node visits produce neutral results and, therefore, are not described.

The examples for the remaining cases of defined functions are given with the traversal of the Figure 8.10 example. For the purpose of the example let us assume that to the ***getVideoItem***, ***openDBConn***, ***obtainingAlbumOrTrack***, ***obtainingAudioBook***, and ***closeDBConn*** events of the Figure 8.10 example correspond STS1 and ETS1, STS2 and ETS2, STS3 and ETS3, STS4 and ETS4, and STS5 and ETS5 scenario trace columns, respectively.

The traversal again starts with ***scTrace*** mapping for the top instance of the ***Contain*** metaclass. It uses, similarly to the previous example, the third case of the ***contSel*** mapping. The third case intersects result of ***sAf*** for the ***getAudioItem*** and ***openDBConn***, ***eBe*** for the ***getAudioItem*** and ***closeDBConn***, and ***scTrace*** for the ***p21*** instance. Functions ***sAf*** and ***eBe*** produce relations satisfying STS1<STS2 and ETS1>ETS5, respectively.

When visiting ***p21***, the ***scTrace*** function uses the fourth case of the ***prSel*** mapping. The ***prSel*** mapping uses the bottom case of the ***plpfSelSEA*** function with ***openDB-Conn*** and the instance of ***Alternatives***. Here, each alternative has to additionally satisfy the precede relation with the ***closeDBConn***. The result of this mapping is the function satisfying the condition ((ETS2<STS3) AND (ETS3<STS5)) OR ((ETS2<STS4) AND (ETS4<STS5)). When incorporating the previous conditions, the tuples in the resulting relation satisfy the following condition (STS1<STS2) AND (ETS1>ETS5) AND ((ETS2<STS3) AND (ETS3<STS5)) OR ((ETS2<STS4) AND (ETS4<STS5)). This is, actually, the condition which describes the specified scenario. However, because this is the fourth case of the ***prSel*** mapping, this resulting relation is intersected results of ***scTrace*** for the ***openDB-Conn*** scenario event and the ***p22*** instance. The ***scTrace*** result for the ***openDBConn*** is the neutral result, and the ***scTrace*** result for the ***p22*** instance enters the bottom case of the function ***plpfSel***.

The bottom case of the function ***plpfSel*** is the case when the first event of the preceding sub scenario is an alternative. For computing this case is used function ***plpfSelASE***. The function ***plpfSelASE*** is defined in the following.

Let $sc \in Scenario, ps, pd \in SubScenario, a_{ps} \in Alternatives$, and $alt \subseteq AtoR$. Then:

$$plpfSelASE(sc,ps,pd) = \begin{cases} (plpfSel(sc, altSubSc(a_{ps},1),pd)) \cup ... \cup \\ (plpfSel(sc, altSubSc(a_{ps},n),pd)) \\ \quad \text{if } \exists a_{ps}, alt \forall (a,ass) \in alt((fEv(ps) = lEv(ps)) \\ \quad \wedge (a_{ps} = fEv(ps)) \wedge (a = a_{ps}) \wedge (n = card(alt))) \\ \\ ((plpfSel(sc, altSubSc(a_{ps},1),pd)) \cap \\ (plpfSel(sc, ss_{altps}, altSubSc(a_{ps},1)))) \cup ... \cup \\ ((plpfSel(sc, altSubSc(a_{ps},n),pd)) \cap \\ (plpfSel(sc, ss_{altps}, altSubSc(a_{ps},n)))) \\ \quad \text{if } \exists ss_{altps} \in SubScenario, a_{ps}, alt \forall (a,ass) \in alt \\ \quad ((fEv(ps) \neq lEv(ps)) \wedge (a_{ps} = fEv(ps)) \wedge (a = a_{ps}) \\ \quad \wedge (n = card(alt)) \wedge (ss_{altps} = lEvAltP(a_{ps}, ps))) \end{cases}$$

The function ***plpfSelASE*** similarly to ***plpfSelSEA*** recognizes two cases. The upper case is the case when alternatives must not satisfy some other precede relation. And the bottom when they do. In visiting ***p22*** each alternative must satisfy the condition that it is followed by the ***closeDBConn*** event. It produces the condition (ETS3<STS5) OR (ETS4<STS5). This result is already contained in the previous result and, therefore, does not have an impact on it. The the results of traversals the remaining nodes are neutral results.

It should be noticed that this case is used also when both first event of the preceding and last case of the preceded sub scenario are alternatives. The recursive logic of this function does not change if the last event of the preceded sub scenario is not a scenario event but a At the end, grouping events into one relation later used in statistical analysis is illustrated with the example in Figure 8.11.

The example in Figure 8.11 represents a specification of a group containing two alternative blocks of the scenario in Figure 8.7 as elements of that group. This specification is used later for the illustration of metrics and assessment metamodel parts.

The function ***grTr*** is defined as follows.

Let $g \in Group$. And let $(g, me_1), (g, me_2), ..., (g, me_n) \in GtoME$ be all pairs of ***GtoME*** having ***g*** as the value of the first entry. Then the function ***grTr*** is defined as:

$$grTr(group) = msEvTrace(me_1) \cup msEvTrace(me_2) \cup ... \cup msEvTrace(me_n)$$

This function groups all specified measured event into one relation by performing union of group measured events relations. For example, if the group is defined as in Figure 8.11, where the specified scenario is the scenario represented in Figure 8.8, the resulting relation is given in Figure 8.12.

The resulting relation contains executions of both measured events summarized in one table. The resulting group relation for Figure 8.1 contains only executions of the ***getItem*** operation.

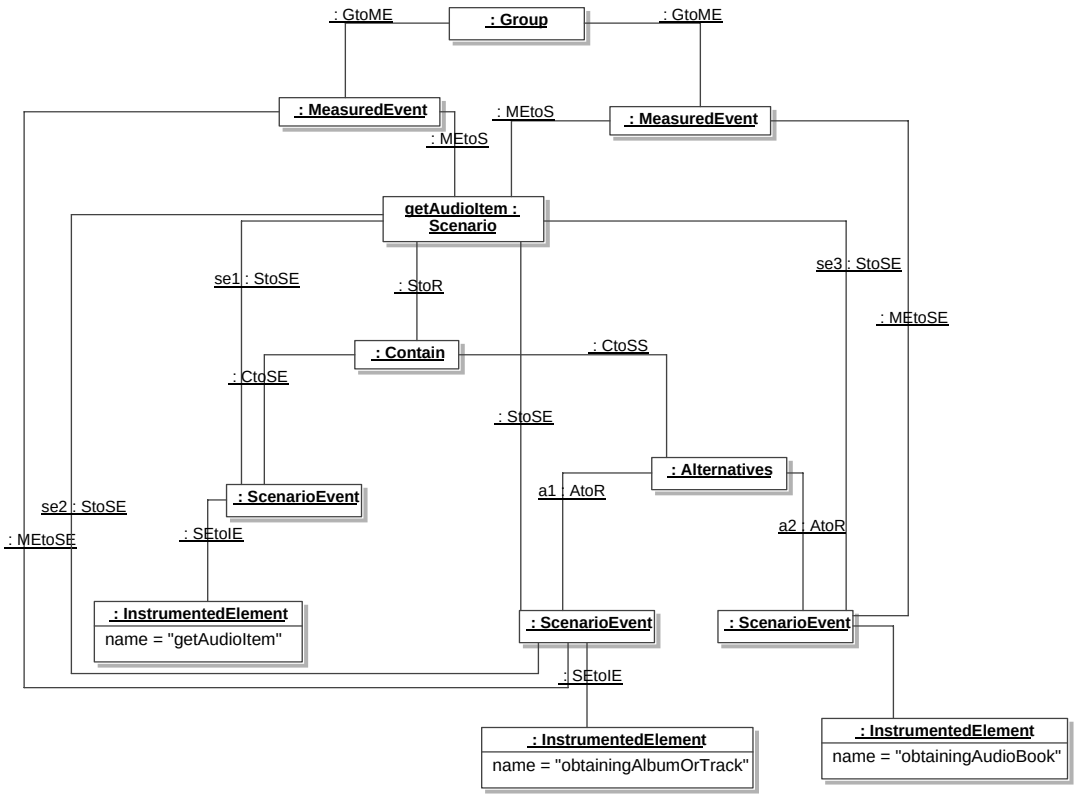


Figure 8.11.: Example of the ***Group*** metaclass usage. In the figure can be seen a grouping of alternative sub scenarios of Figure 8.7 for further assessment.

At the end, it should be noticed the need for the well-formedeness rule 7 at the page 77. If there would be sharing of instances representing scenario events in a scenario, the computation of metrics might not be correct. The incorrectness may arise when it comes to a situation in which the values in the columns corresponding to the shared event have to satisfy $<$ or $>$ relation with themselves as the result of specified interrelations in the scenario.

IE	STS	ETS
...	...	...
obtainingAlbumOrTrack	2 243 210	2 245 102
obtainingAlbumOrTrack	2 758 771	2 788 821
obtainingAudioBook	3 243 210	3 245 120
obtainingAudioBook	3 758 771	3 788 821
....	...	...

Figure 8.12.: Example of the ***Group*** metaclass usage. In the figure can be seen a grouping of alternative sub scenarios of Figure 8.7 for further assessment.

8.3. Assessment and Metrics Metamodel Part Formal Semantics

Assessment and metrics specification subsequently use the relation obtained with ***grTr*** function. The group events and their duration intervals are further used for the computation of specified metrics. As defined in the metamodel, there exist two assessments, a simple and a composite one. Mapping of an assessment to the appropriate query is defined by function ***assessment***, formally specified in the following. Let $as \in Assessment$. Then,

$$assessment(as) = \begin{cases} sAssesment(as) \\ \quad \text{if } \exists sas \in SimpleAssessment((as, sas) \in AsSA) \\ \\ cAssessment(as) \\ \quad \text{else} \end{cases}$$

The ***assessment*** function simply recognizes if the assessment is simple or composite and, accordingly, the image if this function is the result of ***sAssessment*** and ***cAssessment***, respectively.

The ***sAssessment*** function computes desired metrics for a simple assessment. This mapping is defined in the following.

Let $sas \in SimpleAssessment$. In order to avoid repetitions let $ass \in Assessment, m \in Metric, is \in IntervalSet, tis \in TimeIntervalSet, gr \in Group$. Then:

$$sAssessment(sas) = \begin{cases} \pi_{1,2,8}(Apply[/]_{6,7}(Apply[-]_{2,1}(Aggr_1[5:c](\sigma[\geq]_{2,4}(\sigma[<]_{1,3}(\\ iSRel(is) \times grTr(gr))))))) \\ \quad \text{if } \exists ass, m, is, tis, gr \exists or \in OccurrenceRate(((ass, sas) \in AsSA) \\ \quad \wedge((ass, tis) \in AstoTIS) \wedge ((is, tis) \in ISTIS) \wedge \\ \quad ((sas, m) \in SAtoM) \wedge ((sas, g) \in SAtoG) \wedge ((m, or) \in MOR)) \\ \\ \pi_{1,2,3,8}(Apply[/]_{6,7}(Aggr_1[5:c]((Aggr_3[5:c](\sigma[\geq]_{2,4}(\sigma[<]_{1,3}(\\ iSRel(is) \times grTr(gr)))))))) \\ \pi_{1,2,8}(Apply[/]_{6,7}(Apply[-]_{2,1}(Aggr_1[5:c](\sigma[\geq]_{2,4}(\sigma[<]_{1,3}(\\ iSRel(is) \times grTr(gr))))))) \\ \quad \text{if } \exists ass, m, is, tis, gr \exists op \in OccurrencePercentage \\ \quad (((ass, sas) \in AsSA) \wedge ((ass, tis) \in AstoTIS) \wedge \\ \quad ((is, tis) \in ISTIS) \wedge ((sas, m) \in SAtoM) \wedge ((sas, g) \in SAtoG) \\ \quad \wedge((m, op) \in MOP)) \\ \\ durAssessment(sas) \\ \quad \text{if } \exists m \exists d \in Duration(((sas, m) \in SAtoM) \wedge ((m, d) \in Duration)) \end{cases}$$

Function ***sAssessment*** defines different mappings according to the metric specified for computation. In the case that the specified metric is occurrence rate, the function classifies events according to time intervals in which they occur. Time intervals are specified in the ***timeIntervalSet*** attribute of the ***SimpleAssessment*** instance. When they are classified, then they are counted for each interval and divided by the duration of the interval. Finally, the resulting relation is obtained by performing projection on the columns representing interval start times, interval end times, and values of occurrence rates.

In the case that the specified metric is occurrence percentage, the group events are again classified according to time intervals in which they occur. Then the total number of events in a particular interval, and number of particular modeling element executions in the same interval are computed. Thereafter, count of modeling element execution is divided by total number, and the desired percentage is calculated. Finally, the performed projection isolates interval start and end time instants, modeling element, and percentage of execution. For example, if the percentage of different alternatives specified in Figure 8.7, at the page 95, is of interest the computation would be performed in the following way. First, the number of records in the relation 8.8. Then, the numbers of occurrences of ***obtainingAlbumOrTrack*** and ***obtainingAudioBook*** would be computed. Rations of these numbers with the total numbers of occurrences are percentages of occurrences. The final relation consists of the following four columns: interval start time, interval end time, the instrumented element identifier, and the value of percentage.

When the specified metric to be computed is some statistical analysis of duration, the result of ***sAssessment*** function is result of ***durAssesssment*** function, as specified in the following.

Let $sas \in SimpleAssessment$. Similarly to the previous case, for the reason of brevity and avoidance of repetitions,$ass \in Assessment, m \in Metric, d \in Duration, a \in Analysis, is \in IntervalSet, tis \in TimeIntervalSet, gr \in Group$ in both cases.

$$durAsssessment(sas) = \begin{cases} \pi_{1,2,7}(Aggr_2[6 : agg(sf)](Apply[-]_{5,4}(\sigma[\geq]_{2,4}(\sigma[<]_{1,3}(\\ iSRel(tis) \times grTr(gr)))))) \\ \quad \text{if } \exists ass, m, d, a, is, tis, gr \exists sa \in StatisticalAnalysis \\ \quad (((ass, sas) \in AsSA) \wedge ((ass, tis) \in AstoTIS) \wedge \\ \quad ((is, tis) \in ISTIS) \wedge ((sas, m) \in SAtoM) \wedge \\ \quad ((sas, g) \in SAtoG) \wedge ((m, d) \in MD) \wedge ((d, a) \in DA) \wedge \\ \quad ((a, sa) \in ASA) \wedge (sa = (sf)) \wedge (sf \in Statistics)) \\ \\ durDisA(sas) \\ \quad \text{if } \exists m, d, a \exists da \in DistributionAnalysis((sas, m) \in SAtoM) \\ \quad \wedge((m, d) \in MD) \wedge ((d, a) \in DA) \wedge ((a, da) \in AD)) \end{cases}$$

The duration assessment is done with some statistical analysis of all occurrences in time interval. Occurrence durations are assessed either with some statistical function which describes all durations in one number, or in distribution histograms. Therefore, ***durAssssessment*** recognizes two cases.

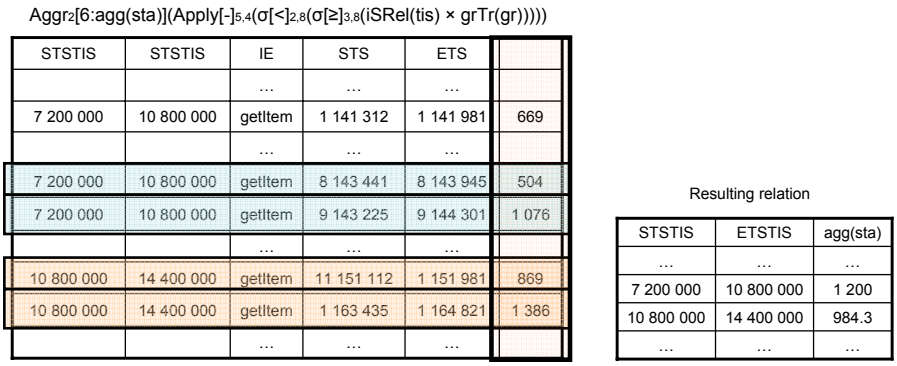


Figure 8.13.: The illustration of the duration statistical analysis. The statistical value is computed for each calendar time interval

In one case the assessment is performed according to a statistical function which describes all occurrences of an interval in one number. Here, durations of all events of a group which occurred in the same time interval are banded, their durations computed, and then statistically analyzed with the predefined function. Finally, projection is performed to leave in only the start and end time instant of the interval and the value of the function specified in resulting relation. The computation is illustrated in Figure 8.13.

The second case is when duration of modeling elements are assessed with some distribution function. Then, the result of the function ***durAssessment*** is result of function ***durDistA***. Function ***durDistA*** is defined in the following.

Let $sas \in SimpleAssessment$. Here is also assumed that $ass \in Assessment$, $m \in Metric$, $d \in Duration$, $a \in Analysis$, $di \in Distribution$, $is, is_d \in IntervalSet$, $tis \in TimeIntervalSet$, $gr \in Group$ for the reason of brevity and avoidance of repetitions.

$$durDisA(sas) = \begin{cases} \pi_{1,2,3,4,10}(Apply[/]_{9,10}(Aggr_2[5:c](Aggr_4[5:c](\sigma[<]_{2,8}(\sigma[\geq]_{3,8}(iSRel(is_d) \\ \times Apply[-]_{5,4}(\sigma[\geq]_{2,4}(\sigma[<]_{1,3}(iSRel(is) \times grTr(gr))))))))) \\ \quad \text{if } \exists ass, tis, is, is_d, m, d, a, di(((ass, sas) \in AsSA) \wedge \\ \quad ((ass, tis) \in AstoTIS) \wedge ((is, tis) \in ISTIS) \in (sas, gr) \in SAtoG) \wedge \\ \quad ((sas, m) \in SAtoM) \wedge ((m, d) \in MD) \wedge ((d, a) \in DtoA) \wedge \\ \quad (((a, di) \in AD) \wedge = (kind)) \wedge (kind \in DistributionKind) \wedge \\ \quad (kind = density) \wedge ((di, is) \in DtoIS)) \\ \\ \pi_{1,2,3,4,10}(Apply[/]_{9,10}(Aggr_2[5:c](Aggr_4[5:c](\sigma[\geq]_{2,8}(iSRel(is_d) \\ \times Apply[-]_{5,4}(\sigma[\geq]_{2,4}(\sigma[<]_{1,3}(iSRel(is) \times grTr(gr)))))))) \\ \quad \text{if } \exists ass, tis, is, is_d, m, d, a, di(((ass, sas) \in AsSA) \wedge \\ \quad ((ass, tis) \in AstoTIS) \wedge ((is, tis) \in ISTIS) \in (sas, gr) \in SAtoG) \wedge \\ \quad ((sas, m) \in SAtoM) \wedge ((m, d) \in MD) \wedge ((d, a) \in DtoA) \wedge \\ \quad (((a, di) \in AD) \wedge (di = (kind)) \wedge (kind \in DistributionKind) \wedge \\ \quad (kind = cumulative) \wedge ((di, is) \in DtoIS)) \end{cases}$$

Function ***durDistA*** recognizes two cases: assessment with histogram, and with cumulative histogram. In the case where density distribution is specified as the assessment metric, histogram is computed for each time interval. Events are clustered according to the time interval in which they occur. Subsequently, the duration of each event is computed and count of events in the time interval. Afterwords, events are grouped according to their duration. All events which is in the same intervals of the ***is*** attribute of the ***Distribution*** metaclass are clustered, and their count computed. Then, count of events of each interval cluster is divided by the count of events in the time interval for which histogram is computed. Finally, the projection leaves only columns of start and end of time interval for which is histogram computed, start and end values of each interval in histogram, and values of bars in the histogram. The procedure is the same for the cumulative histogram, only events in the histogram are clustered in the way, that all events whose duration is smaller than the end value of the interval, belong to that interval in the cumulative histogram. The density distribution histogram computation is illustrated in Figure 8.14.

