Model Management with Relation Types

by

Michalis Famelis

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Model Management with Relation Types

Michalis Famelis

Master of Science

Graduate Department of Computer Science

University of Toronto

2010

Software development in application domains where model-based techniques are employed faces challenges related to managing sets of inter-dependent models of various types. In such settings, capturing the semantic structure of sets of models, as well as being able to check the validity of relationships between models that constitute this structure, can be of significant importance. Additionally, as inconsistencies can arise between models that are not obviously related, the ability to infer implicit relationships between models can help to further enforce the semantic cohesiveness of sets of related models.

To this end, we propose an approach to model management centered around the declaration and definition of relation types. We describe how hierarchies of such relation types can be created to express their purpose, application scenarios and well-formedness rules. Such hierarchies consist of abstract relation types via which the purpose for relating models can be expressed. Concrete relation types for specific application scenarios can then be created by extending the abstract ones. Such concrete relationships are tied to particular metamodels, constrained by well-formedness rules, which can be used to verify the proper application of relation types.

We showcase our approach by applying it to the domain of automotive software engineering, a domain where model-based techniques are becoming increasingly prominent, to demonstrate how typed relationships can be employed in multi-model environments to check for consistency between models and to compose relationships to infer new ones.

Dedication

This work is dedicated to my parents, Polymnia and Giorgos and my siblings, Marietta, Panagiota and Panagiotis.

Acknowledgements

I would like to thank the following people for helping me complete this work.

Professor Marsha Chechik for her guidance and patience. Robert Baillargeon of Panasonic and Sighe Wang of General Motors for their valuable input. Rick Salay and Jocelyn Simmonds for helping me shape my ideas about how to structure and define Relations. Jorge Aranda for providing me with healthy food during the composition of the thesis. And last but not least, the communities of developers that helped create the fantastic open source tools that I used to carry out and present this work.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem area

The ever increasing complexity of the technical and the relevant socio-technical systems that are involved in the lifecycle of software artifacts poses a variety of difficult problems. These problems stem from the appearance of accidental complexity in a range of settings in the overall creation process and lifecycle of software. Model Driven Engineering (MDE) has been proposed as an approach to tackle those, based on the usage of models to help raise the level of abstraction, highlight various perspectives of systems and reduce effort by supporting transformation, model analysis and code generation [22].

In development settings where model based techniques are involved, development is bound to include large numbers of models. Such models can originate from diverse sources and can be dependent in a variety of ways. The need to comprehend, represent, analyze and manipulate such sets of interdependent models has given rise to the discipline of model management. Various model management techniques have been developed in an attempt to tackle the complexities of handling sets of interdependent models, such as [12], [16], [15] and [40].

As any approach to model management is concerned with the interdependencies be-

tween models, the issue of handling relationships among models that express such dependencies is fundamental. Related models can be heterogeneous, as it is possible that they are defined using diverse tools and notations. Moreover, there are various kinds of dependencies that can exist between models, which reflect the various ways models can constrain one another [40]. The creation of mappings between models often relies on human intervention or heuristics [15]. Similar issues are of concern in the area of schema integration, where efforts are made to raise the level of abstraction and create tools to facilitate human designers in creating and managing mappings [13].

1.2 Summary of problem statement and approach

In this research context, i.e., of facilitating the management of relationships between models, we aim to create a conceptual infrastructure by which human designers engaged in managing models can be able to do so with a better understanding of their semantics, as proposed in [13]. We therefore focus on the issue of managing the various ways that models can be related.

Therefore, in this thesis we propose an approach to managing models in multi-model environments that is centered around the declaration and definition of hierarchies of relation types that can convey the purpose for establishing dependencies between models in various application scenarios. We argue that employing such an approach can enable developers to semantically structure sets of heterogeneous models, as well as check them to ensure that such a semantic structure is valid.

Moreover, by using relation types to organize the various dependencies between models, we can create support for various MDE tasks in multi-model environments, such as composing relationships to infer new ones, thus exposing implicit knowledge and possibly non-evident inconsistencies between models that are non obviously related.

We demonstrate this approach by applying it to the domain of automotive software

engineering, an area where model-based techniques are becoming increasingly prominent. This is evident by industry-wide collaborations to create model-based standards, such as AUTOSAR [3], to address issues of the domain, as well as by the various practical model-based approaches to automotive software development that have been presented, such as [10], [9] and others. The various challenges facing the domain of automotive software engineering have lead to calls from software researchers in the field to tackle the particular challenges in the automotive software domain, including issues related to integrated, seamless model-driven approaches [18], [36].

The domain of automotive software engineering is therefore a good example of a domain where our approach to model management can be employed with tangible benefits. By demonstrating its application in such a domain we hope to showcase how our approach attempts to tackle issues pertaining to the wider model management research area and therefore to the application of model based techniques in software engineering in general.

1.3 Organization of the thesis

In Chapter 2, we give an overview of some basic principles of the domain of automotive software engineering, along with its basic model types and establish a running example, which we use throughout the thesis to demonstrate our relation-centric approach.

In Chapter 3, we present the theoretical background for our approach, which is then presented in Chapter 4. We outline the basic mechanics of our approach, describing how hierarchies of relation types can be defined, from abstract relations to concrete ones, tied to particular application scenarios.

In Chapter 5, we present two applications of our approach to practical problems from the area of model management, in particular, consistency checking and relationship inference. Chapter 6 contains a description of how the Model Management Tool Framework (MMTF) can provide tool support for our approach.

Chapter 7 presents a brief overview of related work, and the thesis concludes in Chapter 8 with an assessment of our approach and future steps.

Chapter 2

Motivating example: the Automotive Domain

The emergence of automotive software has been an evolutionary, bottom-up process [18]. This, along with the inherent constraints of the domain have contributed to a number of domain-specific characteristics. In the following we present a brief overview of some basic principles and kinds of models of the automotive software engineering domain, along with establishing a motivating example which we use in the following chapters to demonstrate the concepts relevant to our approach. Our view of the automotive domain has been shaped by literature such as [10], [37], [36] and [18] and discussions with domain experts.

2.1 Principles

2.1.1 Physical architecture

Historically, automotive software systems have comprised of software embedded in dedicated controllers (*Electronic Control Units*, ECUs). ECUs can be dedicated to particular functions, however applications can also be distributed among several communicating ECUs.

This points to a degree of coupling between application software and physical ECU networking architecture. The various software components that are being built are ultimately assigned to particular (possibly sets of) ECUs, and configured accordingly. Moreover, issues pertaining to networked, parallel software components, such as communication, synchronization and data integrity, need to also be addressed.

2.1.2 Reactive systems

While nowadays software in vehicles can be found even in secondary application domains such as entertainment, the majority of automotive software components are reactive systems. That is, the functionality of the embedded software controllers can be summarized in an infinite loop consisting of a step reading input from sensors and a step of triggering actuators according to the input [8].

This points to a set of issues related to scheduling controllers, as well as calibrating data access mechanisms, such as polling or data pulling/pushing. Issues related to networking and parallelism are also relevant. Moreover, the sensor/controller/actuator schema is pervasive in the domain and often defines the architectural alignment of software components.