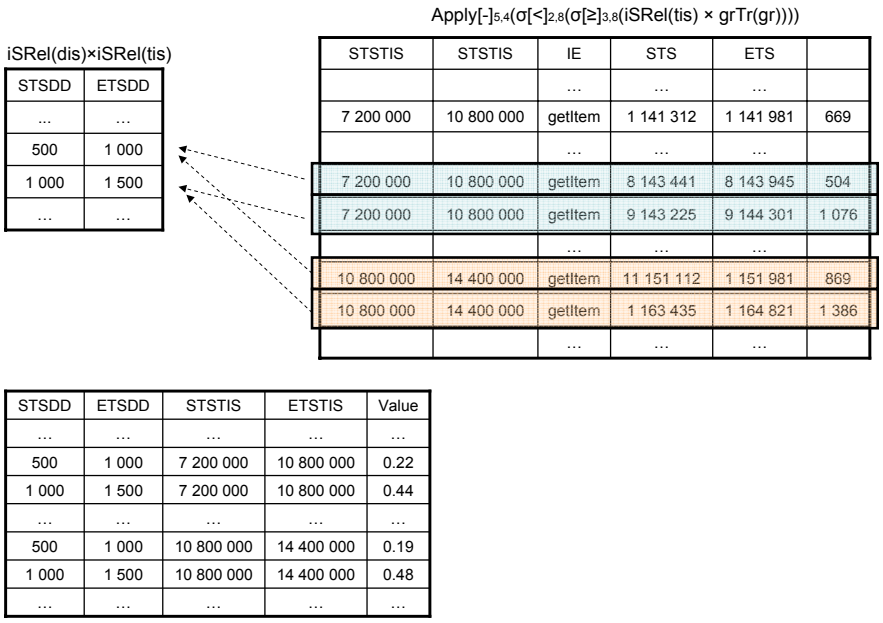


Figure 8.14.: The illustration of the histogram computation. The histogram computed for each calendar time interval. It computed by dividing numbers of events of a calendar time interval belonging to histogram intervals with the number of events of that time interval

Besides simple assessment, also composite assessment can be specified. Composite assessment enables further statistical analysis of simple assessment results. An example of such an analysis is the computation of mean throughput. In this case, throughput is computed for some representative time intervals. After the computation for time intervals, values of throughput for that time intervals are taken and the mean computed. From this example, it can be noticed that a composite assessment takes results of some already computed metrics and then statistically analyzes them. The computed metrics which are input for the composite analysis are defined in the attribute ***previousLevelAssessement***. Composite

assessment computation is formally defined in the following.

Let $cas \in CompositeAssessment$. Similarly to previous cases, in order to avoid repetitions in all cases is $ass, pla \in Assessment, a \in Analysis, is \in IntervalSet, tis \in TimeIntervalSet$.

$$cAssessment(cas) = \begin{cases} \pi_{1,2,7}(Aggr_3[5 : agg(sta)](\sigma[\geq]_{2,4}(\\ \sigma[<]_{1,3}(iSRel(tis) \times assessment(pla))))) \\ \quad \text{if } (\exists ass, pla, a, is, tis \exists sa \in StatisticalAnalysis \\ \quad \wedge(((ass, cas) \in AsCA) \wedge ((ass, tis) \in AstoTIS) \wedge \\ \quad \wedge((is, tis) \in ISTIS) \wedge ((cas, pla) \in CAtoAS) \wedge \\ \quad ((cas, a) \in CAtoA) \wedge ((a, sa) \in ASA) \wedge (sa = sta)) \\ \\ cAssDistA(cas) \\ \quad \text{if } \exists a \exists di \in Distribution(((cas, a) \in CAtoA) \wedge ((a, di) \in AD)) \end{cases}$$

Function ***cAssessment*** separates two cases, similarly to duration simple assessment. In the case when the composite assessment is specified in such a way that the attribute ***metricAnalysis*** is an instance of ***StatisticalFunction***, for each interval defined in the ***timeIntervalSet*** attribute of the defined statistical function from ***Statistics*** is computed in the following way. Previous level assessment, which contains start and end of time interval for which they are computed, and the value of computed metric, are grouped according to the intervals of composite analysis to which they belong. Afterwords, for each time interval of composite assessment, statistical function specified in the attribute ***statisticalFunction*** are computed on these grouped values.

Alternative to statistical analysis where set of values are represented with a single number is the analysis of previous level assessment values with distribution functions. Such an example is the density distribution of throughput. The computation is performed in the following way. As in the previous case, values of previous level assessment metrics are grouped according to the interval of composite assessment to which they belong. Later, for these groups, according to kind of distribution function, density or cumulative, and width of histogram or cumulative histogram, bars and the histogram bar values are computed. The computation is similar to the computation of distribution histograms in simple assessment of durations. Formally, this function is defined in as follows.

Let $cas \in CompositeAssessment$. Again, to avoid repetitions and to shorten the definition, it is assumed that $ass, pla \in Assessment, a \in Analysis, di \in Distribution, is, is_d \in IntervalSet, tis \in TimeIntervalSet$.

$$cAssDistA(cas) = \begin{cases} \pi_{1,2,3,4,10}(Apply[/]_{8,9}(Aggr_2[5:c](Aggr_4[5:c](\sigma[<]_{2,7}(\\ \sigma[\geq]_{2,7}(iSRel(is_d) \times \sigma[\geq]_{2,4}(\sigma[<]_{1,3}(iSRel(is) \times ass(pla))))))))) \\ \quad \text{if } \exists ass, pla, a, di, is, is_d, tis((ass, cas) \in AsCA) \wedge \\ \quad ((ass, tis) \in AstoTIS) \wedge ((is, tis) \in ISTIS) \wedge \\ \quad ((cas, pla) \in CAtoAs) \wedge ((cas, a) \in CAtoA) \wedge ((a, di) \in AD) \wedge \\ \quad (di = (kind)) \wedge (kind \in DistributionKind) \wedge (kind = density) \wedge \\ \quad ((di, is_d) \in DtoIS)) \\ \\ \pi_{1,2,3,4,10}(Apply[/]_{8,9}(Aggr_2[5:c](Aggr_4[5:c](\sigma[\geq]_{2,7}(\\ iSRel(is_d) \times \sigma[\geq]_{2,4}(\sigma[<]_{1,3}(iSRel(is) \times assessment(pla)))))))) \\ \quad \text{if } \exists ass, pla, a, di, is, is_d, tis((ass, cas) \in AsCA) \wedge \\ \quad ((ass, tis) \in AstoTIS) \wedge ((is, tis) \in ISTIS) \wedge \\ \quad ((cas, pla) \in CAtoAs) \wedge ((cas, a) \in CAtoA) \wedge ((a, di) \in AD) \wedge \\ \quad (di = (kind)) \wedge (kind \in DistributionKind) \wedge \\ \quad (kind = cumulative) \wedge ((di, is_d) \in DtoIS)) \end{cases}$$

Part III.

Evaluation

9. Chapter
PEMA: A UML Profile for PErformance Measurement and Assessment

Model Driven Architecture, the mostly used MDE approach as described in Chapter 2, in praxis uses UML as de facto standard for software intensive systems engineering. With means of UML Profile, UML can be adapted to some domain-specific usage. The size of the UML metamodel, starting with version 2.0, grew very large and became very complex. Such a big metamodel is often unneeded for particular uses of UML. Therefore, it is suggested that for a particular purpose, only the needed subset is used [France et al., 2006]. In this chapter, the part of UML which enables modeling of the reactive and the transformational part is specified in Section 9.1. Section 9.2 contains specification of the PEMA UML profile as an implementation the metamodel presented in the previous chapter.

9.1. UML Metamodel Subset

MoDePeMART is a general approach for integration of performance measurement and assessment in an MDE software development process. In the measurement and assessment, as explained in Chapter 5, two models are used: a software design model, and a measurement and metric computation specification model. In MDA the UML is used for both models. For evaluation of the approach application to MDA, UML Class Diagrams, UML State Diagrams, and UML Sequence Diagrams are used for software functionality modeling.

UML Class Diagrams and UML State Diagrams are used as prescriptive models from which code is generated. Generation of code from State and Class diagrams is already widely used, e.g. by Hubert [Hubert, 2002] and Harel and Gery [Harel and Gery, 1996].

For bodies of methods in UML Class and State diagrams in some approaches an action languages are used, e.g. by Raistrick et al. Raistrick et al. [2004]. UML, starting with version 2.0, includes action semantics in its metamodel. Action semantics is an abstract syntax of action languages. However, there is no specified concrete syntax of an action language. For this reason, most of currently available tools for UML modeling do not contain an action language. Such a tool is the tool used for evaluation in this thesis. Consequently, in evaluation as an action language, standard Java syntax is used.

UML Sequence Diagrams are not used for code generation. Nevertheless, they can be used as prescriptive models for software performance prediction, as suggested by Cortellessa and Mirandola [Cortellessa and Mirandola, 2000]. In the next two subsection subsets of UML Class and UML State Diagrams used for evaluation are explained.

9.1.1. Class Diagrams UML Metamodel Subset

Class diagrams are used for modeling software intensive systems structure. UML Class Diagram metamodel subset used for evaluation is specified in Figure 9.1.

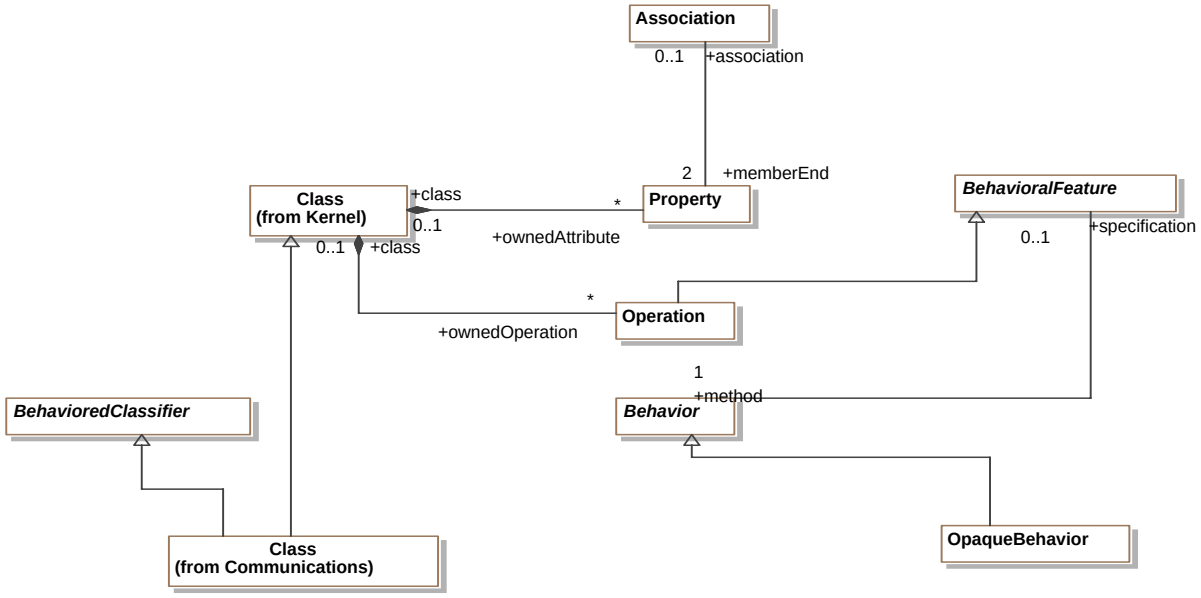


Figure 9.1.: UML Class Diagrams UML metamodel subset

The metaclass used for the definition of a class in the model is ***Class*** metaclas from the ***Communications*** package. ***Class*** metaclass from ***Kernel*** package is generalization of concepts used for various definitions in the UML metmodel. Metaclass ***Class*** from ***Communications*** package inherits ***BehavioredClassifier*** metaclass. ***BehavioredClassifier*** is the metaclass which owns a behavior additional to one specified in operations. This enables the specification a state machine as a reactive class behavior. The metamodel part for specification of reactive behavior is explained in the next subsection.

The ***Class*** from ***Communications*** package has attributes ***ownedAttribute*** and ***ownedOperation*** inherited from metaclass ***Class*** from ***Kernel*** package. Attributes ***ownedAttribute*** and ***ownedOperation*** are of types ***Property*** and ***Operation***, respectively. These attributes at M2 layer, enable specification of attributes and operations at M1 layer, respectively.

Metaclass ***Association*** enables specification of associations in class diagrams. In order to enable usage of standard Java instead of action language, attribute method of the class ***Operation*** is of type ***OpaqueBehavior***. Metaclass ***OpaqueBehavior*** enables usage of programming language expression in UML diagrams.

Additionally, metaclasses ***Operation*** and ***Property*** have an additional attribute ***type***, not presented in the diagram. In the case of ***Property*** metaclass, it is used for definition of the M1 class attribute. For ***Operation*** metaclass, it is used for specification of operation return value data type. Furthermore, UML metamodel parts for defining operation arguments are also not depicted in the figure. These are not shown for the reason of their complexity.

Finally, the stereotype ***Client*** is also defined. This stereotype is introduced because of usage of a client server application architecture for evaluation. This stereotype is used for de-

notation of classes which are part of client applications. Furthermore a stereotype ***DataType*** is also defined. The ***DataType*** stereotype is introduced in order to make a difference between the composite data types and the application logic implementation classes. Classes without both stereotypes are considered parts of server application logic. This information is used for code generation, as it is explained in Appendix E.

UML metamodel parts for definition of attribute types, operation arguments and return value of attributes are, for the reason of brevity, here not presented.

9.1.2. State Machines UML metamodel subset

As explained in Section 6.1 the object oriented version of Harel's statecharts are in the core of the UML. In UML they are called UML State Machines. In order to enable reactive modeling, for evaluation, a part of metamodel defining UML State Machines is used. The part used for evaluation is presented in Figure 9.2.

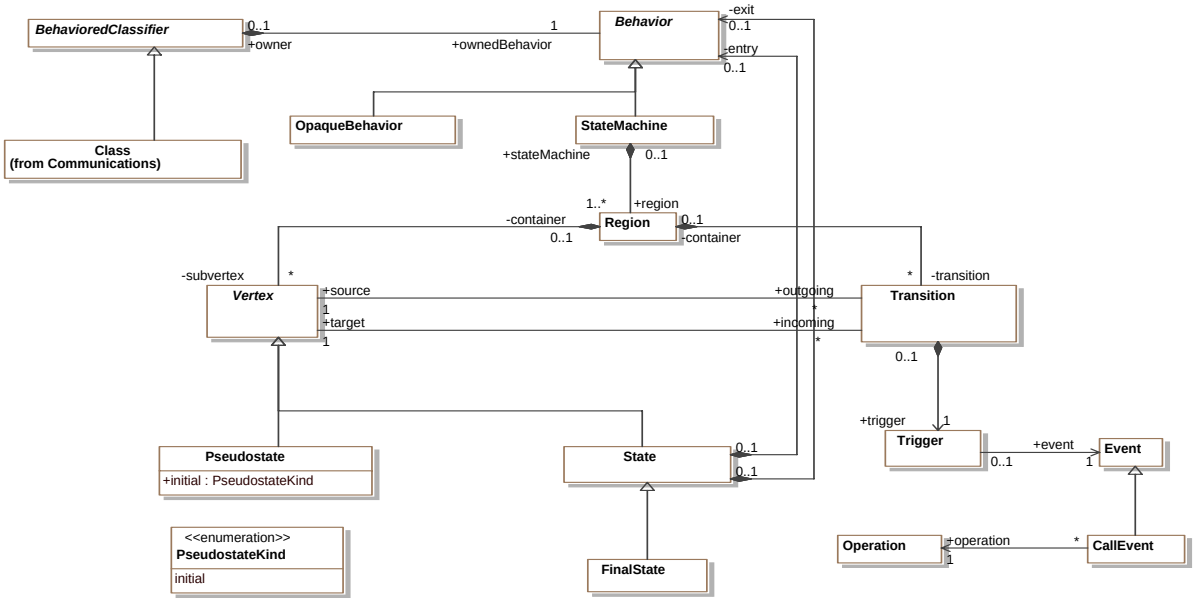


Figure 9.2.: UML State Machines UML metamodel subset

Each class with the reactive behavior has defined a state machine in the attribute ***ownedBehavior***. The state machine which defines the reactive behavior is an instance of the ***StateMachine*** metaclass.

Each state machine consists of states and transitions. States are instances of metaclasses ***State***, ***FinalState*** and ***PseudoState***. Instances of ***FinalState*** metaclass are used for modeling of terminating state of the state machine. Initial state is modeled with the ***PseudoState*** metaclass instances and the value ***initial*** of the attribute ***kind***. There should be noticed that there are several more possible values of the attribute kind, which are either, because of the assumptions, not able to be used, or not needed to show the feasibility of the approach.

The instances of metaclass ***State*** are used for modeling regular states in the model. Each state in the model has entry and exit behavior, defined in ***entry*** and ***exit*** attributes of type

Behavior. These behaviors are defined as ***OpaqueBehavior***s, which is standard Java code.

All metaclasses for states modeling are generalized as the ***Vertex*** abstract metaclass. ***Vertex*** metaclass defines two attributes ***outgoing*** and ***incoming***. Attribute ***outgoing*** of a state is used for specification of transitions whose end result is leaving that state. On the contrary, the attribute ***incoming*** of a state is used for specification of transitions whose end result is entering that state. Both of these attributes are of type ***Transition***.

Each transition has a trigger, source and target state, defined with attributes ***trigger***, ***source*** and ***target***, respectively. Attribute ***trigger*** is of type ***Trigger***. Because of the synchronous communication assumed in Section 6.1, the only event which can activate the trigger is an operation invocation. Therefore, the attribute ***event*** of this metaclass can have only a value of an instance of a ***CallEvent***. The operation which activates the trigger is defined as ***operation*** attribute of the ***CallEvent*** metaclass.

9.2. PeMA UML Profile

Profiling mechanism in UML enables customization of UML for the particular needs. As described in Section 2.3, elements of a UML Profile are stereotypes, stereotype attributes and model libraries.

The PeMA, UML ***P***rofile for ***Pe***rformance ***M***easurement and ***A***ssessment is specified with class diagrams, already explained in Chapter 7. These classes are defined as a model library in the profile. This profile should not be understood as the final product. It is an initial step which is the subject of the further improvement. More about this can be found in Section 12.

In this profile, several differences exist to the classes used in the metamodel specification, and they are depicted in Figure 9.3.

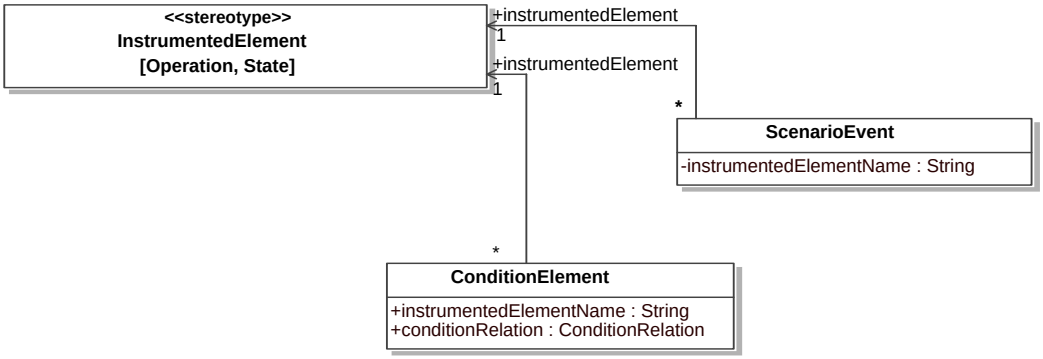


Figure 9.3.: Stereotypes of the PeMA Profile

According the UML metamodel part used for evaluation, defined in the previous section, elements which can be instrumented are states and method invocations. For this reason the stereotype ***InstrumentedElement*** in the PeMA UML Profile is defined to extend metaclasses ***Class*** and ***State***. Instrumented states and operations are annotated in the model with the stereotype ***InstrumentedElement***. Each modeling element stereotyped with an

InstrumentedElement stereotype elements slightly differs code generation. Code generation for instrumented elements adds probes for data collection to the generated code. Furthermore, the information which is stored in the database when execution of that modeling element occurs, is specified in such a way that it describes a modeling and not the construct of the target platform. This is explained in Appendix E.

As previously stated, action semantics in UML, although defined in the metamodel, does not have a concrete syntax. As the state of the art is such that there is no action language concrete syntax, and that behavior specification in method bodies is mostly done by manually coding them in some programming language, ***PeMA*** Profile enables performance measurement and assessment for such an approach.

In order to enable context and performance metrics computation for elements which are specified with code in method bodies, the metaclasses ***ConditionElement*** and ***ScenarioEvent*** are extended with the additional attribute ***instrumentedElementName*** of type ***String***. Such element is intended for elements instrumented at the code level. At the code level, probes have to be added manually. Probes consists of predefined information for database storage, routines for time information collection and some label of the instrumented piece of application. This label user defines alone and it is also added as the value of ***instrumentedElementName*** attribute, either in an instance of ***ScenarioEvent*** or in an instance of ***ConditionElement***.

10. Chapter Transformation to Client Server Applications with Java RMI

With the UML subset and the set of stereotypes defined in the previous chapter client server applications with reactive and transformational behavior can be specified. In order to have implementations of applications modeled with that UML subset, models have to be transformed to execution platforms for client server applications development. Such execution platform is Java with RMI. In this chapter, Section 10.1 depicts transformations from the UML subset and the set of stereotypes to Java with RMI platform. Furthermore, the proof of concept of this transformation is described and validated in Section 10.2.

10.1. Transformation to Client Server Applications with Java RMI

Combined automatic transformations to code for application logic, measurement, and metrics computation is a powerful way to ensure correct data collection and measurements.

The generated code in ***MoDePeMART*** consists of four parts: software functionality code with probes, code for database tables specification, code for database initialization and code for metrics computation. The design of the last three code parts is given as the formal semantics. The design of code for software functionality generation is depicted in this section.

The software functionality code consists of three parts: data collection code, server code and client code. Accordingly, transformation of instrumented client server application model to the target platform is divided to three sub transformations: for performance measurement code generation, for server code generation, and for client code generation. These transformations are defined in Appendix E.

In order to depict the transformation, Figure 10.1 presents Java code generated from the electronic items management application.

Client code consists of a Java class for each client and Java classes which implement clients' data types. Furthermore, client code contains a server access point to the server. This access point is ***ClientSideServerImplementation.java*** class. It is used for sending request and receiving results from the server. It is a Mediator [Gamma et al., 1995] between the client application logic classes and the server.

Server code application logic is defined with ***AudioItemFacade.java***, ***VideoItemFacade.java***, and ***ItemFacade.java***. They are generated from UML Model application logic design classes. Furthermore, for horizontal application logic dimension are generated ***ItemFacadeSTateMachine.java***, ***On.java***, and ***Off.java***. Horizontal server application logic

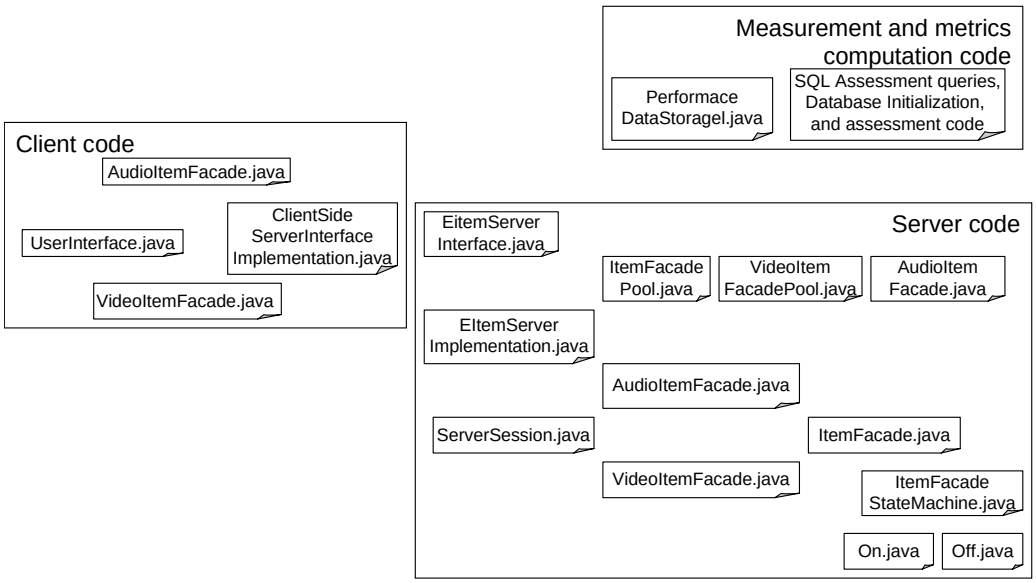


Figure 10.1.: Generated code for electronic item management case study

dimension is ***ItemFacade***'s reactive behavior. It is implemented with the application of State design pattern Gamma et al. [1995]. Classes which provide server services are ***EItemServerInterface.java*** and ***EItemServerImplementation.java*** classes. They are implemented as Facades Gamma et al. [1995] of the server.

The server works in the following way. A client makes a request for the connection to the server. The ***EItemServerImplementation*** allocates a set of application logic classes instances and keeps them for that client. Furthermore, it returns to the server an session identifier. The client sends this request along with the required parameters when making a request to a server's service. Finally, an object of this class also acts as a dispatcher, or as a Mediator [Gamma et al., 1995] between clients and server instances allocated for serving those clients. The server receives a request from a client, and transfers the request exactly to the objects serving that client. Finally, when the client does not need the communication with the server, it sends the disconnecting signal. A server, after having received this signal, returns the allocated objects to the pool and makes them available for serving new clients.

Performance measurement and metrics computation code consists of probes for measurements, interface to the performance database storage, and code for metrics computation. Probes are inserted in application logic implementation classes. Class ***PerformanceDataStorage.java*** implements the interface to the performance database. For metrics computation, the generated is SQL code.

10.2. Validation

The validation of the approach is given with the implementation as a proof of concept and a valuation of the tool. The proof of concept is described in Subsection 10.2.1 and evaluated in Subsection 10.2.2

10.2.1. Implementation

The proof of concept of this approach is implemented with ***MagicDraw 15.1*** Community Edition for UML modeling and profile, ***openArchitectureware 4.2*** for transformation, and ***MySQL 5.2*** as the database management system for data storage and metrics computation.

MagicDraw is the UML modeling tool produced by No Magic, Inc. It is one of widely used tool for modeling with UML. The tool is used for software functionality modeling. The case study class model developed in the tool can be seen in Figure 10.2.

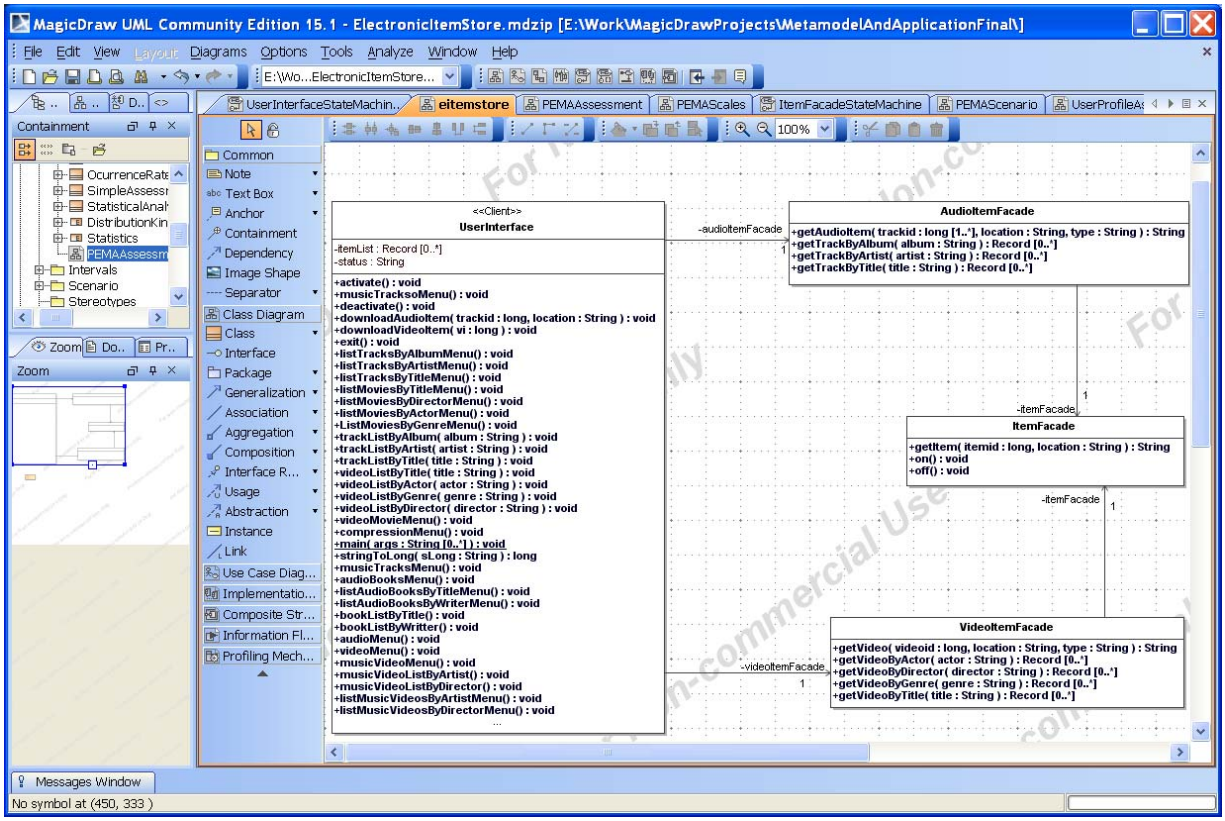


Figure 10.2.: Class diagram implemented in MagicDraw 15.1

The same UML modeling tool is used for the specification of measurement and metrics computation model. This is enabled by the specification of the PEMA Profile as the part of the project. A UML Profile is implemented in tools as a package with the stereotype ≪Profile≫. This package can be exported an imported for different projects. Therefore, it must be developed only once, and simply imported when needed for performance measurement and assessment.

The implementation of the profiling mechanism in ***MagicDraw*** version 15.1 is not completely according to the UML Superstructure specification. Namely, in stereotyped elements the association is not navigable from both sides. A stereotype can be reached when traversing an instrumented element. However, there is no navigability from the opposite side. It is not possible to reach to modeling element from the instance of the stereotype applied to it. For this reason, attribute ***instrumentedElementName*** is used in evaluation for specification of relation to instrumented element. The ***MagicDraw*** PEMA Profile context specification part implementation is presented in Figure 10.3.

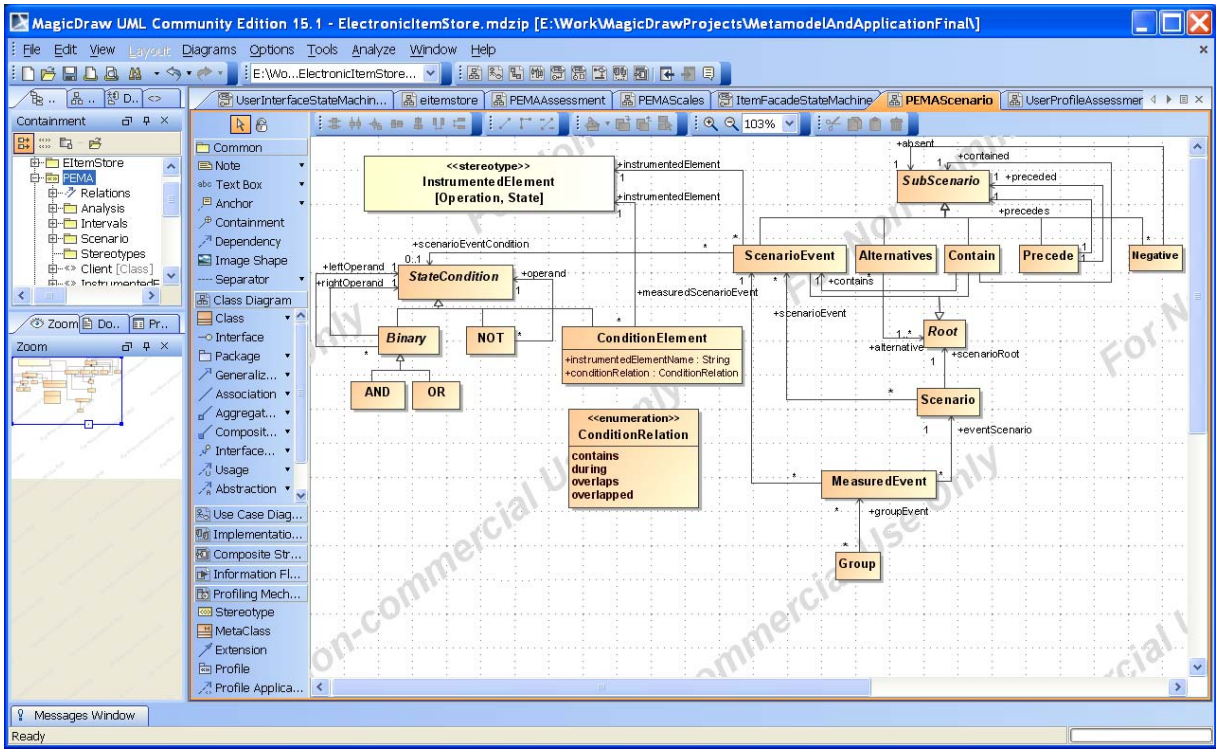


Figure 10.3.: PEMA Profile context and event metamodel implementation in MagicDraw 15.1

Transformations from model to code are implemented in the ***openArchitectureware*** environment and language suite for transformation implementations. It is implemented as an Eclipse plug in.

The language suite of ***openArchitectureware*** consists of four languages: ***Xtext*** for textual domain specific language definition, ***Xpand*** for template based transformation definition, ***Xtend*** for definition of function libraries used in transformations, and ***Check*** for constraints definition.

For the implementation here are used only **Xpand** and ***Xtend*** parts. Transformations defined in Chapter 8 and in Appendix E are implemented with ***Xpand*** language. Most of the embedded functions in templates defined in Appendix E, for example ***uName***, are implemented with ***Xtend*** language. Furthermore, functions like ***fEv***, ***lEv***, ***cNeg***, and so on, used in generating assessment code in Chapter 8 are also implemented in ***Xtend***. However, there are some functions like for example ***toFirstLower*** which are so common, that they

are provided as libraries. These functions are ready to be used in ***Xpand*** templates and are not implemented. The implementation of transformation for server interface code generation is presented in Figure 10.4.

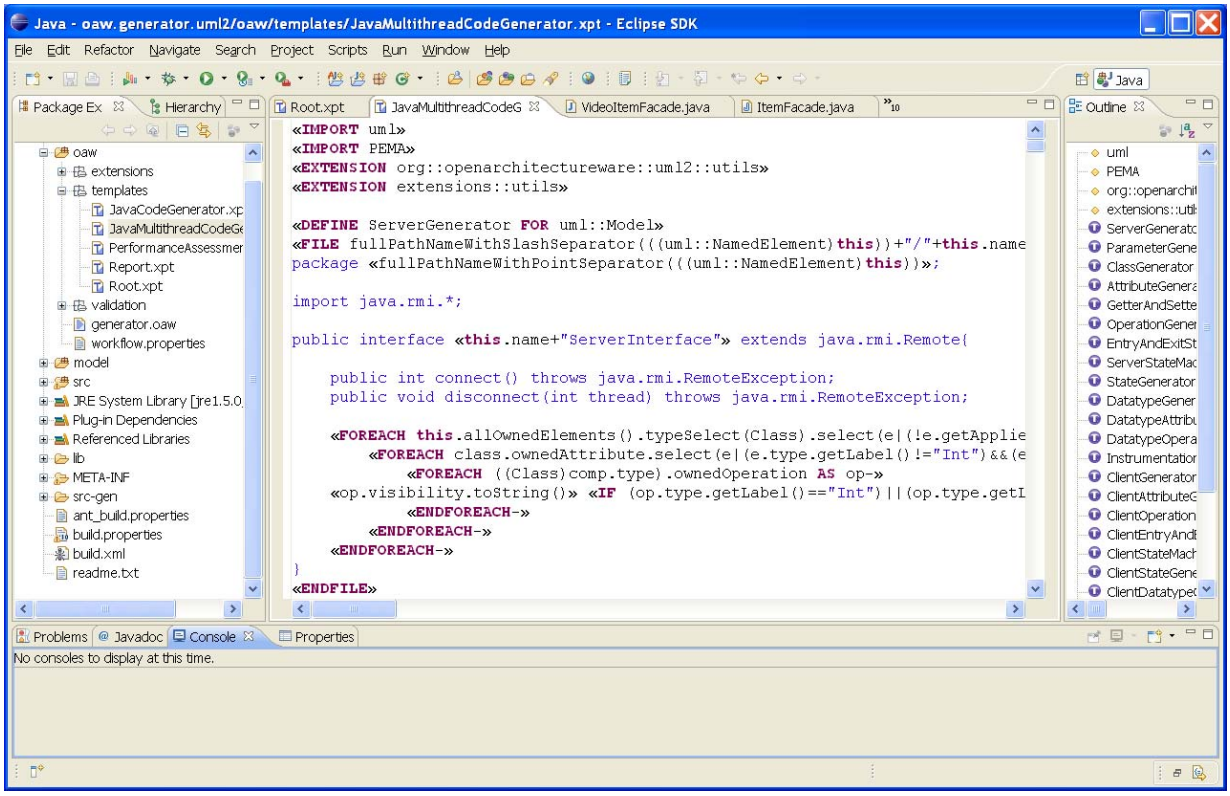


Figure 10.4.: Transformation for server interface generation in openArchitectureware 4.2

For database storage and metrics computation, ***MySQL*** database management system is used. It is an open source database management system with the possibility of extending the set of aggregate functions. In the following, an example of generated performance assessment code is given. The specification of the computation of the ***getItem*** method mean response time when the compression is in the state ***On*** and ***getItem*** is invoked from***getVideoItem*** method is given in Figure 10.5

For the metrics computation specified in Figure 10.5 is generated the following SQL query.

```
SELECT IntervalSet.PeriodStart AS PeriodStart,
IntervalSet.PeriodEnd AS PeriodEnd, AVG(GT.ETS-GT.STS) AS Value
FROM IntervalSet AS IntervalSet,
     (SELECT DISTINCT ET.ElementName, ET.STS, ET.ETS
      FROM ExecutionTrace ET,
           (SELECT SE.Session, SE.STS, SE.ETS
            FROM (SELECT *
                  FROM ExecutionTrace
                  WHERE ElementName='getItem') AS SE,
                 (SELECT Session, STS, ETS
                  FROM ExecutionTrace
```

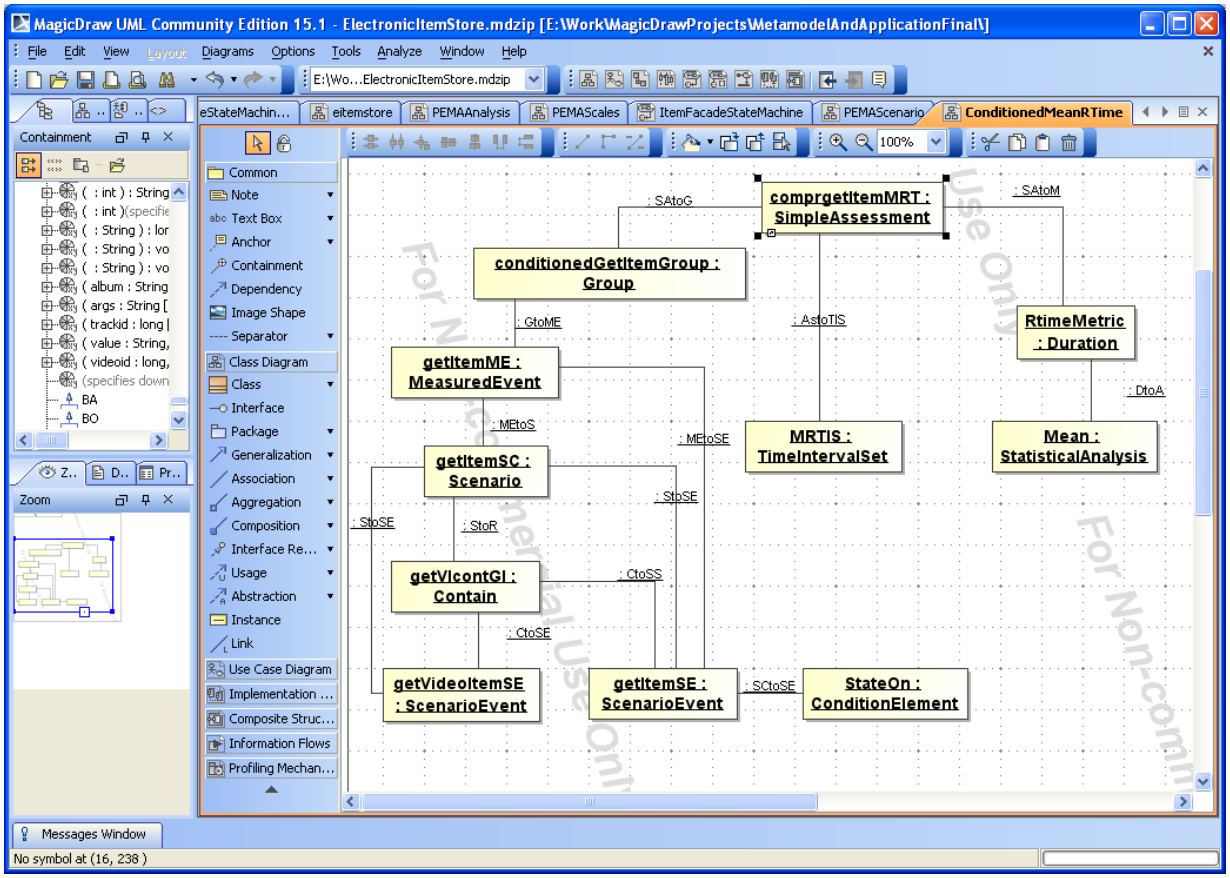


Figure 10.5.: An example of the metrics computation specification. In the figure is specified computation of mean response time when the ***getItem*** operation is invoked from the ***getVideoItem*** method and the compression is turned on

```
            WHERE ElementName='On') AS CE
        WHERE SE.Session=CE.Session AND
             (SE.STS > CE.STS) AND (SE.ETS < CE.ETS)) AS Element1,
      (SELECT Session, STS, ETS
       FROM ExecutionTrace
       WHERE ElementName='getAudioItem') AS Element2
  WHERE ET.Session=Element1.Session AND
       ET.Session=Element2.Session AND
       ET.STS=Element1.STS AND
       ((Element2.STS<Element1.STS) AND
       (Element2.ETS>=Element1.ETS))) AS GT
WHERE IntervalSet.PeriodStart<=GT.STS AND IntervalSet.PeriodEnd>GT.STS
GROUP BY IntervalSet.PeriodEnd
```

10.2.2. Data Collection and Storage Routine Duration

One of the major concerns of instrumentation is the overhead due to performed measurements. It is often characterized as the percentage of instrumented service response time spent for the measurements. The percentage of instrumented service response time depends on two factors: response time of non-instrumented service and the duration of the data collection and storage routine. The response time of the service depends on the design and implementation of the service and it is not the characteristic of measurement routine. For this reason, in this section shows the duration of the measurement routine.