2.1.3 Signal-based communication

The reactive nature of the controllers and the coupling with physical architecture also impact the ways with which physical as well as logical components can communicate. This is further exacerbated by the sensor/controller/actuator schema of automotive software systems.

Stemming from long-standing control engineering traditions by practitioners in the field, automotive software systems tend to be designed in a componentized fashion, where information is passed around by means of software signals. Data calibration is of significance with this regard, as well as the data access mechanisms mentioned earlier.

2.1.4 Variability and reusability

The area of automotive software engineering faces much of the same issues that the general automotive engineering domain has to deal with, such as rapid development cycles, changing requirements and needs for suppression of costs. Therefore, issues of building software in such a manner as to be able to support functional variability, as well as being able to achieve a high degree of reusability are of significant importance.

This often results in designing software components in a very minimalistic manner with regard to their functionality and interfaces. This enables building bundles of existing subcomponents to create components implementing a desired feature set.

2.1.5 Code generation

Model-based techniques are widely employed by the automotive industry [18] [10], as they allow high level design and code generation, while supporting more formal approaches to evaluating and ensuring software quality. Therefore, domain-specific modeling languages are employed in order to create highly stylized models that are suitable for code generation. The use of custom code generators further allows for flexibility in the process, while achieving a degree of correctness and robustness.

2.2 Motivating example

We now present a motivating example which we use in the following in order to demonstrate the various kinds of modeling artifacts that occur in the domain.

We assume the scenario (henceforth referred to as WipersStory) where an automotive company outsources the creation of a windshield wiper subsystem for a vehicle to a supplier, who is tasked with producing the hardware and software for the controller of the system. The automotive company needs to produce a specification of the wipers subsystem and describe how it fits with the larger architecture of the vehicle. The

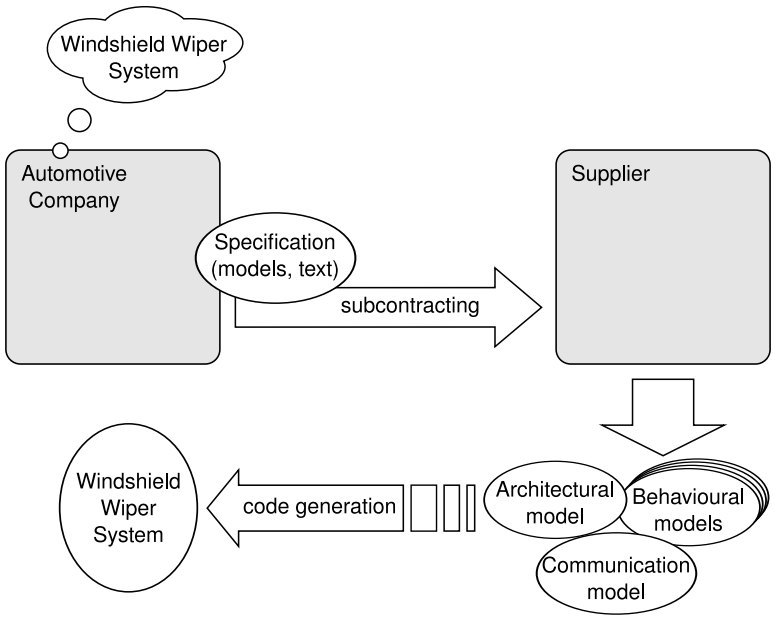


Figure 2.1: Illustration of the motivating example (WipersStory).

supplier must then refine the specification to a design, making critical implementation decisions.

The specification produced by the automotive company describes the behavior of the system, as well as how it fits into the larger architecture of the vehicle. To keep things simple in the context of this thesis, we assume that the behavior is described in English, while the architecture is given by a UML Component Diagram such as the one in Figure 2.2.

In the diagram, the automotive company defines that the windshield wiper subsystem software controller, seen as a black box, should provide a set of interfaces by which its functionality can be invoked by other software subsystems of the vehicle. In particular, the controller should expose interfaces for activating and deactivating the windshield wipers, for setting their speed level (normal, fast and intermediate) and for activating the temporary windshield cleaning mode.

WiperStory takes place in a development environment where model-based techniques are employed. So, the supplier creates a set of models detailing the various views (archi-

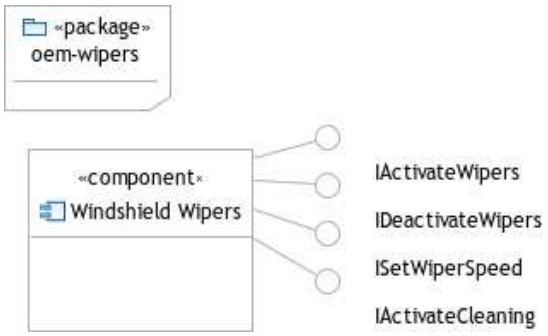


Figure 2.2: UML Component Diagram created by the automotive company.

tectural, behavioral and others) of the system. These models are then used by specialized code-generators to create the actual software system.

Figure 2.1 shows a top level view of WipersStory (the various modeling artifacts produced by the supplier is explained in the following.)

2.3 Modeling artifacts

The principles described in Section 2.1 shape the development process and the produced artifacts for automotive software systems. The overall process can be viewed as refinement in the MDE sense. In this vein, software modeling techniques can be employed to capture the produced artifacts as models of various kinds, such as class diagrams, state machines and others. In the following we present some basic kinds of such models.

In the following, we occasionally employ terminology borrowed from Object Oriented Programming (OOP) for naming some of the domain-specific artifacts. While these artifacts play similar roles as their generic OOP counterparts, they are highly stylized and often have additional characteristics and limitations, stemming from their domain specificity.

The metamodel for these models can be found in the Appendix.

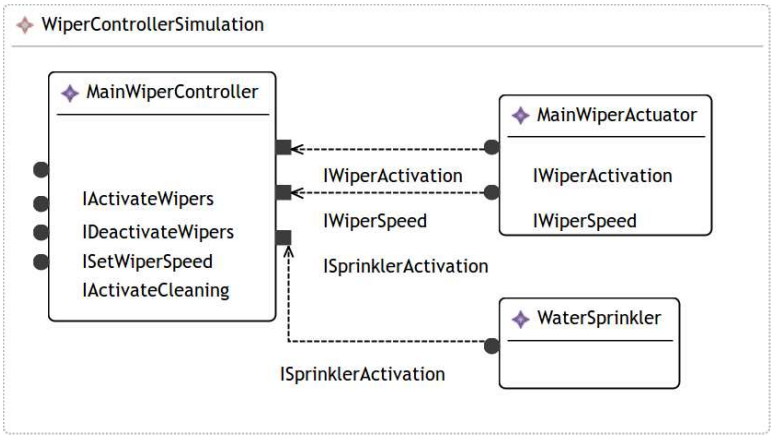


Figure 2.3: Algorithm Block Diagram for WipersStory.