Experiment configuration

For the experiment the following configuration of three nodes connected through 100Mbit switched local area network is used:

- The *server node* executed the server part of the electronic item management application on the Java 2 Standard Edition version 1.5. For the performance data storage interface is used JDBC MySQL Connector/J driver version 5.1.5. The application was running on the Intel Pentium 4 3.00 GHZ hyperthreaded processor (two virtual cores), 1GB of physical memory, and GNU/Linux 2.6.17.13.

- The *client node* runs concurrent client parts of the electronic item management application and MySQL 5.2 relational database management system. The concurrent client parts act as a workload generator, and MySQL is used for performance data storage and metrics computation. They both were running on Intel Pentium M 1.7GHz, 1.0GB of working memory, and Windows XP SP3. The client part of the electronic items management application used Java 2 Standard Edition version 1.5.

Results

The experiment was performed with the intention of showing the duration of the measurement routine for the range of 1-100 concurrent requests. It shows the routine duration in a middle size Internet application. The number of the concurrent requests was increased because the hypothesis of the experiment is that for the reason of the concurrent resource usage the duration of the data collection and storage routine increases with the increase of the number of concurrent requests.

For each number of concurrent requests, the experiment was repeated 10 times. Each repetition contained the complete restart of the server, in order to approximate the impact of the distribution of server software over working memory pages. For each experiment the training period was 2min. The training period is the period of operation of the system during which the initialization of used components take place. This period is not taken into account when computing metrics.

The intention of the experiment was to show the central tendency of the duration of the routine. This should serve as orientation to the performance analyst of how long approximately the routine lasts. For this reason, the mean of the duration routine was computed. Then, median for all 10 repetitions is computed in order to approximate the value of the data collection and storage routine. The obtained results are the following:

Concurrent requests	1	10	20	30	60	100
mean(median)	192ms	204 ms	229ms	260ms	289ms	327ms

Table 10.1.: The mean of the median for various concurrent invocations. The table shows the tendency of the performance data measurement and storage routine duration increase

The results showed that the hypothesis on the increase of data collection and storage routine duration was correct..Capture ratio, i.e., the percentage of total response time spent in measurement routines, depends on both, the business logic and the measurement placed measurement points, and, therefore, varies from system to system. It is possible, previously knowing the total response time of the system and the number of concurrent invocations, to use values from the Table 10.1 to compute the capture ratio of the system. In order to obtain the right values of the capture ratio, the resulting values from Table 10.1 for the appropriate number of concurrent invocations, should be multiplied by the number of the measurement points in the measured service and divided by the obtained value of response time of that service.

11. Comparative Analysis to Related Work

Integration of performance measurement and assessment in software development process has been a topic for long time in software engineering. This chapter gives the overview of the contribution of the ***MoDePeMART*** by comparing it to already existing approaches. In the following, first the related work is explained. Later, a comparative analysis of the related work with the ***MoDePeMART*** according to the concerns specified in Section 3.4 is given.

Tools for Model-driven Instrumentation for Monitoring. The first time the idea on integrating software models and instrumentation is introduced by Klar et al. [Klar et al., 1991]. The authors have developed a set of tools for a model-driven instrumentation. In their work are defined several sets of program models. These models are: functional program model, functional implementation model, monitoring model, and performance model. A program is not considered as a model in this approach.

In a functional program model the functional interdependence of activities is explicitly modeled. These activities are modeled without any implementation details. A functional program model focuses on a functional properties of an algorithm and defines the functional behavior of a program. This model is a prerequisite for a functional implementation model. A functional implementation model is a detailed model concerned with the implementation of the functional program model. Both of previously described models are some formalisms for concurrency description such as Petri-nets. The program code is now implemented in The ***Object C*** programming language according to the functional implementation model.

The two kinds of models left are dedicated to performance validation. A monitoring model is a subset of the functional implementation model. It defines functional interdependence of the implemented program on the desired level of abstraction. The monitoring model is closely related to the implemented program. The relation of a monitoring model and an implemented program is based on names. Each name which can be found in the monitoring model is the representation of an element in the implemented program. Furthermore, it defines the measurement points in the program. Names of functional entities in the monitoring model define which constructs in the program have to be instrumented. Finally, the performance model is created from the monitoring model and collected execution data. It is the monitoring model enriched with the realistic time attributes. A performance model is used for validation and is a prerequisite for predicting the performance. In this model, realistic time attributes are assigned according to measurements performed.

Tools for model driven instrumentation for monitoring provide ability of statistical time analysis, Furthermore, transparent instrumentation is also provided. Moreover, consistency of data types used in metrics computation and in data collection are facilitated with a separate language for their description. Finally, reduction of measurement points enabled with definition of the level of abstraction chosen for measurements. However there are several

shortcomings of this tool set. It provides the ability of only partial path characteristics analysis. It only supports analysis of whether the activities occur concurrently or sequentially. There is no support for specification of execution in branching and loops. For the same reason the isolation of a critical business task is only partially covered. There is no possibility of identifying from which branching alternative an invocation is made. Furthermore, the approach deals only with response times analysis. Throughput analysis of the system is not covered. Moreover, workload characterization is also not covered. Additinally, reactive execution context is not considered at all. Finally, the definition of measurement periods are not supported.

Program Monitoring and Measuring System (PMMS) is introduced by Liao and Cohen [Liao and Cohen, 1992]. It is a declarative language for specification of instrumentation and metrics computation. In PMMS metrics of interest are specified in a first order logic like language. According to the specification, instrumentation is automatically inserted, data collected during testing and metrics computed.

The declarative language developed in PMMS enables statistical analysis of program response times. Furthermore, it enables transparent instrumentation. Additionally, consistency of collected data types and data types used in metrics computation is provided with predefined data structures used in data collection. Finally, the measurement points minimization is provided by automatic addition of measurements in the program only at the places specified in metrics definition. However, there are several limitations. This language supports only response time analysis. Throughput analysis is not supported. Furthermore, there is no support for the definition of a period of measurement. Workload characterization can be only given with the number of requests of particular functions. Path characterization also has limitations. It is given only with the execution relations between method calls. For this reason, the isolation of critical business process task is also limited. Therefore, difference between invocations of a same method in two different alternatives cannot be recognized. Finally, specification of business tasks execution context is limited only on transformational execution.

Metrics Description Language (MDL) developed by Hollingsworth et al. [Hollingsworth et al., 1997] introduced as part of a set of tools for runtime parallel program instrumentation. Instrumentation consists of specification of measurement places, and specification of measurement routines. In **MDL** instrumentation can be placed at the procedure level. A specification of measurement places defines where measurement routines take place. Measurement routines can take place at an entry, an exit, and a procedure call. The body of measurement routines consists of simple control and data operations. It is not possible to define loops in the measurement routines body. Furthermore, timers and counters can be instantiated and used in computations.

In MDL, both, throughput and response times can be analyzed. Furthermore, similarly to **PMMS**, with separate language and automatic instrumentation are facilitated instrumentation transparency, consistency of data types used in measurement and metrics computations, and measurement points reduction. However, as in the **PMMS**, the granularity of specification is on procedure call. Therefore, it has the same drawbacks in isolation of business tasks and path characteristics observations as ***PMMS***. Furthermore, period of metrics computation is also the period of the complete experiment. Workload can be characterized with the number of requests and requests rate, but not with the pattern. Specification of business task execution context is limited to transformational execution context.

Application Response Measurement (ARM) [The Open Group, 1998] is an attempt for standardization of measurements in business applications. The idea behind the standard is to provide one technology neutral set of data types for measurements. These technology neutral data types can be implemented in different languages. Currently available implementations are for ***C/C++*** and ***Java***. The data types this standard defines are used in the application to perform measurements. The most important data type is the data type for representation of a transaction. In this transaction information about its duration is kept. **ARM** also defines the data types for additional description of transaction. Those data types are for counters, gauges, numeric identifiers, and strings. A counter is monotonically increasing non-negative value. A gauge value can go up and down. Numerical identifiers and strings are used for identification in transactions.

ARM standard provides a set of technology neutral-data types for measurements. Therefore, it solves the problem of consistency of data types used in measurements and in metrics computation. Furthermore, with the data type for definition of transaction it enables isolation of business tasks of interest. However, transactions are transformational systems, and therefore, this standard does not facilitate a specification of a business task reactive context. Additionally, it does not provide the support for statistical response time analysis. Moreover, parallel analysis of throughput and response times are also not facilitated. Workload, and path characteristics can not be assessed. Definition of validity period of metrics, instrumentation transparency and measurement points reduction were not intentions of this standard.

Aspect Oriented Programming (AOP) introduced by Kiczales et al. [Kiczales et al., 1997] is a programming approach which can be used for transparent software instrumentation. Transparent means that source code of the software functionality is not mixed with probes. Aspect oriented programming upgrades object oriented programming in such a way, that the crosscutting concerns, such as logging and security, are specified separately from the main functionality. Defined crosscutting concerns are than integrated in the main functionality with definition of ***join points***. Examples of the application of AOP for instrumentation are introduced by Marenholz et al. [Mahrenholz et al., 2002] and Debusman and Geihs [Debusmann and Geihs, 2003]. Debusmann and Geihs [Debusmann and Geihs, 2003] combine AOP with the ARM standard. Marenholz et al. [Mahrenholz et al., 2002] use ***AspectC++*** in operating system debugging, profiling/measurement, and runtime surveillance/monitoring. The similar ideas applied in programming are tried to be used in modeling. Such approach is called ***Aspect Oriented Modeling (AOM)***. AOM in the context of UML is introduced by Zhang et al. Zhang et al. [2007].

The main idea of ***AOP*** is transparent insertion of crosscutting concerns in software main functionality. Therefore, instrumentation transparency is one of the concerns this approach solves. Minimization of measurement points is achieved with adding probes only at the points of interest. However, critical business process isolation is limited because there is no possibility of context specification. Debusmann and Geihs [Debusmann and Geihs, 2003] have shown that **ARM** can be used with **AOP** to additionally provide transformational context specification and consistency between data types in measurements and metrics computation.

Interceptors have the same shortcomings as AOP. Additionally, concerns of isolation of critical business tasks and specification of business task execution context are not facilitated because of possibility of addition only in stubs and skeletons.

Transparent software layer. Performance of the system is can also observed be addi-

tion of a transparent software layer to the application. Such approach is used by Diaconescu et al. [Diaconescu et al., 2004] and Yeung et al. [Yeung et al., 2004]. Diaconescu et al. [Diaconescu et al., 2004] introduce an approach where, at the deployment time, a transparent proxy layer for the data collection of the execution data is automatically generated. Yeung et al. [Yeung et al., 2004] developed a virtual JVM called Verneer which is a Java program running on the top of the original JVM. The virtual JVM is a software layer which intercepts class loading, and fragments methods according to a fragmentation policy. Each method can be fragmented at the point of a method call, a method entry, basic blocks, and return. After the code has been fragmented, probes are added to these fragments.

Transparent software layer, as its name indicates, only makes instrumentation transparent, while all other concerns are not handled. Diaconescu et al. [Diaconescu et al., 2004] use Java Management Extensions (JMX) to provide additionally consistency of data types in measurements and performance analysis. Finally, they implement in software layer statistical analysis and throughput comparison. A limitation of the approach introduced by Diaconescu et al. [Diaconescu et al., 2004] are that their monitoring is at the level of component method invocations. Therefore, critical business tasks can only partially be specified. Furthermore, path characteristics are also limited to probability of executions but there is no support for the numbers of loops. Workload characterization, specification of metrics validity period is not supported. Finally, the complete application is instrumented and there is no measurement points reduction.

The comparative analysis of related work is summarized in Table 11.1.

Measurement and Assessment Concern / Approach	Response time statistical analysis	Throughput	Workload characteristics (number of requests, request rate, pattern)	Path characteristics (probability in branching, loop iteration numbers)	Isolation of critical business tasks	Specification of execution context	Metrics validity period specification	Instrumentation transparency	Measurement and metric computation data types consistency	Measurement points reduction
Klar et al. [1991]	+	-	---	--	o	+/-	-	+	+	+
Liao and Cohen [1992]	+	-	+/--	o/-	o	+/-	-	+	+	+
Hollingsworth et al. [1997]	+	+	++/-	o/-	o	+/-	-	+	+	+
The Open Group [1998]	-	-	---	--	o	+/-	-	-	+	-
Marenholz et al. [2002]	-	-	---	--	o	+/-	-	+	-	+
Debusman and Geihs [2003]	-	-	---	--	o	+/-	-	+	+	+
Young et al. [2004]	-	-	---	--	-	--	-	+	-	-
Diaconescu et al. [2004]	+	+	-	-	o	--	-	+	+	-
MoDePeMART	+	+	++/-	+/-	+	+	+	+(o)	+(o)	+

Table 11.1.: Comparative analysis with related work ((+) facilitated, (-) not facilitated, (o) partially facilitated)

MoDePeMART approach manages all of the previous mentioned concerns except for the loop iterations analysis. This is not addressed because of the specified purpose of this approach and the adopted understanding of qualitative and quantitative functional aspects Selic et al. [Selic et al., 1994]. ***MoDePeMART*** is used for measurement and assessment of performance of a service. A service is being defined with qualitative aspects of a functionality

of a system. Qualitative aspects are specifications of control flow and state of the system. The number of iterations in a loop are considered as a quantitative aspect of the system. Furthermore, some authors consider data characteristics as important. Data characteristics are type, number, and size of data. Here, they are also considered as quantitative functional aspects.

MoDePeMART only partially facilitates assessment of workload characteristic. Number and arrival rate are supported with the throughput and count aggregate functions. However, analysis of arrivals is not covered. Although the metamodel facilitates concerns of instrumentation transparency and keeping data types consistency in measurements, storage, and metrics computation, PEMA Profile in its current implementation does not support them completely. Because of the unavailability of action semantics concrete syntax in current UML modeling tools instrumentation transparency and keeping consistency between datatypes is not supported. By being able to specify both transformational and reactive context execution, the isolation of critical business task is fully supported. Furthermore, the metamodel enables computation of metrics for some day intervals, and herewith different validity periods of metrics can be observed. Minimization of measurement points is facilitated by relying on manual instrumentation, where the user would instrument only the events that he is interested in.

Part IV.

Conclusions and Outlook

12. Chapter Conclusions

This thesis proposes the ***MoDePeMART***: ***Mo***del ***D***riv***e***n ***Pe***rformance ***M***easurement and ***A***ssessment with ***R***elational ***T***races. The approach integrates measurement and assessment in the process of Model Driven Engineering. The main idea of the approach is to assess performance with modeling and not with the platform implementation constructs. Furthermore, the approach suggests declarative specification of performance metric of interest. The metrics are specified in a Domain Specific Modeling Language for performance measurements and assessment. Finally, it recommends usage of relational databases for data storage and performance metrics computation.

The main contribution of the thesis is a metamodel for performance measurement and assessment. The metamodel defines a Domain Specific Modeling Language for performance metrics specification. The metamodel is formally defined using the set theory and Libkin's algebra. Set theory is used for the definition of abstract syntax. Semantics is given with mappings to the Libkin's algebra. Herewith, the transformation is precisely defined.

The approach is evaluated by a proof of concept and comparative analysis with the related work. The proof of concept is implemented as a PeMA UML Profile, a UML profile for Performance Measurement and Assessment. PeMA profile is implemented for UML Class and State Diagrams. For the definition of functionality in Class and State Diagrams plain Java is used due to undefined action semantic's concrete syntax.

Transformation is evaluated by the implementation in **openArchitectureware**, a tool and a language suite for Model Driven Software Development. Transformation generates from a UML States and Class diagrams client server applications in Java with RMI. Furthermore, the transformation generates initialization and performance metrics computation SQL code for MySQL open source database management system.

Through the comparative analysis with the previous approaches for measurement and assessment, the major improvement of this approach is that the execution context can be completely specified. Execution context is the state of the system in which a certain event occurs, as well as the sequence of invocations which the event is part of. This improvement is achieved with facilitating instrumentation of all elements of transformational and reactive execution. For the reason of complete execution context specification, the benefit of this approach is that critical business tasks can also be completely isolated. Critical business tasks are business tasks of great importance. Meeting of performance goals of critical business tasks always has to be validated. Finally, the approach supports assessment of software systems performance metrics defined in MARTE profile.

The major shortcoming of the metamodel defined in this thesis is that it does not support characterization of parameters in method invocations. Data characterization can be important for verifying the appropriate usage of system. The approach does not enable measurement of workload patterns. Furthermore, it does not enable measurement of number of

iteration loops in system paths. However, this shortcoming is in correspondence with the assumptions of the approach. The approach assumes that the elements of the control flaw specification are qualitative functional characteristics of the software execution and that they are used for the defining services. The values of data are quantitative functional values of software execution and their impact is being statistically characterized.

In order to evaluate impact of measurements on the overall behavior a series of experiments has been performed. The series of experiments consisted of repeating measurements of performance data storage procedure response time. The experiment was performed for various number of concurrent invocations. The experiment showed that the duration of measurement and data storage routine increases with the number of requests and should be considered in final performance metrics computations.

12.1. Validity in Real Systems Use

In order to use ***MoDePeMART*** in real systems use, the systems to which the approach is applied have to satisfy some assumptions. These assumptions/limitations are explained in the following.

Measurement and assessment is possible only in systems with concurrency without intercommunication. In the execution model it is assumed that there are no concurrent executions which interfere. Moreover, the invoker of a scenario is not aware of concurrent execution. Such approaches are implemented in JEE Session Beans and in operating systems with one threaded processes Tanenbaum [2007].

In software systems, the synchronous communication is assumed. At the present time ***MoDePeMART*** supports only performance measurement and assessment for the systems which communicate synchronously. Synchronous communication is the one where the caller of an operation is blocked and waits until the callee returns a result before it continues its own execution [Object Management Group, 2007].