2.3.1 Logical architecture

Any particular automotive software application is organized as a collection of algorithmic artifacts, termed *Algorithm Blocks*, the domain-specific counterpart of Classes. Each of them represents one basic functional aspect of the system and roughly corresponds to a particular element in the sensor/controller/actuator schema described in Section 2.1.2. Algorithm Blocks also represent the basic schedulable elements of the system and contain information for calibrating various relevant data storage items.

We refer to a model that depicts the logical architecture of an automotive system as an *Algorithm Block Diagram*. The Algorithm Block Diagram for WipersStory can be seen in Figure 2.3. In the diagram, the Supplier decomposes the overall software subsystem into three logical parts, represented by three Algorithm Blocks, and defines the interfaces by which these communicate amongst themselves and with their wider environment.

In particular, the `MainWiperController` block represents the logical part of the architecture that contains the business logic for controlling the various actuators of the wipers subsystem. The `MainWiperActuator` and `WaterSprinkler` blocks represent the logical part of the architecture that manages the actuators that animate the physical parts of the wipers subsystem. Communication between the actuators and the controller is done via three internal interfaces which are required by the controller block and provided by

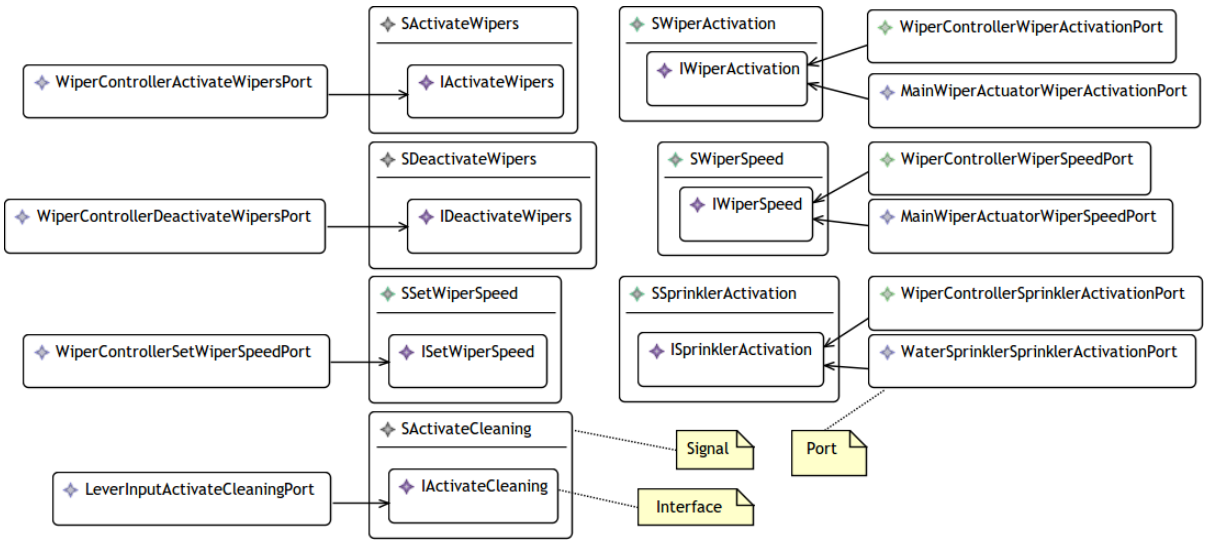


Figure 2.4: Signals Diagram for WipersStory.

the actuator blocks. The main controller also provides the interfaces which are required by the automotive company to be exposed to other software subsystems of the vehicle.

The metamodel for Algorithm Block Diagrams can be found in the Appendix (Figure A.1).

2.3.2 Communication architecture

Algorithm Blocks exchange information via Signals, accessed via domain-specific Interfaces. An Interface contains methods for sending and receiving one particular Signal, as described in Section 2.1.3. Therefore, for a Class to be able to send or receive a number of Signals, it needs to implement or require an equal number of Interfaces. (This limitation arises from the need for reusability discussed in Section 2.1.4.)

We refer to a model that depicts the communication architecture of an automotive system as a *Signals Diagram*. The Signals Diagram for WipersStory can be seen in Figure 2.4. In the diagram, the Supplier creates declarations for all the Signals present in the subsystem. These include both the signals that are needed for internal communication among the various Algorithm Blocks defined in Figure 2.3, and the signals that will be

passed to other software subsystems in the vehicle via the interfaces that the automotive company has specified to be exposed by the component. Each of the signals contains a corresponding Interface declaration. These interface declarations are referenced by the various ports which are declared by relevant Algorithm Blocks.

The metamodel for Signals Diagrams can be found in the Appendix (Figure A.2).

2.3.3 Behavioral definition

As was described above, every Class represents a particular functional aspect of the system. Therefore, every Class is associated with a particular domain-specific Statemachine which captures its behavior. As such Statemachines can ultimately be employed for code generation, various constraints are in place in order to simplify them for code generation per Section 2.1.5 and improve their maintainability.

For instance, in order to ensure that the state transitions are deterministic, Statemachines are constrained in that only one outgoing transition exists from every state. This in turn implies heavy usage of nested states, with priorities of the transitions of the various super/sub-states being used to ensure determinism.

The Statemachine for the `MainWiperController` Algorithm Block in WipersStory can be seen in Figure 2.5. In order to satisfy the requirement that for each state there is only one outgoing transition, additional states have been introduced. These follow a pattern of distinguishing between on/off modes. This is evident in the diagram in the case of the general On/Off pair of states for the entire wiper system, as well as for more detailed behavior such as turning the cleaning system on and off and alternating between the intermediate speed mode being on as opposed to the other two speed modes.

Whenever the states are entered or exited, the statemachine produces relevant signals which were defined for internal communication within the component (see previous section). For example, whenever any of the two states named `Cleaning` is entered, the `SSprinklerActivation` signal is broadcasted to the `WaterSprinkler` algorithm

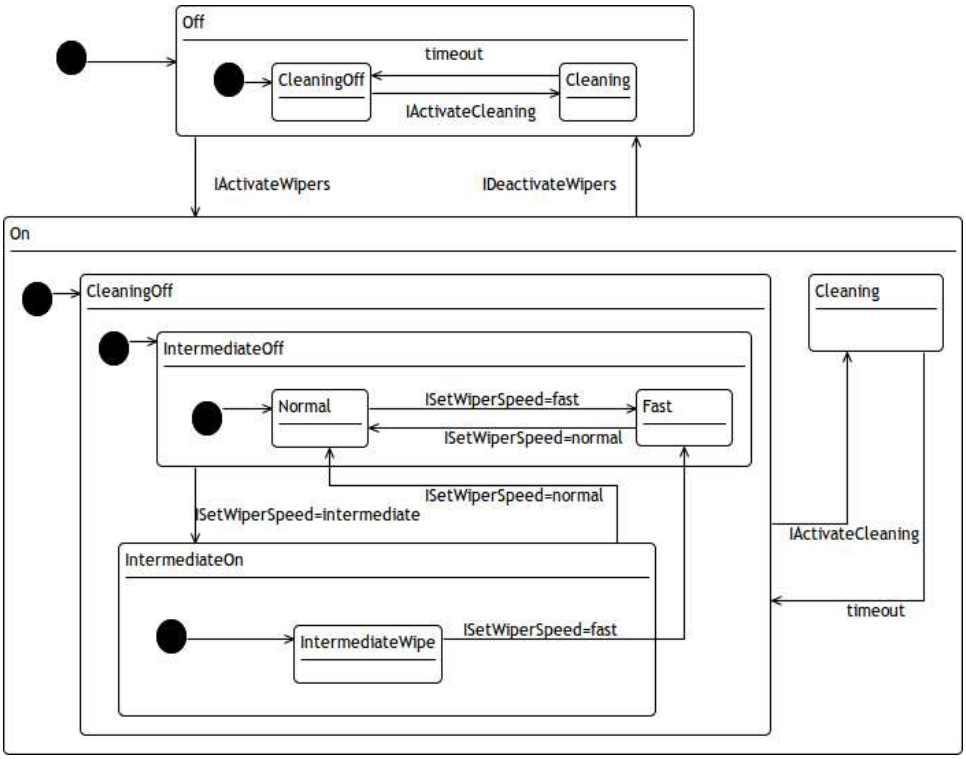


Figure 2.5: Statemachine for the `MainWiperController` Algorithm Block in WipersStory.

block, to trigger sprinkling of the windshield with water. Similar signals are sent to the `MainWiperActuator` algorithm block to trigger wiping at the appropriate speed, and a similar set of signals is broadcasted when exiting the `Cleaning` states.

The metamodel for these domain specific Statemachines can be found in the Appendix (Figure A.3).

2.3.4 System architecture

The various diagrams can be bundled together in Component diagrams, which are used to capture the allocation of the various algorithmic entities to the sets of ECUs on the physical networked system. In the same sense, Component diagrams can be employed to represent the interconnections among various application components, eg. between a particular component and a larger subsystem architecture. For the purposes of this thesis, we do not go into detail about these models, and we represent them as UML

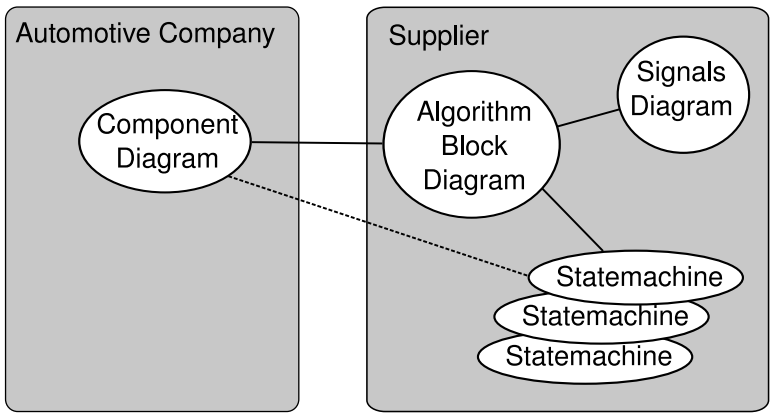


Figure 2.6: Some possible dependencies in WipersStory.

Component Diagrams, conforming to the basic Component Diagram metamodel taken from the UML specification[35], as shown in Figure A.4 of the Appendix.

2.4 Towards an approach for model management of Automotive software

It is evident that the various modeling artifacts created at various stages in WipersStory are related in a variety of ways.

1. On a basic level, the various models are related as parts of a whole. This means that various dependencies exist between them. For example:

   - The interface declarations at the Signals Diagram must be consistent with the interfaces employed in the Algorithm Block Diagram.
   - A state machine must be declared for each algorithm block in the Algorithm Block Diagram.
   - The triggers in the transitions of state machines must correspond to the declared interfaces of their respective algorithm block.

2. In a larger context, which can potentially take into account the software lifecycle, requirements modeling etc, it becomes evident that some way of managing these sets of models is imperative. The ability to make explicit the various ways that the models are related offers a means to accomplish meaningful management and analysis. A first step towards such capabilities is ensuring that the various dependencies between the models are properly captured.

3. If relationships between modeling artifacts are explicitly captured, further relationships can be inferred. For example, if the consistency relationship between the Component Diagram specification and the Algorithm Block Diagram is explicit, a relationship can be inferred to trace the invocations of high level interfaces to low level statemachine triggers and state transitions. Such a trace relationship could be used for troubleshooting and diagnostics.

These examples point us to the direction of further studying how we can effectively employ model management techniques in the automotive domain. From the above discussion, we infer that in order to be able to facilitate model management in the domain, an approach is needed that should have the following characteristics:

1. It should enable developers to reason about how a set of modeling artifacts is semantically structured.

2. It should provide the infrastructure for validating this structure to expose potential inconsistencies.

3. It should facilitate the inference of implicit knowledge about the ways that models are dependent in non-evident ways.

In the following chapters we present an approach to model management based on declaring and defining relation types.

Chapter 3

Background

3.1 Models and model types

Models are traditionally employed in Software Engineering to abstractly describe existing or hypothetical software-related artifacts for a variety of purposes [31]. In the context of Model Driven Engineering (MDE), models are first class items and are used to such ends as design, documentation, code generation and others.

While models are usually employed to describe actual systems, they can also be used to describe other models. This notion is called *metamodeling*. A metamodel that is constructed with respect to concepts related to a particular application domain, along with a (usually graphical) notation, can be used to describe a *domain specific modeling language (DSML)*. In Chapter 2, a high level description of an automotive DSML is presented.

Following the definition in [11], we consider a model to be "a set of objects, each of which has properties, has-a relationships and associations". Every model must conform to a respective metamodel.

A widely adopted approach to metamodeling is the Meta Object Facility (MOF) [32]. As is described in [17], in the context of MOF, a model m is a collection containing typed

data elements e. Each element e is of a type T, denoted as $e : T$. We say that a model m *conforms to* a metamodel M for which we adopt the following definition, based on [40]:

Definition 1. *A metamodel is the tuple $\langle \Sigma, \Phi \rangle$, where Σ is the set of metamodel element types and Φ is the set of metamodel well-formedness constraints. These constraints are either structural, i.e. imposed by the various forms of association between the element types, or constraints attached to the metamodel as a whole. We refer to the former as "structural constraints", symbolized as Φ_s and to the latter as "metamodel constraints", symbolized as C. Obviously, $\Phi = \Phi_s \cup C$.*

While the notions of a metamodel and a *model type* can be conflated [41], in the context of this thesis we consider a model type to be *associated* with a metamodel as follows.

The MOF diagram[1] M of a given metamodel consists of its set of element types Σ and its set of structural constraints Φ_s. Its metamodel constraints C are expressed in a language such as OCL [33] and are attached to the metamodel diagram M. In view of the above, we adopt the following definition of model types:

Definition 2. *We consider a model type T_M to be a tuple $\langle M, C \rangle$, where M is a metamodel diagram and C a (possibly empty) set of constraints associated with it.*

We say that a model m *is of type* T_M, and write $m : T_M$ if the model conforms to the type's associated metamodel and constraints.