There is no support for specification of measurement and metrics computation of loopbacks. A loopback is when in a scenario execution control flow reenters a method whose body is already executing. The simplest form of a loopback is recursion.

Granularity of timing mechanism is large enough that execution of each instrumented element occurs in a different chronon. A chronon is the smallest unit of time supported by the discrete time model. Each storage timing mechanism stored the data at different granularity. The granularity is defined with the smallest time units supported by the timing mechanism, such as milliseconds or nanoseconds. The assumption of this approach is that each instrumented element execution with the same sequence identifier executes in different chronon. In this way, the requirement that the fields of execution trace form the unique key of the execution trace table.

The job flow [Smith and Williams, 2001] is assumed in the composite assessment of throughput. The job flow balance assumption is the assumption that a system is fast enough to handle all arrivals, and thus the competition rate or throughput equals the arrival rate. The validity of a job flow balance can be proved with a simple assessment, where the number of requests and the service mean response time would be observed.

Finally, the approach can be used only for verifying response time and throughput of services. Because the relational algebra is not Turing complete, isolating numbers of loops

for each iteration is not possible. Furthermore, the verification of the pattern of workload arrival is not possible. In order to analyze the pattern of arrival, some data mining techniques can be helpful. Finally, verifying the equivalence between assumptions on data is out of scope of this thesis.

13. Chapter Future Work

MoDePeMART introduces an idea of raising the abstraction level of measurement and assessment in two ways. First, measurement and assessment is specified in the terms of modeling and not in the terms of platform constructs. Second, it suggests a Domain Specific Modeling Language for metrics specification and computation. In order to go further in making performance measurement and assessment a completely engineering approach there are several promising directions for future work.

One could think of further improvement of the metamodel for measurement and assessment. With the comparative analysis discussed in Chapter 11 it is noticed metamodel presented in this thesis is lacking could be extended with the ability of data analysis. Furthermore, it is also reported that there is inability of analysis of numbers of iterations in loops. Finally, a possible improvement would be the recognition of workload arrival patterns. With previously mentioned improvements, a performance analyst or a developer who measures and assesses performance of the system could completely verify: that the workload in experiment corresponds to the workload used in predictions, that under a certain workload predictions with respect to path executions, including numbers of loop iterations and data characteristics are valid. Currently, this is not supported. As mentioned in Section 12.1 at the present moment, it is assumed that the predictions with respect to data characteristics of the workload and numbers of iterations in loops are valid.

Further improvement of the metamodel is the extension which would enable asynchronous communication measurement and assessment. In the metamodel, asynchronous communication specification would further expand the relations between scenario events. Currently, only invocation and sequential composition of methods are supported. In invocation composition, along with the method invocation, also the control flow is transfered to the invoked method, and invoking method waits for the invoked method to finish, in order to continue with the execution. Sequential composition is the execution of two methods one after another whiting a body of the same method. Scenario events in asynchronous communication are not composed with invocation and sequential relation. In asynchronous communication, one method sends a message to another method, but continues its execution and is not aware when the method that received the message stops. This relation is not invocation relation, and for this reason the relations that define scenarios must be extended. Moreover, besides the information about the correlation of method executions, it would be needed to store also information about the communications between threads. Accordingly, storage of execution data has to be redefined for facilitating measurement and assessment in systems with concurrent executions intercommunication. Additionally, if the systems that are communicated are distributed, metrics like latency of communication would also need to be supported.

Finally, the metamodel could be extended for measurement and assessment of resources utilization. Currently, the metamodel enables only the verification of timeliness. Instrumen-

tation of resources utilization could enable this language usable for performance debugging.

It could also be thought of specifying the instrumentation and assessment at the requirements specification level. An interesting idea would be to explore the possibility of specifying instrumentation and assessment at the requirements model level, and then using traceability links to specify propagate instrumentation and metrics computation all way to the final design model.

The ***PeMA*** UML profile and the transformation could also be improved. Current ***PeMA*** profile is developed only for the evaluation purposes and is not convenient for usage in specification of a measurement context. Furthermore, in for the development of the profile are used profile definition recommendations of Selic [Selic, 2006a]. These recommendations suggest to first define pure domain model and the profile elements second. Having the concepts in the thesis already developed in an OMG technology, this profile can be iteratively and incrementally improved to be useful for the main purpose. The main challenge here is to find the constructs in UML witch most closely match to the constructs in the metamodel. Usage of activity and sequence diagrams for specification of execution scenario of interest could be explored. These diagrams are usually used for control flow description/prescription. This qualifies them as a good basis for execution context specification. Furthermore, the instrumentation and performance measurement of models in other UML diagrams could be explored. Here, at least two problems arise. The first one is that in the UML there is no standard way for defining run-time semantics. The run-time semantics is typically an informal language description. In this sense the most appropriate application is to activity diagrams. Activity diagrams are one more established approach for the specification of transformational (algorithmic) behavior. Design of a UML profile for these diagrams would further prove the application for instrumentation transparency and data consistency. These two issues are currently only partially facilitated in the ***PeMA*** UML profile, as discussed in Chapter 11. However, in the adoption to the other UML diagrams the second problem arises. This problem is the definition of instrumented elements in the design language. In UML State and Class diagrams this mapping was one on one, states and operations are instrumented elements of the metamodel. However, in the UML Activity Diagrams this is not a case. For example, the ConditionedNode modeling element, presented in Figure 13.1

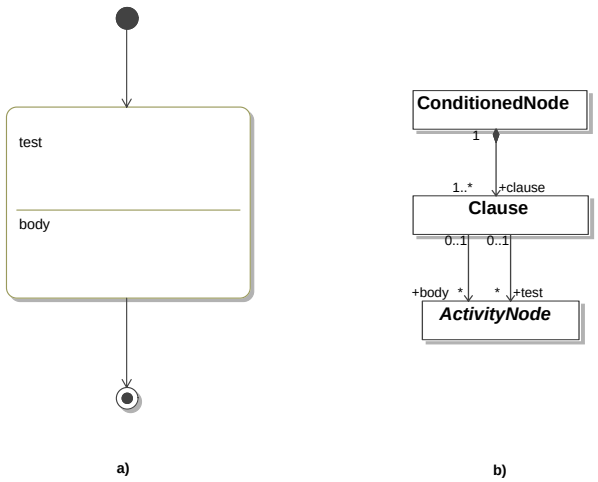


Figure 13.1.: A ***ConditionedNode*** (a) and its metamodel(b)

The body of a ***ConditionedNode*** needs to have a possibility of instrumentation. However, this is not possible with the simple stereotyping. The body is a metaatribute in the metamodel, and cannot be stereotyped at the M1 level. Therefore, some additional stereotypes and techniques for specification of the instrumented element need to be developed.

Mathematically supported automatic measurements point placements is one interesting direction of research. Currently, automatic instrumentation techniques instrument the complete application. This can result in a high instrumentation overload, and in invalid conclusions about the system performance. Improving automatic instrumentation based on the probabilities of execution specified in prediction would be a major improvement of the instrumentation process. ***Reduction of collected data*** with the automatic instrumentation would minimize requirements for database storage capacity. Furthermore, overhead induced with performance data collection routines would also be reduced.

Predictions of measurement overhead is also an interesting research direction. Measurements always influence the executing system. In electrical engineering, for example, the potential difference is measured with a voltmeter. In such measurements exist a notion of voltmeter's resistance. According to values of resistors in the electric circuits, and the value of voltmeter's resistance, one could decide whether the voltmeter is appropriate for measurements. Such a theory is still not explored in software performance engineering. The question which could this theory answer is whether a particular granularity of timing mechanism, the response time of performance data obtaining procedures, and the response time of procedures for storing them are appropriate for verification of software systems with some characteristics of response times and workload.

Finally, with given formal semantics of transformation in the Libkin's algebra one could consider ***query optimizations for performance metrics computation***. Furthermore, one could consider the predictions on resources requirements, such as time and storage space, according to characteristics of experiments, such as workload characteristics and number of measurement points.

Since the MDE approach for software development evolved from Computer Aided Software Engineering (CASE) approach, it is mostly concentrated on predictions of some properties of a system. Measurements and assessment of software languages, models, and systems characteristics are areas which are currently becoming the great attention in research community.

Bibliography

UML Profile for CORBA Specification, version 1.0, OMG document formal/02-04-01. web: http://www.omg.org/cgi-bin/apps/doc?formal/02-04-01.pdf, April 2002. Accessed May 2009.

MOF 2.0, OMG document ptc/04-10-15. web: http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-14.pdf, October 2004. Cited August 2006.

James F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of ACM*, 26(11):832–843, 1983. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/182.358434.

Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. doi: http://doi.acm.org/10.1145/258915.258924.

Martin Arlitt, Diwakar Krishnamurthy, and Jerry Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Transactions on Internet Technology*, 1(1):44–69, 2001. ISSN 1533-5399. doi: http://doi.acm.org/10.1145/383034.383036.

Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *Software, IEEE*, 20(5):36–41, Sept.-Oct. 2003. ISSN 0740-7459. doi: 10.1109/MS.2003.1231149.

Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. *Database Systems - Concepts, Languages and Architectures*. McGraw-Hill Book Company, 1999. ISBN 0-07-709500-6.

Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2004.9.

Frédéric Bertrand and Michel Augeraud. Bdl: A specialized language for per-object reactive control. *IEEE Transactions on Software Engineering*, 25(3):347–362, 1999. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/32.798324.

Grady Booch, Alan Brown, Sridhar Iyengar, James Rumbaugh, and Bran Selic. An MDA Manifesto. *In The MDA Journal: MDA Straight from the Masters*, November 2004.

Jacques Bouman, Jos Trienekens, and Mark Van der Zwan. Specification of Service Level Agreements, Clarifying Concepts on the Basis of Practical Research. In *STEP '99: Proceedings of the Software Technology and Engineering Practice*, page 169, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0328-4.

Marko Bošković and Wilhelm Hasselbring. Model driven performance measurement and assessment with modepemart. In *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 62–76, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04424-3. doi: http://dx.doi.org/10.1007/978-3-642-04425-0_6.

David A. Carr. Interaction Object Graphs: An Executable Graphical Notation for Specifying User Interfaces. In Philippe Palanque and Fabio Paternò, editors, *Formal Methods in Human-Computer Interaction*, pages 141–155. Springer, 1997.

James Clifford and Abdullah Uz Tansel. On An Algebra for Historical Relational Databases: Two Views. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, pages 247–265, New York, NY, USA, 1985. ACM. ISBN 0-89791-160-1. doi: http://doi.acm.org/10.1145/318898.318922.

Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of ACM*, 13(6):377–387, 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362384.362685.

Vittorio Cortellessa and Raffaela Mirandola. Deriving a Queueing Network based Performance Model from uml Diagrams. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 58–70, New York, NY, USA, 2000. ACM. ISBN 1-58113-195-X. doi: http://doi.acm.org/10.1145/350391.350406.

Markus Debusmann and Kurt Geihs. Efficient and Transparent Instrumentation of Application Components using an Aspect-oriented Approach. In *14th IFIP/IEEE Workshop on Distributed Systems: Operations and Management (DSOM 2003)*, volume 2867 of *Lecture Notes in Computer Science (LNCS)*, pages 209–220, Heidelberg, Germany, October 2003. Springer.

Ada Diaconescu, Adrian Mos, and John Murphey. Automatic Performance Management in Component Based Systems. In *Proceedings of the First International Conference on Autonomic Computing ICAC'04*, pages 214–221. IEEE Computer Society, 2004.

Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Englewood Cliffs, NJ, USA, 1976. ISBN 013215871X.

Dragan Djuric, Dragan Gasevic, and Vladan Devedzic. Ontology Modeling and MDA. *Journal of Object Technology*, 4(1):109–128, 2005. doi: http://www.jot.fm/issues/issue_2005_01/article3.

Curtis E. Dyreson and Richard Thomas Snodgrass. Timestamp semantics and representation. *Inf. Syst.*, 18(3):143–166, 1993. ISSN 0306-4379. doi: http://dx.doi.org/10.1016/0306-4379(93)90034-X.

Bruce Eckel. *Thinking in Java*. Prentice Hall Professional Technical Reference, 2002. ISBN 0131002872.

Jean-Marie Favre. Towards the basic theory to model model driven engineering. In *WiSME2004: the 3rd Workshop in Software Model Engineering*, 2004. web: http://www.metamodel.com/wisme-2004/present/22.pdf, Accessed May 2009.

Robert B. France, Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg. Model-driven development using uml 2.0: Promises and pitfalls. *Computer*, 39(2):59–66, 2006. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2006.65.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

Donald Gross and Carl M. Harris. *Fundamentals of Queuing Theory (2nd ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 1985. ISBN 0-471-89067-7.

William Grosso. *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001. ISBN 1565924525.

Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of trace exploration tools and techniques. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004.

David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/0167-6423(87)90035-9.

David Harel and Eran Gery. Executable Object Modeling with Statecharts. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 246–257, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7246-3.

Wilhelm Hasselbring and Ralf Reussner. Toward Trustworthy Software Systems. *Computer*, 39(4):91–92, 2006. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.2006.142.

Jeffrey K. Hollingsworth, Oscar Niam, Barton P. Miller, Zhichen Xu, Marcelo J. R. Goncalves, and Ling Zheng. MDL: A Language and a Compiler for Dynamic Program Instrumentation. In *PACT '97: Proceedings of the 1997 International Conference on Parallel Architectures and Compiler Techniques*, pages 201–213, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8090-3.

Richard Hubert. *Convergent Architecture: Building Model-Driven J2EE Systems with UML®*. John Willey and Sons Inc., New York, NY, USA, 2002. ISBN 0-201-72229-1.

Mika Katara, Reino Kurki-Suonio, and Tommi Mikkonen. On the Horizontal Dimension of Software Architecture in Formal Specifications of Reactive Systems. In *FOAL'04 Workshop on Foundations of Aspect-Oriented Languages*, pages 37–43, Lancaster, UK, March 2004.

Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley and Sons Inc., Hoboken, New Jersey, USA, 2008. ISBN 978-0-470-03666-2.

Stuart Kent. Model Driven Engineering. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 286–298, London, UK, 2002. Springer-Verlag. ISBN 3-540-43703-7.

Gregor Kiczales and Erik Hilsdale. Aspect-oriented Programming. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, page 313, New York, NY, USA, 2001. ACM. ISBN 1-58113-390-1. doi: http://doi.acm.org/10.1145/503209.503260.

Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management*. John Wiley & Sons, 2004. ISBN 0470845252.

Rainer Klar, Andreas Quick, and Franz 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*, pages 165–180. Elsevier Science Publisher B.V., February 1991.

Nils Klarlund and Michael I. Schwartzbach. A domain-specific language for regular sets of strings and trees. *IEEE Transactions on Software Engineering*, 25(3):378–386, 1999. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/32.798326.

Anthony Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of ACM*, 29(3):699–717, 1982. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322326.322332.

Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321197704.

Thomas Kühne. What is a model? In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

Thomas Kühne. Linguistic Classification: Two Dimensions of Modeling, INFWEST Seminar on Model Driven Software Engineering. Pirkkala, Tampere, Finland, August 2006, 2006.

Reino Kurki-Suonio. *A Practical Theory of Reactive Systems: Incremental Modeling of Dynamic Behaviors (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. ISBN 3540233423.

Yingsha Liao and Donald Cohen. A Specificational Approach to High Level Program Monitoring and Measuring. *IEEE Transactions on Software Engineering*, 18(11):969–978, 1992. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.177366.

Leonid Libkin. Expressive Power of SQL. *Theoretical Computer Science*, 296(3):379–404, 2003. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/S0304-3975(02)00736-3.

David J. Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-64105-5.

Torsten Lodderstedt. *Model Driven Security from UML Models to Access Control Architectures*. PhD thesis, Fakultät für Angewandte Wissenschaften der Albert-Ludvigs-Universität Freiburg, 2003.

Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schroeder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *ISORC '02: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 249–256, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1558-4. doi: http://doi.ieeecomputersociety.org/10.1109/ISORC.2002.1003713.

Markus Völter. openArchitectureWare, a Flexible Open Source Platform for Model-Driven Software Development. In *Eclipse Technology Exchange Workshop*. web: http://www.voelter.de/data/workshops/EtxMarkusVoelter.pdf, July 2006. Accessed May 2009.

Tom Mens, Pieter Van Gorp, Dániel Varró, and Gabor Karsai. Applying a model transformation taxonomy to graph transformation technology. *Electronic Notes in Theoretical Computer Science*, 152:143–159, 2006.

Joaquin Miller and Jishnu Mukerji. MDA Guide (Version 1.0) OMG document formal/03-06-01. web: http://www.omg.org/docs/omg/03-06-01.pdf, May 2003. Accessed May 2009.

David Neuendorf. Review of MagicDraw ® 11.5 Professional Edition. *Journal of Object Technology*, 5(7), 2006.

Object Management Group. UML®Profile for Schedulability, Performance, and Time Specification, OMG document formal/05-01-02. web: http://www.omg.org/cgi-bin/apps/doc?formal/05-01-02.pdf, January 2005a. Accessed May 2009.

Object Management Group. MOF 2.0/XMI Mapping Specification, v2.1, OMG document formal/05-09-01. web: http://www.omg.org/cgi-bin/apps/doc?formal/05-09-01.pdf, September 2005b. Accessed May 2009.

Object Management Group. UML 2.1.1. Specification: Superstructure, OMG document formal/2007-02-05. web: http://www.omg.org/cgi-bin/doc?formal/2007-02-05, February 2007. URL `http://www.omg.org/cgi-bin/doc?formal/2007-02-05`. Accessed May 2009.

Object Management Group. A UML®Profile for MARTE: Modeling and Analyzing Real-Time and Embedded systems, Beta 2, OMG Adopted Specification, OMG document ptc/2008-06-09. web: http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf, June 2008. Accessed May 2009.

Chris Raistrick, Paul Francis, and John Wright. *Model Driven Architecture with Executable UML®)*. Cambridge University Press, New York, NY, USA, 2004. ISBN 0521537711.

Matthias Rohr, André van Hoorn, Simon Giesecke, Jasminka Matevska, and Wilhelm Hasselbring. Trace-Context-Sensitive Performance Models from Monitoring Data of Software-intensive Systems. In Carl Lebsack, editor, *Proceedings of the Workshop on Tools, Infrastructures, and Methodologies for the Evaluation of Research Systems (TIMERS '08) at IEEE International Symposium on Performance Analysis of Systems and Software 2008 (ISPASS '08)*, pages 37–44, April 2008.

Miro Samek. *Practical statecharts in C/C++: Quantum programming for embedded systems*. CMP Publications, Inc., Manhasset, NY, USA, 2002. ISBN 1-57820-110-1.

Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2003.1231147.

Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2003.1231146.

Bran Selic. Domain-Specific Languages and UML Profiles, INFWEST Seminar on Model Driven Software Engineering. Pirkkala, Tampere, Finland, August 2006, 2006a.

Bran Selic. A Short Course on MDA Specifications, INFWEST Seminar on Model Driven Software Engineering. Pirkkala, Tampere, Finland, August 2006, 2006b.

Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time Object-oriented Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 1994. ISBN 0-471-59917-4.

Connie U. Smith and LLoyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley, Massachusetts, Boston, USA, 2001. ISBN 0-201-72229-1.

Richard Thomas Snodgrass. *Developing time-oriented database applications in SQL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-436-7.

Ian Sommerville. *Software Engineering (8th Edition)*. Pearson Addison Wesley, 2007. ISBN 03213137983.

Friedrich Steimann and Thomas Kühne. Coding for the Code. *Queue*, 3(10):44–51, 2005. ISSN 1542-7730. doi: http://doi.acm.org/10.1145/1113322.1113336.

Daniel E. Stevenson and Margaret M. Fleck. Programming Language Support for Digitized Images or, the Monsters in the Closet. In *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 21–21, Berkeley, CA, USA, 1997. USENIX Association.

Seyed M. M. Tahaghoghi and Hugh Williams. *Learning MySQL*. O'Reilly Media, Inc., 2006. ISBN 0596008643.

Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007. ISBN 9780136006633.

Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0. doi: http://doi.acm.org/10.1145/302405.302457.

The Open Group. Application Response Measurement (ARM). web: http://www.opengroup.org/tech/management/arm, 1998. Technical Standard, Version 2, Issue 4.1, Accessed May 2009.

Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons Ltd., Chichester, UK, 2002. ISBN 0-471-33341-7.