We adopt the following subtyping approach:

Definition 3. *Given a model type $T_M \equiv \langle M, C \rangle$ where M is a metamodel and C a (possibly empty) set of constraints, we consider the type $T'_M \equiv \langle M', C' \rangle$ to be its subtype if $M' = M$ and $C \subset C'$.*

In other words, the subtype of a model type has the same metamodel diagram as the supertype and its associated set of constraints is a superset of the supertype's set

[1] The notation for MOF diagrams is a simplified version of UML class diagrams.

of constraints, augmented with constraints particular to the subtype. In this vein, we consider the model type $T_M \equiv \langle M, \varnothing \rangle$ to be the supertype of all the model types that are defined by constraining the metamodel M. This notion of subtyping allows subtype substitutability, in the sense that it allows using a model of some type in a place where a model of one of its supertypes is expected.

3.2 Macromodeling

Macromodeling is an approach to model management which aims to represent the various ways by which models are related, therefore allowing developers to perform various model management tasks, such as model matching, mapping composition, model differencing, model generation, model merging and others [11]. In the following we briefly present the key points.

As presented in depth in [40], macromodeling is a rich approach, however we limit the discussion to aspects that are of particular use to our relation-centric approach as presented in Chapter 4. In this vein, we do not go into the details of the formalism and we do not discuss additional features described in [40], such as support for unrealized models, macrorelations and others.

3.2.1 Model mappings

Fundamental to macromodeling is the idea of relating models. As described in [11], "two models are related when the possible interpretations of one model constrain the possible interpretations of the other model". In the following we use the term "mapping" to refer to the abstract concept of relating two models, while later in the paper we will use the term "relation" to refer to a mapping conceived with a particular intention or purpose in mind.

Below we present a formal foundation of what it means for two models to be related.

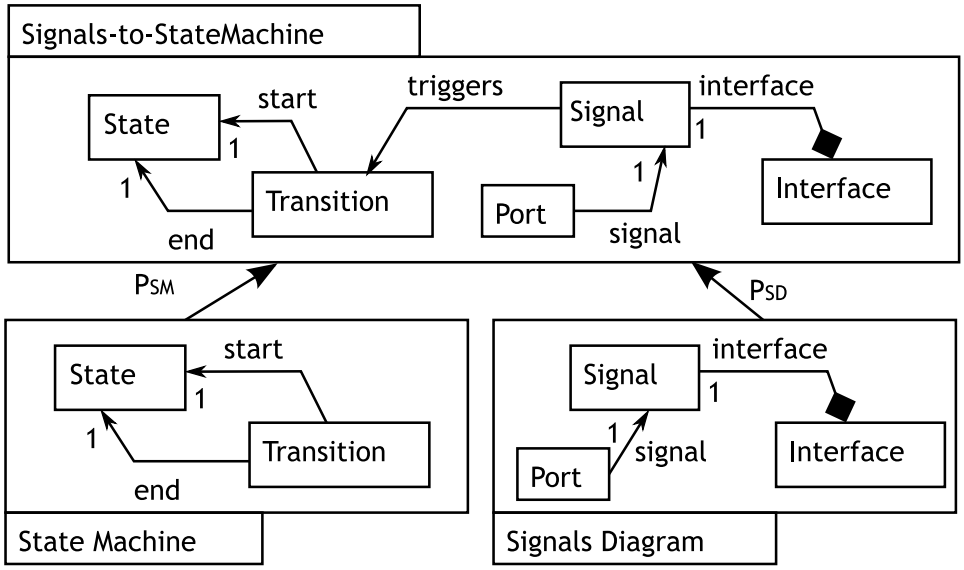


Figure 3.1: Example of using the metamodel morphisms p_{SM} and p_{SD} to create a relator metamodel.

The formalism can be extended for more than two models. We introduce the concept of *metamodel morphisms*, based on institution theory [23], as described in [40]:

Definition 4. *For two metamodels A and B, a metamodel morphism: $f : \langle \Sigma_A, \Phi_A \rangle \rightarrow \langle \Sigma_B, \Phi_B \rangle$ is a homomorphism of the signatures $f_\Sigma : \Sigma_A \rightarrow \Sigma_B$ such that $\Phi_B \models f(\Phi_A)$. The function f is one that translates sentences over Σ_A to ones over Σ_B according to the mapping $f_\Sigma : \Sigma_A \rightarrow \Sigma_B$.*

For each couple of model types that need to be mapped, a *mapping type* associated with a *relator metamodel* is defined. Every mapping is itself a model and for a given pair of model types, their mapping model must conform to the relevant relator metamodel. Metamodel morphisms from them to the relator metamodel are used to ensure the well-formedness of their projections in the mapping metamodel. Figure 3.1 presents an example of using metamodel morphisms to create a relator metamodel from two (simpler versions of) model types defined in Chapter 2.

In particular, the `Signals-to-StateMachine` relator metamodel contains projections of the (simplified) `StateMachine` and `SignalsDiagram` metamodels. The metamodel

morphisms p_{SM} and p_{SD} ensure that the correct projection of the two metamodels in the relator metamodel. Apart from metamodel projections from the endpoint model types, the relator metamodel can be injected with additional elements. In this case, the relator metamodel additionally contains the association `triggers` between the `Signal` element of the signals diagram metamodel projection to the `Transition` element of the state machine metamodel projection, to signify that a transition in state machines can be triggered by signals defined in a signals diagram model.

Metamodel morphisms ensure that relator metamodels represent correctly the mapped model types. Relator metamodels can provide additional semantic constraints relevant to the particular intent which the mapping type attempts to address. For a mapping between metamodels $A \equiv \langle \Sigma_A, \Phi_A \rangle$ and $B \equiv \langle \Sigma_B, \Phi_B \rangle$, and metamodel morphisms p_A and p_B, the relator metamodel is defined as:

$$\langle \Sigma_A \cup \Sigma_B, \quad p_A(\Phi_A) \cup p_B(\Phi_B) \cup \Phi_{RC} \rangle$$

In other words, the relator metamodel's set of elements contains all the element types of the endpoint metamodels A and B. The relator metamodel's set of well-formedness constraints consists of the constraints of the endpoint metamodels as translated by the metamodel morphisms p_A and p_B. The set of well-formedness constraints of the relator metamodel can be augmented by the set of *relator constraints* Φ_{RC} which express semantics particular to the of the relator metamodel itself.

In view of the above, we can describe the relator metamodel shown in Figure 3.1 as follows.

We represent the `StateMachine` metamodel as $\langle \Sigma_{SM}, \Phi_{SM} \rangle$ and the `SignalsDiagram` metamodel as $\langle \Sigma_{SD}, \Phi_{SD} \rangle$. The Σ factors represent the set of elements in each metamodel. For the state machine metamodel this set consists of the elements `State` and `Transition` and for the signals diagram metamodel this set consists of the elements `Signal`, `Port` and `Interface`. The Φ factors represent the well-formedness rules of the two metamodels. For state machines, these rules are that a transition has a start and end state and for

signals diagram that a signal contains an interface and is associated with a port. Any additional metamodel constraints for these, such as OCL constraints would be added to these sets.

The `Signals-to-StateMachine` relator metamodel would then be represented as:

$$\langle \Sigma_{SM} \cup \Sigma_{SD}, \quad p_{SM}(\Phi_{SM}) \cup p_{SD}(\Phi_{SD}) \cup \Phi_{RC} \rangle.$$

The above indicates that the relator metamodel contains all elements contained in the endpoint metamodels and all their well-formedness constraints, as translated by the metamodel morphisms p_{SM} and p_{SD}. The set of relator constraints Φ_{RC} contains well-formedness rules that are particular to this metamodel, in our example, the `triggers` syntactic element.

In general, additional constraints can be injected to a relator metamodel by appending them to its set of relator constraints Φ_{RC}. For example, we could require that all transitions should be triggered by some signal. Such a new rule can be incorporated to the relator metamodel as an OCL constraint added to the its set of relator constraints Φ_{RC}. We employ this method to define the semantics of concrete relation types in Section 4.3.1.

3.2.2 Macromodels

A *macromodel* is a kind of model that provides a way to capture how various models are related at a high level of abstraction. As defined in [40]:

Definition 5. *"A macromodel (is a model that) consists of elements denoting models and links denoting intended relationships between these models with their internal details abstracted away."*

Being a model itself, a macromodel must conform to a metamodel, and therefore the notion of macromodel type can be defined. A macromodel type is characterized with respect to the model and mapping types it allows instances of it to contain.

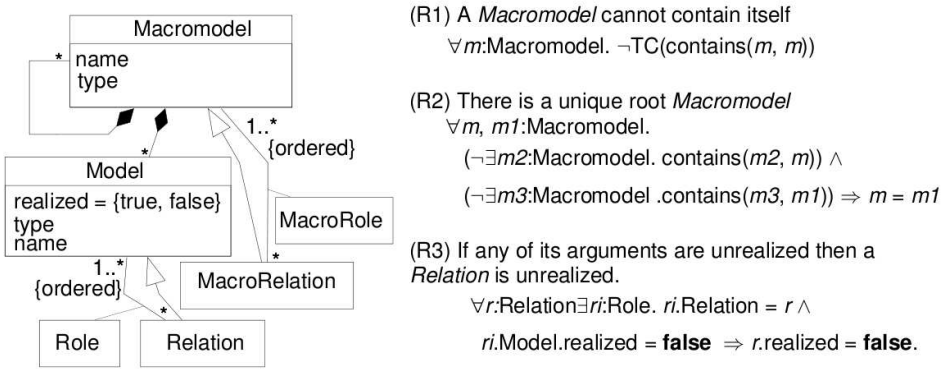


Figure 3.2: Abstract syntax and well-formedness constraints of macromodels [40].

With regard to syntax, the constituent models of a macromodel are represented by typed model elements, while relations between them are represented by typed association links. The metamodel and well-formedness constraints for macromodels are presented in Figure 3.2.

As can be seen in the metamodel, macromodels can contain models, as well as other macromodels. This feature enables the creation of a hierarchical structure of macromodels to represent various levels of abstraction of a modeled system. Additionally, macromodels can also contain relationships, which are themselves models, conforming to relator metamodels that are particular to the endpoint model types that they connect, as discussed in Section 3.2.1. In addition, relationships are labeled with a particular role. In Chapter 4 we present an approach to model management that is based on an expansion of this feature, to create full fledged *relation types*.

The focus of this work being on relation types, we frequently discuss declaring models and then establishing relationships between them. This is always thought in the context of macromodelling. In this respect, whenever we refer to two or more models being related, we assume that there exists a macromodel containing these models as elements. Similarly, creation of relationships between these models is also considered in the same setting.

Chapter 4

Relation-centric Approach

As was discussed in Section 2.4, there exists a need for managing sets of models in the automotive and other domains. We propose an approach for model management, which focuses on explicating dependencies between models as relationships that conform to an explicit relation type, in the context of a macromodeling framework. Relator models conforming to appropriate relation types can be created and employed to capture information about the meaning and the intent of the relationship between models. Moreover, these relator models demonstrate how the related models are dependent upon each other, by using the syntactic structure of model mappings, along with the semantics provided by their respective relation type.

In other words, by focusing on the relationships between models, we can capture *why* and *how* they are related. Additionally, relation types enable checking and enforcing that artifacts are properly related.

Making this meta-information readily available in a macromodeling context enables us to devise relation-centric techniques for accomplishing model management tasks. On the one hand, integrating the intentional aspect of relating models into model management we can achieve better clarity and expressiveness, facilitating comprehension and analysis. On the other hand, by making explicit the ways that artifacts are related, and associating

this to a specific relation type metamodel, we facilitate the validation of consistency and the ensuring of robustness for model-based development processes.

In this chapter, we present the key points and the basic mechanics of the approach. In Chapter 5, we look more closely at how our approach can be employed to facilitate model management with a focus on the automotive domain.

4.1 Relations between models

In Section 3.2.1 we presented how mappings between models are defined in a macromodeling setting. We use the term "mapping" to refer to the generic concept of relating two models, while in this context we use the term "relation" to refer to a mapping conceived with respect to a particular *intent*.

As is the case for mapping models, relationship models must also conform to a relation type. The definition of a relation type corresponds to the acknowledgment of the role that it is supposed to play in the development process. More specifically, a relation type is characterized by:

1. a purpose, i.e. what the relation is about
2. its meaningful applications, i.e. to what kinds of models it can be applied
3. a set of rules for determining the well-formedness of relationship models

With these in mind we introduce a metamodel for defining relation types. The metamodel consists of the diagram shown in Figure 4.1, along with the additional OCL constraints shown in Figure 4.2. Relation types which have the aforementioned characteristics can be defined as instances of this metamodel.

As indicated by the metamodel and its first associated constraint, we differentiate between abstract and concrete (non-abstract) relation types. Abstract relation types are explicitly not associated with any endpoint model types and relator metamodels,