Peter Wegner. Why Interaction is More Powerful than Algorithms. *Commun. ACM*, 40(5): 80–91, 1997. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/253769.253801.

Roel J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*. Morgan Kaufmann Publishers, San Fransisco, CA, USA, 2003.

Kwok Yeung, Paul H. J. Kelly, and Sarah Bennett. Dynamic Instrumentation for Java Using a Virtual JVM. In *Performance Analysis and Grid Computing*, pages 175–187, Norwell, MA, USA, 2004. Kluwer Academic Publishers. ISBN 1-4020-7693-2.

Jing Zhang, Thomas Cottenier, Aswin van den Berg, and Jeff Gray. Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. *Journal of Object Technology*, 6(7), 2007.

A. The Case Study

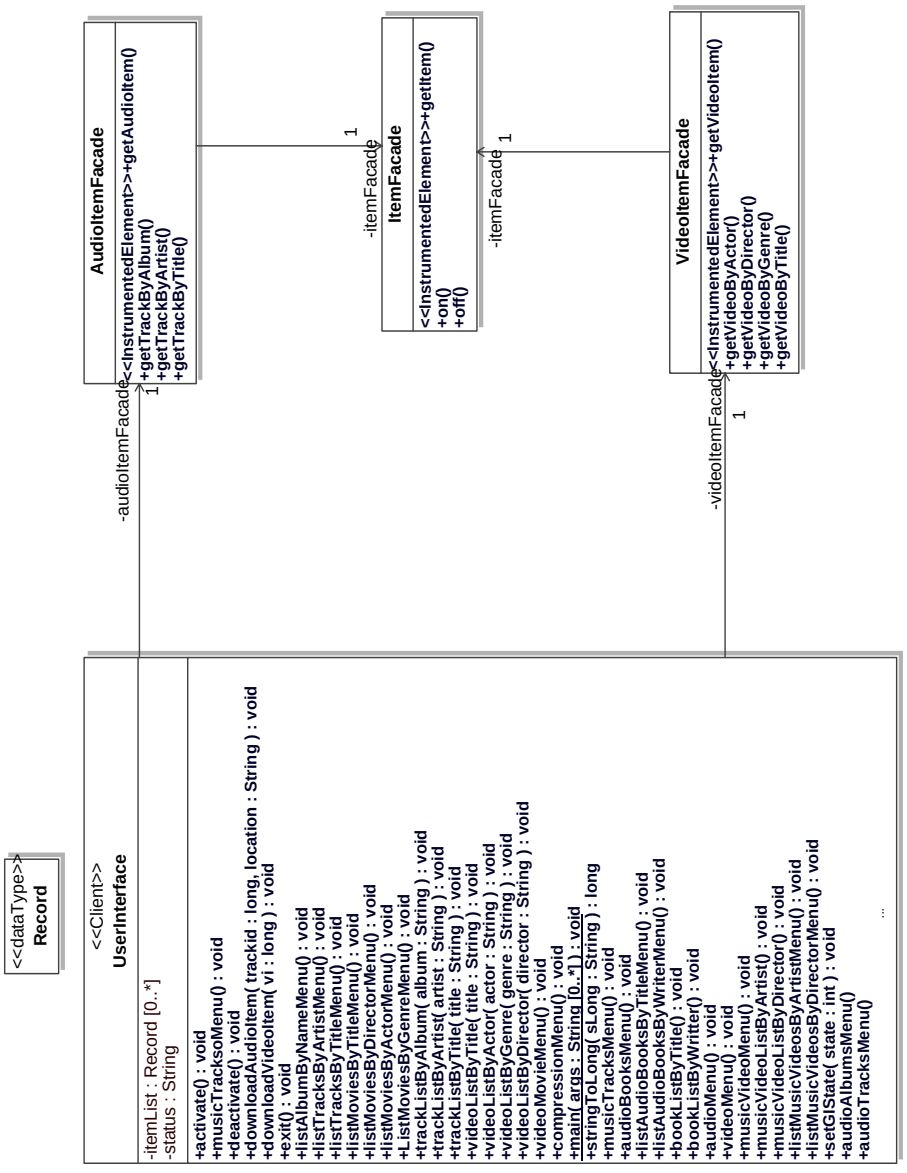


Figure A.1.: The complete UML Class Diagram of the case study. Complete signatures of the ***ItemFacade***, ***AudioItemFacade***, ***VideoItemFacade*** operations are given in Section 6.1

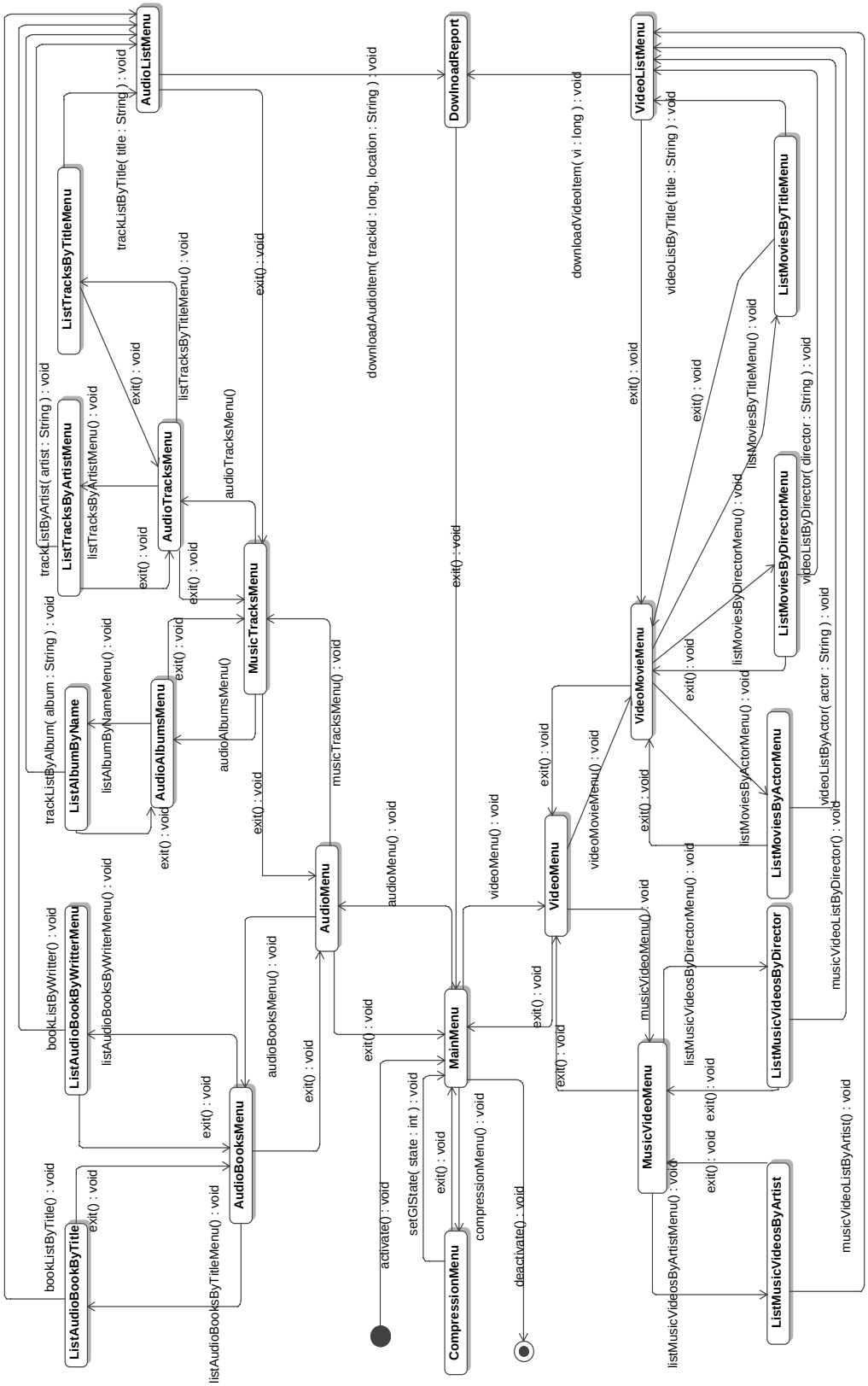


Figure A.2.: The complete UML State Diagram of ***UserInterface***

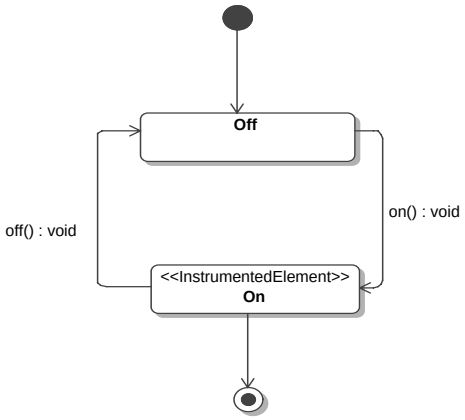
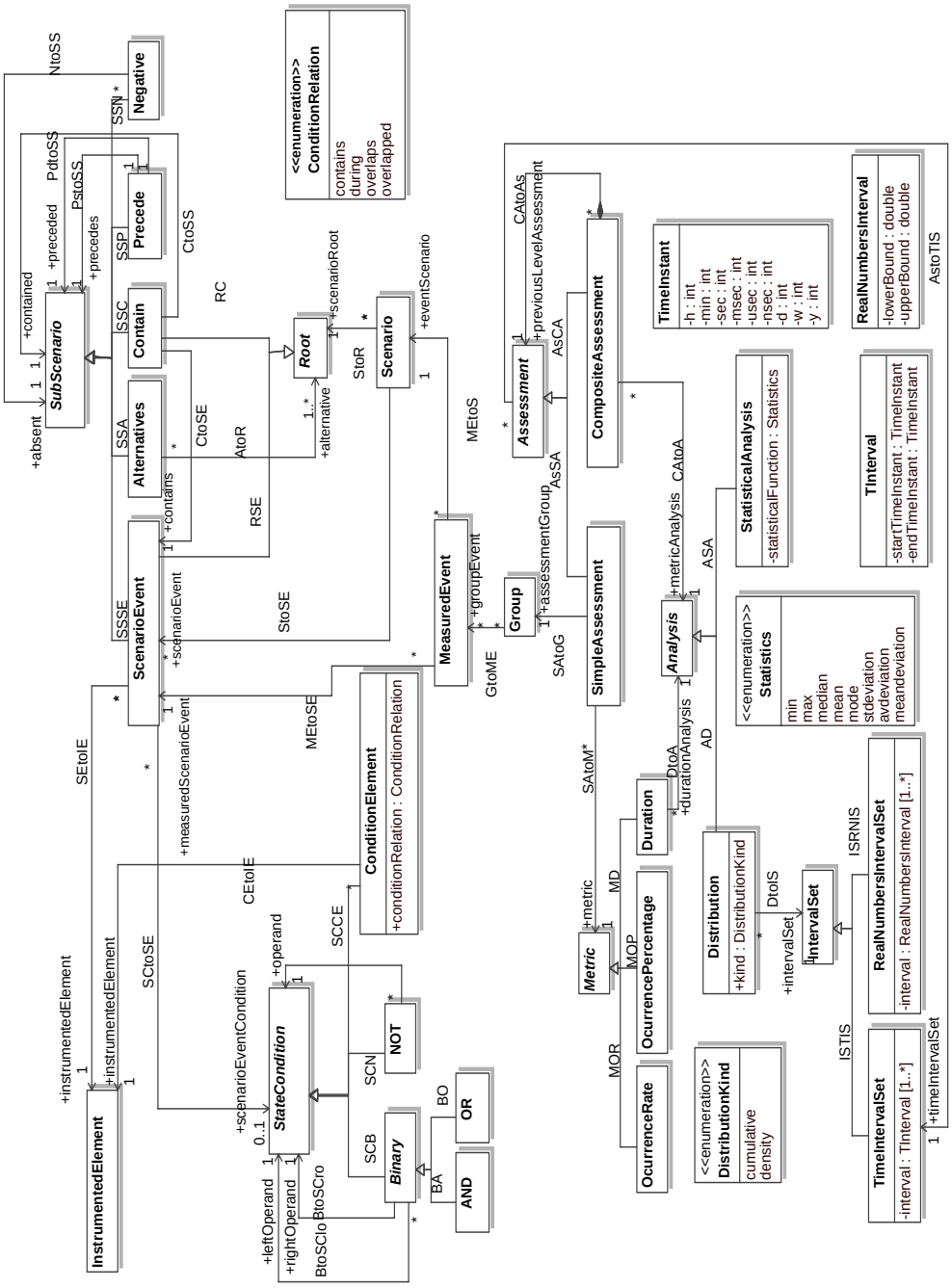


Figure A.3.: The UML State Diagram of ***ItemFacade***

B. A Linguistic Metamodel for Measurement And Assessment



C. Ordinal Functions

1. The formal definition of the ***altSubSc*** function.

The function ***altSubSc*** can be any function which uniquely maps an alternative sub scenario and a natural number smaller than the number of alternative sub scenarios in that alternative, to one of its alternative sub scenarios.
Formally, let $a \in Alternatives$. And let $alt \in AtoR$ such that $\forall(a_{alt}, r_{alt}) \in alt(a = a_{alt})$. And let $n_{alt} = cardalt$ and $n < n_{alt}$. Then ***altSubSc*** is an *injective* function defined as:

$$altSubSc(a,n) = \{ss : (ss \in SubScenario) \wedge \exists r \in Root, se \in ScenarioEvent, c \in Contain \nexists n_1 \in N((n \neq n_1) \wedge (n_1 < n_{alt}) \wedge (ss = altSubSc(a,n_1)) \wedge (((r,c) \in RC) \wedge ((ss,c) \in SSC)) \vee (((r,se) \in RSE) \wedge ((ss,se) \in SSSE)))\}$$

2. The formal definitions of the ***ordNumSEv*** and ***scOrdNum*** functions.

Functions ***ordNumSEv*** and ***scOrdNum*** are always defined in pair. Similarly to the function ***altSubSc***, the ***ordNumSEv*** can be any function which uniquely maps a scenario and a natural number smaller than the number of scenario events in that scenario, to one of its scenario event.
Formally, let $sc \in Scenario$. And let $sev \in StoSE$ such that $\forall(sc_{sev}, se_{sev}) \in sev(sc = sc_{sev})$. And let $n_{sev} = cardsev$ and $n < n_{sev}$. Then, the function ***ordNumSEv*** can be any function.

$$ordNumSEv(sc,n) = \{se : (se \in ScenarioEvent) \wedge \nexists n_1 \in N ((n \neq n_1) \wedge (n_1 < n_{sev}) \wedge (se = scOrdNum(sc,n_1)) \wedge ((sc,se) \in StoSE))$$

Now, the function ***scOrdNum*** is defined as follows. Let $sc \in Scenario$ and $se \in ScenarioEvent$. Then:

$$scOrdNum(sc,se) = \{n : (n \in N) \wedge se = ordNumSEv(sc,n)\}.$$

D. Temporal Data Types and Relations

Generally, there exist two kinds of happenings with respect to how long they last: events and intervals[Allen, 1983]. **Events** are happenings which, practically, do not have duration. They occur at one instant of time and end at that same moment. Opposite to events, **intervals** are happenings which start, last some time, and end. Events can be seen as intervals with infinitely small durations.

As the time model is usually adopted the discrete time model. According to this time model, the time is divided into small intervals of time called chronons. Chronon is the smallest duration an interval can last in the discrete time model. In the following definitions **a** and **b** are considered as intervals at a_s and b_s and ending at a_e and b_e respectively.

1. **a before b**, the relation which satisfies the condition $a_e < b_s$,
2. **a after b**, the relation which satisfies the condition $a_s > b_e$,
3. **a during b**, the relation which satisfies the condition $a_s > b_s \wedge a_e < b_e$,
4. **a contains b**, the relation which satisfies the condition $a_s < b_s \wedge a_e > b_e$,
5. **a overlaps b**, the relation which satisfies the condition $a_s < b_s \wedge a_e < b_e$,
6. **a overlapped-by b**, the relation which satisfies the condition $a_s > b_s \wedge a_e > b_e$,
7. **a meets b**, the relation which satisfies the condition that the interval **b** starts in the chronon after the end of the interval **b**, or mathematically $a_e + 1 = b_s$,
8. **a meet-by b**, the relation which satisfies the condition $a_s = b_e + 1$,
9. **a starts b**, the relation which satisfies the condition $a_s = b_s \wedge a_e < b_e$,
10. **a started-by b**, the relation which satisfies the condition $a_s = b_s \wedge a_e > b_e$,
11. **a finishes b**, the relation which satisfies the condition $a_s > b_s \wedge a_e = b_e$,
12. **a finished-by b**, the relation which satisfies the condition $a_s < b_s \wedge a_e = b_e$,
13. **a equals b**, the relation which satisfies the condition $a_s = b_s \wedge a_e = b_e$.

E. The Transformation to Client Server Applications with Java RMI

Transformation of an instrumented client-server application model to the target platform is divided into three sub transformations: for performance measurement code generation, for server code generation, and for client code generation. These transformation are defined in Sections E.2, E.3, and E.4, respectively. They are defined with the notation described in Section E.1.

E.1. Transformation Notation

Notation used for transformation definition is based on the notation specified by Lodderstedt [2003]. This notation defines a transformation as a set of transformation rules. Each rule takes UML model elements and produces a text output for them. The structure of a transformation rule is

$$< parameter\ declaration > \mapsto_{<rule\ name>} < output >$$

.

The rule name is specified in the index of the arrow. A transformation rule parameters are declared at the left side. They can be a model or a collection of modeling elements. The declaration of parameters defines both parameter types and parameter names. An example of parameter definition is ***Class(c)***. In this case the parameter name is ***c*** and it is of type ***Class***. When several parameters are defined, they are separated with commas. Parameter declaration also serves as a precondition that must be satisfied in order to apply the rule.

On the right side or in the next line of the arrow is the text output. The text consists of static text and embedded instructions. These instructions are surrounded with [[]] symbols. Embedded instructions may execute a function or apply a transformation rule. In both cases the output has to be text. Generated text is embedded in the surrounding static text at the place where the instruction is situated.

An example of the syntax for function call is [[name(c)]]. This function writes the name of the class c at the point in the text where the function call resides. Parameters of the embedded function definition are either parameters or variables of enclosing transformation rule.

An embedded rule application first define a set of elements and then executes specified transformation rule for each of defined elements. The syntax of the transformation rule is as follows

$$[\forall < variable > \in < set >]\ apply < listofrules >$$

. An example of such a rule application is

$$[\![\forall oA \in ownedAttribute(c)\ \text{apply}\ \mapsto_{type} (oa)]\!].$$

This rule applies $\mapsto_{type}$ rule for each element of the ***ownedAttribute*** function resulting set.

Following the previously defined transformation language, the transformation for code generation is defined as

$Model(m) \mapsto_{javarmiclientserver}$
$[\![$ apply $\mapsto_{performanceDataStorage}(m)]\!]$
$[\![$ apply $\mapsto_{javaRMIServerCode}(m)]\!]$
$[\![$ apply $\mapsto_{javaRMIClientCode}(m)]\!]$.

These transformations are explained in the following three sections.

E.2. Transformation to Performance Measurement Code

Performance measurement code consists of probes and globally visible unique methods for performance data storage. Probes are inserted in the application logic implementation code. For that reason, probes are generated with transformation to application code. Application code generation is described in the next section. This section describes the design of the transformation to data storage code. Generation of code for data storage is defined with the following transformation:

$Model(m) \mapsto_{PerformanceDataStorage}$
public class PerformanceDataStorage{
    ...
    private Statement statement;
    ...
    private static PerformanceDataStorage instance=null;

    $[\![\forall ie \in ownedElements(m)$ apply $\mapsto_{STS}$(ie)$]\!]$

    protected PerformanceDataStorage(){
        ...
    }

    public static PerformanceDataStorage instance(){
        ...
    }

    $[\![\forall ie \in ownedElements(m)$ apply $\mapsto_{setSTS}$(ie)$]\!]$

    $[\![\forall ie \in ownedElements(m)$ apply $\mapsto_{storeData}$(ie)$]\!]$

    public void incrementSize(int session){
        ...

```
    }

    public void finalize(){
        ...
    }
}.
```

Global visibility and singularity requirement of performance data storage code is realized with the implementation of ***Singleton*** [Gamma et al., 1995] pattern. The ***Singleton*** is in this class realized with private static attribute ***instance***, static method with the same name (***instance***) and public visibility, and the constructor with visibility ***protected***. The unique instance of this class is kept in the ***instance*** attribute. The instance can be reached only with ***instance*** method invocation. This method is a service of the class and can be globally accessed. Further instantiation of the class is restricted with the private constructor. Beside constraining instantiation with its ***protected*** visibility, constructor also makes a connection with the database for performance data storage. The connection to the database is ended when ***finalize*** method is invoked. Because of brevity, code of these methods is not shown.