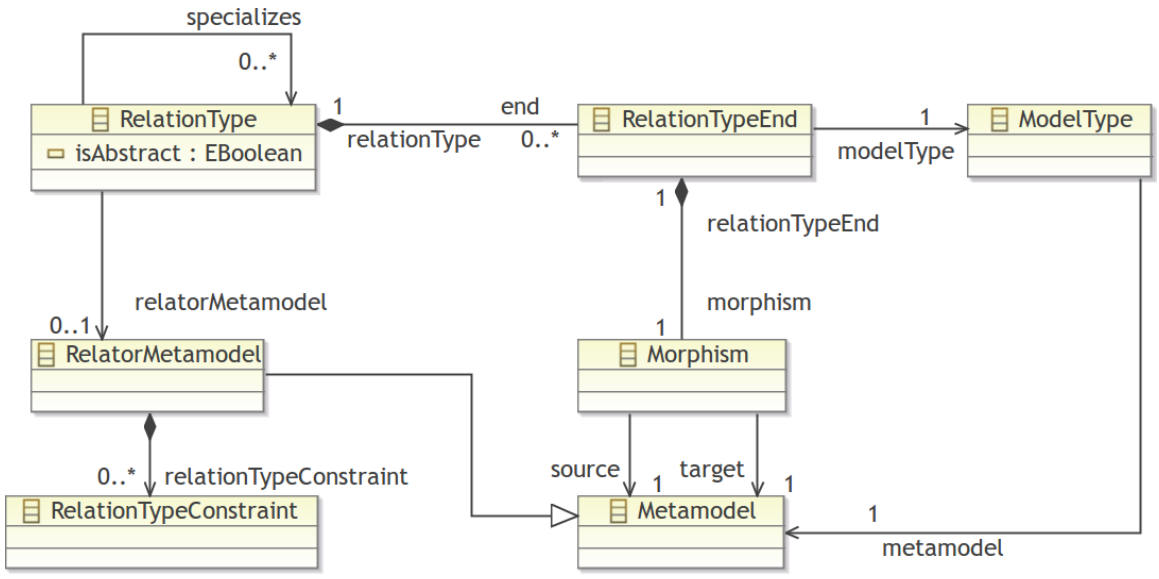


Figure 4.1: Metamodel for defining relation types.

context RelationType **inv**:

(self.isAbstract = true implies

(self.end−>isEmpty() and self.relatorMetamodel−>isEmpty()))

and

(self.isAbstract = false implies

(self.end−>notEmpty() and self.relatorMetamodel−>notEmpty()))

context Morphism **inv**:

self.source = self.relationTypeEnd.modelType.metamodel

and

self.target = self.relationTypeEnd.relationType.relatorMetamodel

Figure 4.2: Correctness constraints for defining relation types.

while concrete ones are required to be associated with at least one endpoint model type and some relator metamodel. This is enforced by the first OCL invariant. Abstract types serve as the declaration of an intention for relating models; concrete subtypes for specific application scenarios can be defined by specializing them to create a relation type hierarchy.

Concrete relation types are associated with one or more endpoint model types. The various application scenarios for an abstract relation type are thus made concrete by creating concrete subtypes of it which are tied to specific endpoint model types. A concrete relation type is associated with a relator metamodel, where the endpoint model types are represented via metamodel morphisms, as described in Section 3.2.1. The second OCL invariant ensures that Morphism elements are associated with the correct source and target metamodels. The rules necessary to specify the correctness criteria for the concrete relation type are expressed as `RelationTypeConstraints` associated with the type's relator metamodel.

The concepts discussed above are described in more detail in the following sections.

4.2 Abstract relation types

As we described earlier, we introduce the notion of *abstract relation types*, to represent the first characteristic of relation types described in Section 4.1, i.e., that relations represent a purpose. For example, we can declare the abstract relation type `Refinement` (Figure 4.3), to represent the situation where a model captures more concrete design decisions than some other model [29].

We declare the abstract relation type `Refinement` as a specialization of the most generic relation type, which we name `Relation`, that represents the abstract situation where any number of models of any kind are somehow related.

`Refinement` itself is not tied to any particular arity, model type, metamodel or se-

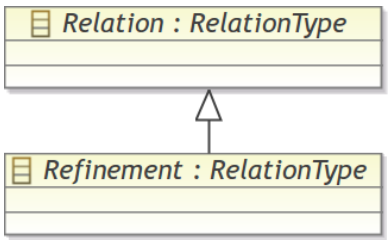


Figure 4.3: Defining the abstract relation type `Refinement`.

mantic constraints. In essence, the abstract relation type defines a *family* of relation types, all of which correspond to the notion of a model capturing more concrete design decisions than some other model. Each of the relation types in that family inherits this semantic definition from it, adding the details needed for particular scenarios, e.g. for refinement relationships between state machines.

Abstract relation types therefore provide us with a generic way to group together relation types that have the same purpose. This way, we can differentiate between the various ways that two models can be related. For example, two state machines can be related in various ways, such as refinement, evolution, naming consistency etc. Moreover, model transformations can also be represented as a relationship between the input and output models. Using abstract relation types for each of these cases we can treat each of them accordingly.

4.3 Concrete relation types

To represent a particular application scenario for some relation type, we need to specialize the relevant abstract relation type for it. This way we realize the second characteristic of relation types discussed in Section 4.1, i.e. that there are particular model types with which a relation type can be used.

In our example, we create the concrete relation type `RefinementSM-SM` to capture the situation where a state machine refines another state machine. To do this we need

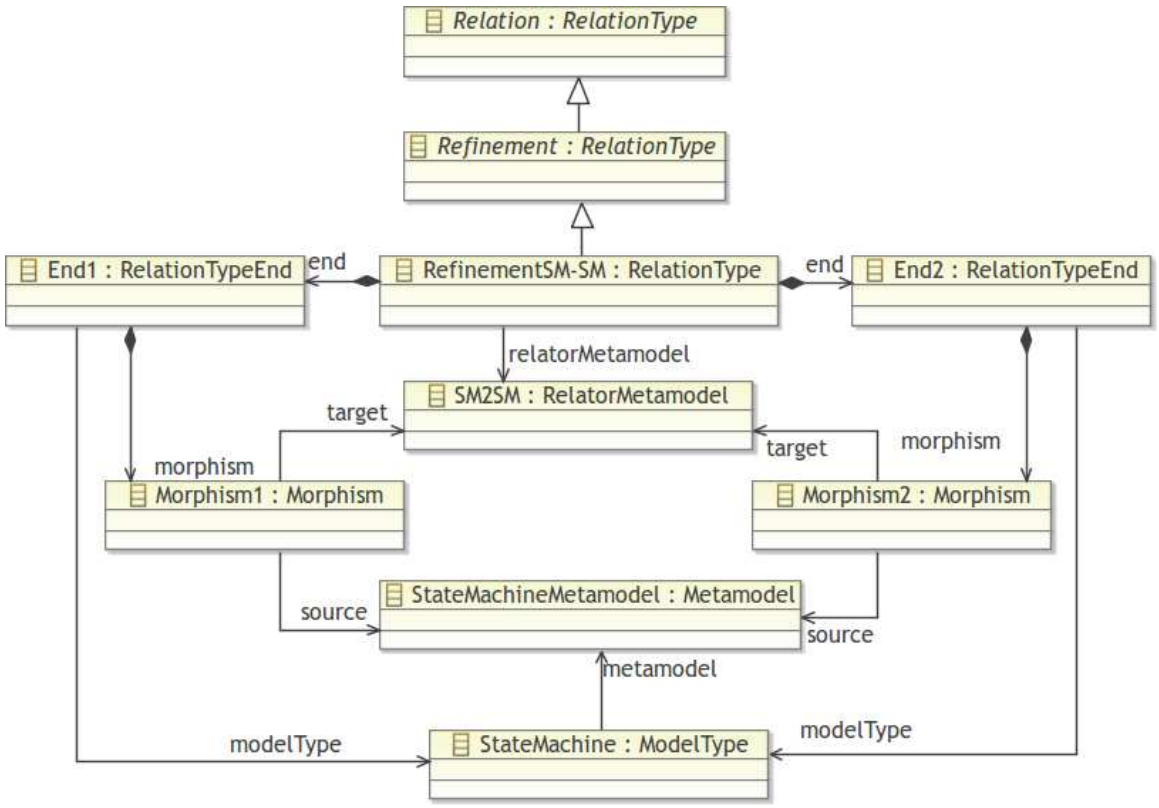


Figure 4.4: Defining the concrete relation type `RefinementSM-SM`.

to reference the relevant metamodels and create the appropriate metamodel morphisms (see Section 3.2.1). The result can be seen in Figure 4.4.