For each instrumented element execution have to be stored the start and the end time stamp of its execution. To achieve that, for each instrumented element a unique attribute and two unique methods are defined.

The unique attribute of an instrumented element is used for temporarily storing the start time stamp of its execution. The start time stamp is stored just before the instrumented element's execution. It is stored by invocation of the unique setter method. The attribute and the setter method are generated with $\mapsto_{STS}$ and $\mapsto_{setSTS}$ templates, respectively. Function ***uName(e)***, used in these mappings, computes a unique string for the element e.

$InstrumentedElement(ie) \mapsto_{STS}$

```
private long ⟦uName(ie)⟧STS[];
```

$InstrumentedElement(ie) \mapsto_{setSTS}$

```
public void ⟦uName(ie)⟧SetSTS(int session, long sts){
        try{
            ⟦uName(ie)⟧STS[session]= sts;
        }
        catch (Exception ex){
            ex.printStackTrace();
        }
    }
```

Templates $\mapsto_{STS}$ and $\mapsto_{setSTS}$ are invoked for each element of the model. Each element of the model is the result of the function ***ownedElement(m)***, where ***m*** is the model. However, when executing the transformation rules, the code is generated only for those in set ***InstrumentedElement***. Elements in ***InstrumentedElement*** set are elements with ≪InstrumentedElement≫ stereotype.

The server application allows running multiple sessions concurrently. Accordingly, instrumented modeling element execution can occur in several concurrent sessions. For this reason

the attribute for storing start time stamp is a vector with one element for each application session identifier. Consequently, session identification is the function parameter along with the value of the time stamp. With session identification the setter method recognizes for that session reserved element in the vector.

The second unique method defined for an instrumented element is the method invoked when the end of the instrumented element occurs. This method is defined with $\mapsto_{storeData}$ template.

$InstrumentedElement(ie) \mapsto_{storeData}$
```
public void ⟦uName(ie)⟧StoreData(int session, long ets){
        try{
           statement.executeUpdate("INSERT INTO ExecutionTrace (Element, Session,
STS, ETS) VALUES ("'⟦uName(ie)⟧' " +", "+session+", "+(⟦uName(ie)⟧STS[session])+", "
+(ets)+")");
        }
        catch (Exception ex){
           ex.printStackTrace();
        }
    }
```

Parameters of performance data storage methods are session identification and end time stamp. These methods store in the database, an instrumented element identifier, a session identifier, a start time stamp and an end time stamp. The storage in the database is performed with the invocation of ***statement*** attribute ***executeUpdate***. Attribute ***statement*** is of type JDBC (Java Database Connectivity) *Statement*. JDBC ***Statement*** instances used for execution of SQL in a database. The SQL command which is executed is defined as a ***String*** parameter of ***executeUpdate*** method.

Method ***incrementSize*** extends the length of vectors for instrumented elements' start time stamp execution. This method is implementation of ***Lazy Acquisition*** pattern [Kircher and Jain, 2004]. ***Lazy Acquisition*** pattern, in order to optimize resource use, defers resource acquisitions to the last possible time in use. In this case it is the number of elements in the vectors for temporary start time stamps holding. Number of elements in these vectors corresponds to the number of sessions which concurrently serve users' requests. When the number of sessions increases also the number of elements in vectors is increased.

E.3. Transformation to Server Code

Server code consists of interface code, application logic code, and datatypes code. Therefore, the transformation to server code is defined with
$Model(m) \mapsto_{javaRMIServerCode}$
⟦ apply $\mapsto_{javaRMIServerInterfaceCode}$(m)⟧
⟦ apply $\mapsto_{javaRMIApplicationLogicCode}$(m)⟧
⟦ apply $\mapsto_{javaRMIServerDataTypesCode}$(m)⟧.

E.3.1. Server Interface Code

According to Java RMI specification a server has to be implemented with a publicly available interface which contains services this server provides. Furthermore, this interface has to be implemented in a class which extends specific library classes. Finally, to enable multi-session execution specified in Section 6.2 the server session class has to be defined. Consequently, the server code transformation is specified in the following way
$Model(m) \mapsto_{javaRMIServerInterfaceCode}$
$[\![apply \mapsto_{serverInterface}(\mathrm{m})]\!]$
$[\![apply \mapsto_{serverImplementation}(\mathrm{m})]\!]$
$[\![apply \mapsto_{serverSessionImplementation}(\mathrm{m})]\!]$.

The server interface, is generated with the following transformation:

$Model(m) \mapsto_{serverInterface}$
import java.rmi.*;

public interface ⟦name(m)⟧ServerInterface extends java.rmi.Remote{

public synchronized int connect() throws java.rmi.RemoteException;
public synchronized void disconnect(int session) throws java.rmi.RemoteException;

$[\![\ \forall client \in ownedElements(m)$ apply $\mapsto_{sSIntMethodNames}\ (client)]\!]$

}.

The server interface extends **java.rmi.Remote** interface, according to Java RMI specification. This interface also contains connecting and disconnecting operation declarations. Operation for connecting, ***connect***, is used when a client registers for using services of the server. It returns the identifier of the session dedicated to carrying out requests of that client. Client sends this identifier with each request he makes. Contrary, to ***connect***, operation ***disconnect***, releases the session dedicated to that client. It is invoked when the client does not intend to use services of the server any more.

Beside methods for connecting and disconnecting, server interface contains operations of application logic used by a client class. These operations are generated with $\mapsto_{sSIntMethodNames}$ mapping.

$Client(cli) \mapsto_{sSIntMethodNames}$

$[\![\forall oP \in ownedAttribute(cli)(type(oP) \notin DataType)$ apply $\mapsto_{sSIntPropertyTypeMethodNames}$ $(oP)]\!]$

$Property(p) \mapsto_{sSIntPropertyTypeMethodNames}$

$[\![\forall oO \in ownedOperation(type(p))$ apply $\mapsto_{sSIntOperation}\ (oO)]\!]$

$Operation(o) \mapsto_{sSIntOperation}$

⟦toString(visibility)⟧ syncrhonized ⟦$t = type(o)$ apply $\mapsto_{type}$ (t)⟧ ⟦uName(o)⟧
(int clientSession, ⟦$\forall oP \in ownedParameter(o)$ apply $\mapsto_{param}$ (oP)⟧) throws
java.rmi.RemoteException;

Mapping $\mapsto_{sSIntMethodName}$ is invoked in $\mapsto_{serverInterface}$ transformation. It is executed for each client in the model. Operations defined in the interface are operations of classes which do not have ≪***DataType***≫ stereotype, and there is at least one client's attribute of their type. They are generated with the execution of $\mapsto_{sSIntPropertyTypeMethodNames}$ mapping for each attribute of the client. All attributes of the client are contained in the set which is the result of ***ownedAttribute*** function. The name of function ***ownedAttribute*** is according to a convention used in the design of transformations to code. The convention specifies the rule according to which when a function has a name of an attribute of a metaclass (M2 layer), the result of this function is the value of this attribute in the model (M1 layer). In this example, function ***ownedAttribute*** is a M2 layer attribute. At M1 layer value of this attribute is a set of attributes specified in an M1 class. Therefore, the result of this function is the set of attributes defined in the parameter class.

In the transformation $\mapsto_{sSIntPropertyTypeMethodNames}$ for each operation of the type of an attribute, template $\mapsto_{sSIntOperation}$ is executed, and the declaration for that function is generated. In our example, server interface operations correspond to operations of ***AudioItemFacade*** and ***VideoItemFacade***. Each operation in the interface has the same visibility and return value data type as the corresponding UML class operation. Visibility and return value are generated with converting visibility to string and writing the name of the operation type with embedded function ***toString*** and mapping $\mapsto_{type}$, respectively. Furthermore, the function in the interface is entitled with the result of ***uName*** function of corresponding operation. Additionally, it has the set corresponding operation parameters extended with the session identification. Session identification is of type ***int*** and it is used for identifying client. Java operation parameters corresponding to UML operation parameters are generated with function $\mapsto_{param}$. This function generates the data type and name of Java operation parameter for each UML class parameter. Finally, each operation in the interface throw ***java.rmi.RemoteException***, according to Java RMI specification.

Server code consists of RMI server interface implementation code and application logic implementation code. RMI server interface implementation class is defined with the next transformation:

$Model(m) \mapsto_{serverImplementation}$

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import java.net.*;

public class ⟦name(m)⟧ServerImplementation extends UnicastRemoteObject implements

⟦name(m)⟧ServerInterface{

private java.util.Vector serverSessionPool=null;
...
public ⟦name(m)⟧ServerImplementation() throws java.rmi.RemoteException{
...
}

public int connect() throws java.rmi.RemoteException{
...
}

public void disconnect(int session) throws java.rmi.RemoteException{
...
}

public static void main(String[] args){
System.setSecurityManager(new RMISecurityManager());
try {
LocateRegistry.createRegistry(...);
Naming.rebind(...);
}
catch (Exception ex){
ex.printStackTrace();
}
}

⟦$\forall client \in ownedElement(m)$ apply $\mapsto_{sImplOperations}$(client)⟧

}
$Client(cli) \mapsto_{sImplOperations}$

⟦$\forall oP \in ownedAttribute(cli)(type(oP) \notin DataType)$ apply $\mapsto_{sImplPropertyTypeOperations}$ (oP)⟧

$Property(p) \mapsto_{sImplPropertyTypeOperations}$

⟦$\forall oO \in ownedOperation(type(p))$ apply $\mapsto_{sImplOperationDef}$ (oO)⟧

$Operation(o) \mapsto_{sImplOperationDef}$

⟦$toString(visibility)$⟧ ⟦$t = type(o)$ apply $\mapsto_{type}$(t)⟧ ⟦uName(o)⟧(int clientSession, ⟦$\forall oP \in ownedParameter(o)$ apply $\mapsto_{param}$(oP)⟧) throws java.rmi.RemoteException{

ServerSession ss=

(ServerSession)serverSessionPool.elementAt(clientSession);

$[\![return(o)]\!]$ ss.$[\![uName(o)]\!]$($[\![\forall p \in parameter(o)$ apply $\mapsto_{paramName}$(p)$]\!]$);
}.

This class implements the previously defined RMI interface and, according to Java RMI specification has to extend ***java.rmi.server.UnicastRemoteObject*** class. In this class all operations of the server interface have their implementations. Additionally, public static function ***main*** is defined. Function ***main*** starts the server, and registers it to ***Java RMI registr***y. ***Java RMI registry***, makes the interface accessible to clients.

At runtime all clients share the same object of the server interface implementation class. Concurrent requests are handled through separate threads of Java RMI platform. Threads are not visible to the programmer. Nevertheless, in the case that this server object has attributes, they are shared by all clients. This is not kind of execution which is assumed in Section 6.2.

In order to provide multiple concurrent client serving without data sharing, which is the platform execution model specified in Section 6.2, the server implementation acts during runtime like a pool of ***ServerSession*** class instances. The set of server session instances is contained in ***serverSessionPool*** attribute. When a client connects to the server, one ***ServerSession*** class instance from the pool is acquired to serve to requests of that client. Contrary, when it disconnects, the serving instance which is returned to the pool. For processing clients' requests, the server instance only has to recognize to which ***ServerSession*** instance it should forward the invocation. Accordingly, each server interface method, except for ***connect***, ***disconnect*** and ***main***, have their corresponding method in the ***ServerSession*** class. The ***ServerSession*** instance to which the invocation is forwarded is specified with the Session identification contained in each client request. The code for forwarding invocations is generated with $\mapsto_{sImplOperationDef}$ mapping. In $\mapsto_{sImplOperationDef}$ transformation $\mapsto_{paramName}$ writes only names of argument parameter. Class ***ServerSession*** is defined with the following transformation:

$Model(m) \mapsto_{serverSessionImplementation}$

public class ServerSession{

$[\![\forall client \in ownedElement(m)$ apply $\mapsto_{sSesAttribute}$(client)$]\!]$

$[\![\forall client \in ownedElement(m)$ apply $\mapsto_{sSesOperation}$(client)$]\!]$

public ServerSession(int i)
...
}
}
$Client(cli) \mapsto_{sSesAttribute}$
$[\![\forall oA \in ownedAttribute(cli)$ apply $\mapsto_{sSesAttributeDef}(oA)]\!]$

$Property(p) \mapsto_{sSesAttributeDef}$
$[\![toString(visibility(o))]\!]$ $[\![t = type(p)$ apply $\mapsto_{type}$(t)$]\!]$ $[\![$uName(p)$]\!]$;

$Client(cli) \mapsto_{sSesOperation}$
$[\![\forall oA \in ownedAttribute(cli)$ apply $\mapsto_{sSesPOperationDef}(oA)]\!]$

$Property(p) \mapsto_{sSesPOperationDef}$
$[\![\forall oA \in ownedOperation(class(p))$ apply $\mapsto_{sSesOperationDef}(p, oA)]\!]$

$Property(p), Operation(o) \mapsto_{sSesOperationDef}$
$[\![toString(visibility(o))]\!]$ $[\![t = type(o)$ apply $\mapsto_{type}$(t)$]\!]$ $[\![$uName(o)$]\!]$(int clientSession, $[\![\forall oP \in ownedParameter(o)$ apply $\mapsto_{param}$(oP)$]\!]$) throws java.rmi.RemoteException{

$[\![return(o)]\!]$ $[\![uName(p)]\!]$.$[\![name(o)]\!]$($[\![\forall p \in parameter(o)$ apply $\mapsto_{paramName}$(p)$]\!]$);
}.

ServerSession class has a corresponding operation for each operation of the server interface except for ***connect***, ***disconnect***, and ***main***. Operations of ***ServerSession*** class are generated with invocation of $\mapsto_{sSesOperation}$ transformation function for each client. ***ServerSession*** class operations only forward invocation to appropriate operation of a suitable application logic implementation class instance.

Attributes of ***ServerSession*** are application logic implementation classes instances. Class ***ServerSession*** contains, a corresponding attribute for each client attribute whose type is a non stereotype containing class. In our example, ***ServerSession*** class has corresponding attributes to ***audioItemFacade*** and ***videoItemFacade UserInterface*** class attributes. ***ServerSession*** attributes are generated with $\mapsto_{sSesAttribute}$ transformation invocation for each client.

Having the server object executing like a pool of sessions is not enough for providing the assumed execution model. Each server session has to have a set of application logic implementation class instances, dedicated to one client. This requirement is provided with specific transformations to application logic code.

E.3.2. Application Logic Code

To provide the execution model specified in Section 6.2 application logic design UML class, which is a UML class without stereotypes, is transformed to code of two Java classes: a class which implements the application logic defined in the UML class, and a Java class which serves as a pool of instances. Accordingly

$Model(m) \mapsto_{javaRMIApplicationLogicCode}$
$[\![\forall oE \in ownedElements(m)(oE \notin Client(m) \wedge oE \notin DataType(m))$ apply $\mapsto_{classPool}$(oE)$]\!]$
$[\![\forall oE \in ownedElements(m)(oE \notin Client(m) \wedge oE \notin DataType(m))$ apply $\mapsto_{class}$(oE)$]\!]$

The class which serves as a pool of instances is defined with the following transformation:

$Class(c) \mapsto_{classPool}$

```
public class ⟦name(c)⟧Pool{

    private java.util.Vector ⟦toFirstLower(name(c))⟧;

    private static ⟦name(c)⟧Pool instance=null;

    protected ⟦name(c)⟧Pool(){

        ⟦toFirstLower(name(c))⟧=new java.util.Vector(10, 1);

        for(int i=0; i<10; i++){
            ⟦toFirstLower(name(c))⟧.add(new ⟦name(c)⟧());
        }
    }

    public static ⟦name⟧Pool instance(){
        ...
    }

    public synchronized int indexOf(Object o){
        return ⟦toFirstLower(name(c))⟧.indexOf(o);
    }

    public synchronized void incrementSize(int session){
        ...
    }

    public synchronized ⟦name(c)⟧ getAt(int at){
        return (⟦name⟧)(⟦toFirstLower(name(c))⟧.get(at));
    }

    public synchronized void initializeAt(int at){
        ...
    }

    public synchronized void initializeAll(){
        ...
    }

    public synchronized void resetAt(int at){
        ...
    }
```

```
    public synchronized void resetAll(){
        ...
    }
}
```

The set of pool application logic class instances is contained in the vector named with embedded function ***toFirstLower(name(c))***. The function only changes the first symbol of the application logic class name from capital to small. This vector is initialized in the constructor. In the code presented here the vector is initialized with 10 application logic implementation class instances.

Similarly to data storage class, for each pool class design patterns ***Singleton*** and ***Lazy Acquisition*** are used. ***Singleton*** design pattern is implemented with private static ***instance*** attribute, public static ***instance*** method, and the protected constructor.

Lazy Acquisition pattern is implemented with ***incrementSize*** method. The number of instances in the pool at runtime is the same as the number of allocated sessions. Instances are kept in the attribute with the same name as the application logic implementation class. When the number of sessions increases, immediately, new instances in the pool are allocated. New allocation is done with the ***incrementSize*** method invocation. Method ***incrementSize*** is invoked with the number of sessions in the system as ***session*** parameter. When the value of ***session*** parameter in the ***incrementSize*** method invocation is smaller then the number of sessions in the pool, there is no additional allocation. In the case when the value of session parameter is larger than the number of currently available instances in the pool, the system allocates new application logic implementation class instances. In the constructor code presented above the initial number of allocated objects in the pool is 10.

Operations ***indexOf*** and ***getAt*** are used for retrieving index informations and objects from the pool. Operation ***indexOf*** returns the index of an object in the pool. Operation ***getAt*** returns a reference to the object which with appropriate index in the pool. As it can be later seen index is used in the session identification. Finally, methods ***initializeAt***, ***initializeAll***, ***resetAt***, and ***resetAll*** are used for acquisition and releasing objects for serving user requests.

Application logic implementation classes are generated with two transformations. One transformation generates Java class implementing transformational behavior of UML classes. This transformation is followed by the transformation which generates state machines of the application logic classes with reactive behavior. Transformation for generation of vertical and horizontal class dimension code are $\mapsto_{class}$ and $\mapsto_{stateMachine}$ mappings, respectively. In these functions the result of ***triggerOp*** function is the set of transition triggering operations for the parameter state machine.

$Class(c) \mapsto_{class}$
public class ⟦name(c)⟧{

⟦$\forall a \in ownedAttribute(c)$ apply $\mapsto_{attr}$(a)⟧

⟦$\forall b \in ownedBehavior(c)$ apply $\mapsto_{behaviored}$(c, b)⟧

public void initialize(){

$[\![\forall b \in ownedBehavior(c)$ apply $\mapsto_{smInitialize}$(c, b)$]\!]$

$[\![\forall oA \in ownedAttribute(c)(type(oA) \in Class)$ apply $\mapsto_{classAtributes}$(c, oA)$]\!]$

}

$[\![\forall oO \in ownedOperation(c)(oO \notin InstrumentedElement \wedge$
$oO \notin triggerOp(ownedBehavior(c)))$ apply $\mapsto_{operation}$(oO)$]\!]$

$[\![\forall oO \in ownedOperation(c)(oO \in InstrumentedElement)$
apply $\mapsto_{instrumentedOperation}(oO)]\!]$

$[\![\forall oO \in ownedOperation(c)(oO \notin InstrumentedElement$

$\wedge oO \in triggerOp(ownedBehavior(c)))$ apply $\mapsto_{triggerOperation}(oO)]\!]$

$[\![\forall sm \in ownedBehavior(c)$ apply $\mapsto_{entryAndExitStateActions}(sm)]\!]$

}

$[\![\forall sm \in ownedBehavior(c)$ apply $\mapsto_{stateMachine}$(sm)$]\!]$

Vertical dimension is implemented in such a way that for each UML application logic design class one application logic implementation Java class is generated. Each generated Java class attribute has a corresponding UML class attribute. Attributes are generated with $\mapsto_{attr}$ template. For the brevity, this mapping is here not described.

Java class code which corresponds to an operation of a UML class depends on whether this operation is instrumented or not. Mappings for code generating of instrumented and not instrumented UML model methods are defined with $\mapsto_{operation}$ and $\mapsto_{instrumentedOperation}$ mappings, respectively.

$Operation(o) \mapsto_{operation}$

$[\![toString(visibility)]\!]$ $[\![t = type(o)$ apply $\mapsto_{type}$(t)$]\!]$ $[\![$name(o)$]\!]$(
$[\![\forall oP \in ownedParameter(o)$ apply $\mapsto_{param}$(oP)$]\!]$

$[\![$toString(method(o))$]\!]$
}

$Operation(o) \mapsto_{instrumentedOperation}$

$[\![$toString(visibility)$]\!]$ $[\![t = type(o)$ apply $\mapsto_{type}$(t)$]\!]$ $[\![$name(o)$]\!]$Impl(
$[\![\forall oP \in ownedParameter(o)$ apply $\mapsto_{param}$(oP)$]\!]$

⟦toString(method(o))⟧
}

⟦$toString(visibility)$⟧ ⟦$t = type(o)$ apply $\mapsto_{type}$(t)⟧ ⟦name(o)⟧Impl(
⟦$\forall oP \in ownedParameter(o)$apply $\mapsto_{param}$(oP)⟧){
⟦$t = type(o)$ apply $\mapsto_{returnValueDeclaration}$(t)⟧

PerformanceDataStorage.instance().⟦uName(o)⟧SetSTS(
⟦name(class(o))⟧Pool.instance().indexOf(this), System.currentTimeMillis());

⟦hasType(o)⟧⟦name(o)⟧

PerformanceDataStorageFacade.instance().⟦uName(o)⟧StoreData(
⟦name(class(o))⟧Pool.instance().indexOf(this),System.currentTimeMillis());

⟦$t = type(o)$ apply $\mapsto_{returnValueReturn}$(t)⟧

}

For a non triggering and non instrumented UML operation, the corresponding Java code consists of one function with the equivalent header as the UML operation. Furthermore, the body of the Java function contains copied code of the UML Class function body. As explained in the previous chapter, Java is used instead of a action language.

Instrumented UML operations are implemented with two Java operations. One is the application logic implementation function with suffix ***-Impl***, and the wrapper function in which the measurement takes place. The application logic implementation operation is generated in the same way as non instrumented operation. The wrapper function has the same header and the parameter list as the UML class. In this function, before the invocation of the function with suffix ***-Impl***, the session id and the start time stamp of this method is sent to ***PerformanceDataStorage*** class instance. After the invocation of the function with ***-Impl*** suffix, the method execution time is stored in performance data database.

When function has a return value, with transformation $\mapsto_{returnValueDeclaration}$ a declaration of a variable named ***returnValue*** is generated. Data type of generated ***returnValue*** variable is the type the operation return value. Furthermore, in this case, after the storing of start time stamp, the return value of the operation in which the application logic is implemented is given to ***returnValue*** variable. Finally, after storing the time interval in which the application logic took place, the return value of the wrapper function is the value of ***returnValue*** variable. For example, in the case study, these two functions for instrumented ***getItem*** method look like:

```
public class ItemFacade{

    public String getItem(long itemid, String location){
```

```
        PerformanceDataStorageFacade.instance().⟦uName(getItem)⟧SetSTS(
ItemFacadePool.instance().indexOf(this), System.currentTimeMillis());
        String returnValue = null;

        returnValue = getItemImpl(itemid, location);
        PerformanceDataStorage.instance().⟦uName(getItem)⟧StoreData(
ItemFacadePool.instance().indexOf(this), System.currentTimeMillis());
        return returnValue;
    }
    String getItemImpl(long itemid, String location){
        //The UML model getItem method Java code
        ...
    }
}
```

In the instrumented ***getItem*** code for the reasons of comprehension ⟦uName(getItem)⟧ is left. When the instrumented method does not have a return value, ***returnValue*** variable declaration, assigning return value of the method with suffix ***-Impl*** to ***returnValue***, and return of ***returnValue*** are not generated.

When the application logic implementation class has defined reactive behavior, transformation templates $\mapsto_{behaviored}$, $\mapsto_{entryAndExitStateFunctions}$, and $\mapsto_{entryAndExitStateActions}$ are executed. These transformations are defined as follows:

$Class(c), StateMachine(sm) \mapsto_{behaviored}$

```
    private ⟦name(c)⟧ ⟦toFirstLower(name(c))⟧StateMachine=null;

    public void setState(⟦name(sm)⟧ state){
        ⟦toFirstLower(name(c))⟧StateMachine=state;
    }
```

$StateMachine(sm) \mapsto_{entryAndExitStateActions}$
$\llbracket \forall r \in region(sm)$ apply $\mapsto_{entryAndExit}$(r)$\rrbracket$

$Region(r) \mapsto_{entryAndExit}$
$\llbracket \forall sv \in subvertex(r)(sv \in State \cup InstrumentedElement \wedge sv \notin FinalState)$
apply $\mapsto_{entryAndEnd}$(sv)$\rrbracket$

$State(s) \mapsto_{entryAndEnd}$

```
    void ⟦name(s)⟧Entry(){
        toString(entry(s));
    }

    void ⟦name(s)⟧Exit(){
```

toString(exit(s));

}

$Operation(o) \mapsto_{triggerOperation}$

$[\![toString(visibility)]\!]$ $[\![t = type(o)$ apply $\mapsto_{type}$(t)$]\!]$ $[\![$name(o)$]\!]$
($[\![\forall oP \in ownedParameter(o)$ apply $\mapsto_{param}$(oP)$]\!]$){

$[\![$toFirstLower(name(class(o)))$]\!]$StateMachine.$[\![$name(o)$]\!]$($[\![$name(class(o))$]\!]$Pool
.instance().indexOf(this), $[\![\forall oP \in ownedParameter(o)$ apply $\mapsto_{paramName}$(oP)$]\!]$);
}

$[\![toString(visibility)]\!]$ $[\![t = type(o)$ apply $\mapsto_{type}$(t)$]\!]$ $[\![$name(o)$]\!]$Impl(
$[\![\forall oP \in ownedParameter(o)$ apply $\mapsto_{param}$(oP)$]\!]$){

toString(method(o));
}

Mapping $\mapsto_{behaviored}$ defines the state machine attribute. More precisely, this attribute contains the current state of the object. The state of the instance is set by invocation of method ***setState*** with the new state as the argument. Method ***setState*** is also generated with $\mapsto_{behaviored}$ template.

Transformation $\mapsto_{entryAndEndStateFunctions}$ generate operations in the Java class which conform to ***entry*** and ***exit*** state actions. These operations are invoked when entering and exiting corresponding state, respectively. State transitions are explained later in this section.

Finally, trigger operations of the application logic implementing class are implemented with two functions. Similarly to instrumented operation, implementation of the UML triggering operation application logic is in the Java class with suffix -***Impl***. The Java operation with the same name as the UML operation just forwards the trigger invocations with the tread identification to the state object in the state machine attribute. The mechanism of state machine implementation is depicted later in this section.

State machine code is generated according to ***State*** and ***Singleton*** design patterns [Gamma et al., 1995]. According to ***State*** pattern, a state machine is implemented as a variable for storing current state, one state generalization class, and a set of classes having one class for each state. The variable for storing current state is generated with $\mapsto_{behaviored}$, as previously explained. The state generalization class is generated with transformation $\mapsto_{stateMachine}$.

$StateMachine(sm) \mapsto_{stateMachine}$
public class $[\![$name(owner(sm))$]\!]$StateMachine{

private static $[\![$name(owner(sm))$]\!]$StateMachine instance=null;

$[\![\forall r \in region(sm)$ apply $\mapsto_{trRegion}]\!]$

}
$[\![\forall r \in region(sm)$ apply $\mapsto_{vertRegion}(r)]\!]$

The state generalization class is used as the data type of the application logic implementation class state machine variable. Furthermore, operations of this class are empty versions of all transition triggering operations in the state machine. Empty versions of all transition triggering operations are generated with $\mapsto_{trRegion}$ transformation, and their need is explained later.

$Region(r) \mapsto_{trRegion}$
$[\![\forall t \in transition(r)$ apply $\mapsto_{trOperation}]\!]$

$[\![Transition(t) \mapsto_{trOperation}]\!]$

public [[t=type(operation(event(trigger(t)))) apply $\mapsto_{type}$(t)]] [[name(operation(event(trigger(t))))]](int session, $[\![\forall oP \in ownedParameter(operation(event(trigger(t))))$ apply $\mapsto_{param}$(oP)]]){

$[\![t = type(operation(event(trigger(t))))$ apply $\mapsto_{retNull}$(t)]]

}

Mapping $\mapsto_{retNull}$ is here not defined for the reason of its simplicity. It generates ***return null;*** code when the trigger operation has a return value.

States of a UML State Machine in the model are implemented as classes. State classes extend extend the state machine class which corresponds to their UML State Machine. These classes are generated with the following mapping:

$Region(r) \mapsto_{vertRegion}$
$[\![\forall sv \in subvertex(r)(sv \notin FinalState \wedge sv \notin PseudoState)$ apply $\mapsto_{state}$(sv)]]

$State(s) \mapsto_{state}$
public class [[name(s)]] extends [[name(stateMachine(container(s)))]]{

public static [[name(s)]] instance=null;

protected [[name(s)]]{}

public static [[name(s)]] instance(){
...
}

⟦∀*og* ∈ *outgoing*(*s*) apply ↦$_{outgoingTrans}$(og)⟧
}

Transition(*t*) ↦$_{outgoingTrans}$

public ⟦*t* = *type*(*operation*(*event*(*trigger*(*t*)))) apply ↦$_{type}$(t)⟧
⟦name(operation(event(trigger(t))))⟧(
int session, ⟦∀*oP* ∈ *ownedParameter*(*o*) apply ↦$_{param}$(oP)⟧){

⟦name(owner(stateMachine(container(t))))⟧Pool.instance().
getAt(session).⟦name(source(t))⟧Exit();

⟦returnValueDeclaration(operation(event(trigger(t))))⟧

⟦hasType(operation(event(trigger(t))))⟧
⟦name(owner(stateMachine(container(t))))⟧Pool
.instance().getAt(session).⟦name(operation(event(trigger)))⟧
Impl(⟦∀*p* ∈ *parameter*(*operation*(*event*(*trigger*))))) apply ↦$_{paramName}$(p)⟧);

⟦*so* = *source*(*t*) apply ↦$_{instSource}$(so)⟧

⟦*ta* = *target*(*t*) apply ↦$_{instTarget}$(ta)⟧

⟦name(owner(stateMachine(container(t))))⟧Pool.instance()
.getAt(session).setState(⟦name(target(t))⟧.instance());

⟦name(owner(stateMachine(container(t))))⟧Pool.instance()
.getAt(session).⟦name(target(t))⟧Entry();

⟦retValueReturn(type(operation(event(trigger(t)))))⟧
}

InstrumentedElement(*ie*) ↦$_{instSource}$

PerformanceDataStorage.instance()
.⟦uName(ie)⟧StoreData(session, System.currentTimeMillis());

InstrumentedElement(*ie*) ↦$_{instTarget}$

PerformanceDataStorage.instance()
.⟦uName(ie)⟧SetSTS(session, System.currentTimeMillis());

Classes which represent states are singleton classes. All instances of the application logic class share same state machine objects. This is often the case with ***State*** pattern, because a state defines actions which are executed at entry and exit of that state. Moreover, it

defines a subset of triggering operations which cause the state transitions. Subset is defined with extending the state machine class and overlapping only outgoing transitions' triggering operations. Overlapping operations are generated with $\mapsto_{outgoingTrans}$ transformation. The state machine execution is described in the rest of this section.

Each triggering operation invoked at the application logic implementation class is forwarded to the appropriate operation of the current state object contained in the state machine attribute. A state definition class extends the states generalization class and, therefore, has all triggering operations defined. Outgoing transitions' triggering operations of the current state are overlapped and execute transitions to their target states. In this way, an object changes its state only if the operation which triggers the outgoing transition of the current state is invoked. Else, an empty function from the states generalization class is executed, and there is no effect.

A transition from one state to another consists of four steps. The first step is execution of the exit action of the current state. This is done with the invocation of the appropriate operation of application logic implementation class. Exit action operations are generated with previously defined $\mapsto_{entryAndExitStateActions}$ transformation. The execution of triggering operation code is performed by invoking the application logic class operation with the name of the triggering operation and suffix ***-Impl***. Next, the execution of triggering operation follows. The execution of triggering operation code is followed by setting the target state as the current state. Setting the new state is performed by invoking application logic class operation ***setState*** with the target state object as the argument. Finally, the entry action of the transition's target state is executed.

For the relation between the transition trigger operation and source and target state of the transition ***Moore*** automaton semantics is adopted [Samek, 2002]. According to this automaton, the triggering operation execution belongs to the state which is being left. In the case of instrumented state entry the entry time stamp is saved in the attribute in **PerformanceDataStorage** instance. Similarly, when leaving the instrumented state, after the execution of the exit action of that state and the triggering operation, the execution interval of this state is saved in the performance storage database. Similarly to the instrumented method, in the case of a triggering method with a return value, a variable ***returnValue*** is declared. This variable gets the value of the triggering operation execution. After the transition to the target state is executed, this value is returned.

For the reasons of transformation triviality and brevity $\mapsto_{javaRMIServerDataTypesCode}$ is not described here.

E.4. Transformation to Client Code

Client code defines presentation of data in client server application. It consists of code generated from UML classes stereotyped with ≪Client≫ and classes associated with them, code for communication with server, and datatypes code. Correspondingly

$Model(m) \mapsto_{javaRMIClientCode}$
$[\![$ apply $\mapsto_{client}(\mathrm{m})]\!]$
$[\![$ apply $\mapsto_{ClientSideServerInterfaceImplementation}(\mathrm{m})]\!]$
$[\![$ apply $\mapsto_{javaRMIServerDataTypesCode}(\mathrm{m})]\!]$.

Here should be noticed that datatypes are generated also at the client side for the reasons of availability in a remote environment. The transformation for client code generation is defined in the following:

$Client(c) \mapsto_{client}$
public class ⟦name(c)⟧{

⟦$\forall a \in ownedAttribute(c)$ apply $\mapsto_{attr}$(a)⟧
⟦$\forall b \in ownedBehavior(c)$ apply $\mapsto_{behaviored}$(c, b)⟧
public ⟦name(c)⟧(){
...
}

⟦$\forall oO \in ownedOperation$ apply $\mapsto_{operation}$(oO)⟧
⟦$\forall sm \in ownedBehavior(c)$ apply $\mapsto_{entryAndExitStateActions}$(sm)⟧

}

$Client(c) \mapsto_{clientSide}$
⟦$\forall oA \in ownedAttribute(c)(type(oA) \in Class \wedge type(oA) \notin DataType \wedge type(oA) \notin Client$ apply $\mapsto_{clientSideServers}$⟧

Java code which implements classes with ≪Client≫ stereotype is similar to server application logic implementation classes. It consists of a Java class and state machine implementation classes. For each attribute of the UML client class there is an attribute in the appropriate Java class, and for each UML client class operation, there is an appropriate Java class operation. Furthermore, client can also have reactive behavior defined. Code generation for clients' state machines is equivalent to servers' state machines, and the definition is here not repeated.

Classes which are data types of client attributes and do not have stereotype ≪DataType≫ are generated with the following transformation:

$Property(p) \mapsto_{clientSideServers}$
public class ⟦name(type(p))⟧{

protected ⟦name(type(p))⟧(){}

⟦$\forall oO \in ownedOperation(type(p))$ apply $\mapsto_{clientSideServerOperations}$(oO)⟧

}

$Operation(o), Client(cli) \mapsto_{clientSideServerOperations}$
⟦toString(visibility(o))⟧ ⟦type(o)⟧ ⟦name(o)⟧(⟦$\forall op \in ownedParameter(o)$ apply $\mapsto_{param(op)}$⟧){

⟦hasType(type(o))⟧ ⟦name(cli)⟧ClientSideServerInterfaceImplementation.instance().

⟦uName(o)⟧(⟦$\forall p \in ownedParameter(o)$ apply $\mapsto_{paramName}$⟧);

}

Clients' attributes datatype classes without stereotypes have only operations in their definition. Because their control flow implementation is in the server code, at the client side they only forward the invocation to the corresponding operation of the server communication class. The mapping $\mapsto_{ClientSideServerInterfaceImplementation}$ generates the class for communication with the server.

$Class(c) \mapsto_{ClientSideServerInterfaceImplementation}$

```
import java.rmi.*;
import java.rmi.registry.*;
import java.net.*;

public class ⟦name(c)⟧ClientSideServerInterfaceImplementation{

    private int session=-1;

    private static ⟦name(c)⟧ClientSideServerInterfaceImplementation instance=null;

    private ⟦name(model(c))⟧ServerInterface server;

    protected ⟦name(c)⟧ClientSideServerInterfaceImplementation(){

        System.setSecurityManager(new RMISecurityManager());

        try{

            server=(...);

            session=server.connect();

        }

        catch (Exception ex){
            ex.printStackTrace();
        }

    }

    public static ⟦name(c)⟧ClientSideServerInterfaceImplementation instance(){
        ...
```

```
    }
    ⟦∀oA ∈ ownedAttribute(c)(oA ∈ Class ∧ oA ∉ DataType ∧ oA ∉ Client)
                                        ↦operationImplementations(oA)⟧
}

Property(p) ↦operationImplementations
⟦∀oO ∈ ownedOperation(type(p)) apply ↦opImplementation(oO)⟧

Operation(o) ↦opImplementation

    ⟦toString(visibility(o))⟧ ⟦type(o)⟧ ⟦uName(o)⟧(
                            ⟦∀p ∈ ownedParameter apply ↦param(p)⟧){
        ⟦returnValueDeclaration(o)⟧

        try{
            ⟦retValue(o)⟧ ⟦name(model(o))ServerInterface.⟧⟦uName(o)⟧(
                        ⟦session, ∀p ∈ ownedParameter(p) apply ↦ paramName(p)⟧);

        }

        catch (Exception ex){
            ex.printStackTrace();

        }

        ⟦returnValueReturn(o)⟧

    }
```

Class for communication with the server is a singleton class which, when instantiated, makes a connection with the server. With making the connection, it receives the identification of the session for serving its requests. Operations defined in this class conform to operations of the server interface except for connection and disconnection. Operations of this class only add the session identification to parameters received from client and forward the invocation to the server. In this way, they mediate between the client and the server.

At the end, there should be noticed that data type definitions have to be available both at the client and server. Therefore, for both sides code from classes with stereotype ≪DataType≫ also generated. However, for the reasons of brevity and triviality of the transformation, this code is here not depicted.

Curriculum Vitae

Personal Information

Name: Marko Bošković
Address: 6675 Dow Avenue, Suite 109, V5H 3E1, Burnaby, BC, Canada
Place of Birth: 19300 Negotin, Serbia
Date of Birth: August 18th, 1980

Studies

04/2005-03/2009 PhD student, University of Oldenburg, Germany
04/2005-03/2008 Scholarship Holder, DFG Research Training Group "Trustsoft"
9/2004 Diploma Engineer of Computer Technology and Informatics
10/1999-9/2004 Student of Computing Technology and Informatics, Military Academy, Belgrade, Serbia

Work Experience

11/2009-12/2009 Research Assistant, Athabasca University, Alberta, Canada
07/2009-10/2009 Researcher, Athabasca University, Alberta, Canada
04/2008-09/2008 Scientific Help, University of Oldenburg, Germany
10/2004-03/2005 Database and Network Administrator, Sector for Human Resources, Government of Serbia and Montenegro

School

09/1995-08/1999 Military Grammar School, Belgrade, Serbia
07/1987-07/1995 Primary School "Branko Radičević, Negotin, Serbia

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Dissertation selbstständig verfasst habe und nur die angegebenen Hilfsmittel verwendet habe. Teile der Dissertation wurden bereits veröffentlicht bzw. sind zur Verentlichung eingereicht, wie an den entsprechenden Stellen angegeben. Die Dissertation hat weder in Teilen noch in ihrer Gesamtheit einer anderen wissenschaftlichen Hochschule zur Begutachtung in einem Promotionsverfahren vorgelegen.

Burnaby, im März 2010 Marko Bošković