As can be seen in the diagram, the concrete relation type `RefinementSM-SM` specializes the abstract `Refinement` relation type, described in the previous section. While the latter is not associated with any endpoint model types, the former is explicitly associated with the `StateMachine` model type for both relation endpoints. In particular, we explicitly associate the `RefinementSM-SM` relation type with the `SM2SM` relator metamodel, shown in Figure 4.5, and we declare morphisms between `SM2SM` and the `StateMachine` metamodel (see Appendix, Figure A.3).

In our example, the model type of both relation type endpoints is the same, i.e., the state machine model type. Relation types with different model type endpoints can be defined in the same way (examples of that can be found in Chapter 5).

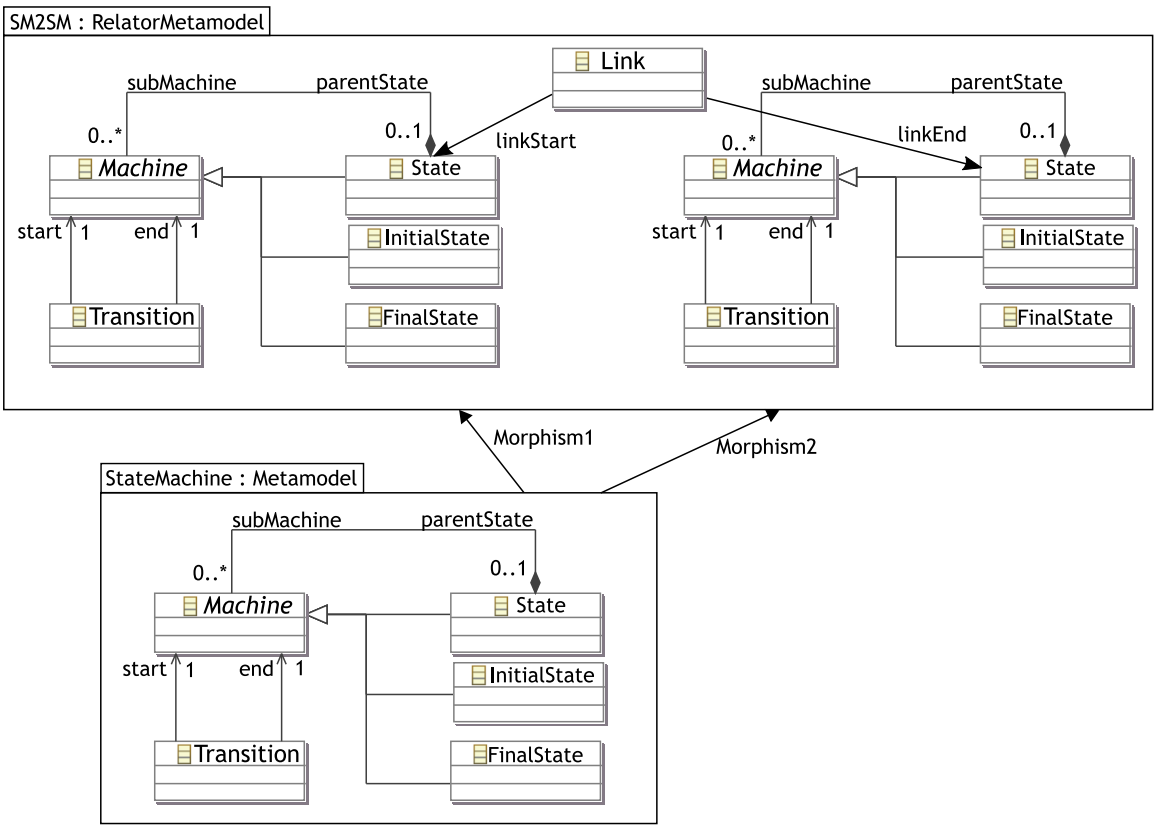


Figure 4.5: The relator metamodel `SM2SM`, shown with the morphisms relating it to the State Machine metamodel.

It is evident that to create definitions for other members of the `Refinement` relation type family, we only need to create concrete subtypes of it by declaring all the necessary elements. For example, to create the refinement relation type between UML Component Diagrams and Algorithm Block Diagrams, one would need to declare the `RefinementCD-ABD` concrete relation type, along with the appropriate morphisms from the Component and Algorithm Block to the `CD2ABD` relator metamodel.

4.3.1 Constrained mappings

Concrete relation types are tied to a specific relator metamodel. In our example, the `SM2SM` metamodel, shown in Figure 4.5, is the relator metamodel for generic mappings between State Machines. It is generic in the sense that it carries no particular semantics

context Link **inv**:

self.linkStart.name = self.linkEnd.name and

self.linkStart.subMachine−>isEmpty() and self.linkEnd.subMachine−>notEmpty()

Figure 4.6: Constraint over `SM2SM` for the `RefinementSM-SM` relation type.

other than describing there can exist links from the elements of one snapshot of the StateMachine metamodel to elements of another.[1]

As was described in Section 4.1, the third characteristic of relation types is that they are associated with a set of well-formedness rules. One way to realize this is by creating constraints over the relator metamodel. The idea here is to base the definition of relation types on extending the generic mapping model types described in Section 3.2.1 by means of constraining their metamodels. The additional constraints that are added to generic mapping types correspond to the semantics of the role that any particular relation type plays in the development process.

With regard to any set of model types that can be related in any way, we can construct a generic mapping in the form of a relator metamodel. For each particular relation type that we want to implement, we can create specific constraints over their generic mapping type, by elaborating the set of relator constraints Φ_{RC} of the relator metamodel, defined in Section 3.2.1.

In the case of our example, we implement a *very* simplistic notion of refinement of state machines, where we consider a simple state in the source model to be refined by a state in the target model if the latter is a complex state, i.e. the superstate of a more specific state machine, and thus "captures more concrete design decisions". The relation type is not a refinement transformation but rather a declarative means to define correspondences between models that are part of the same refinement process. The OCL

[1]We have kept the metamodel shown in Figure 4.5 very simple in that we allow for links to only be declared between `State` elements. This is done purely for simplicity and demonstration purposes: in reality, these generic relator metamodels can allow links to be created between arbitrary elements.

constraint for this semantics is presented in Figure 4.6.

The OCL invariant is built around the `Link` element type of the relator metamodel. For each link, the invariant requires that the name of the refined state is preserved. Moreover, for each link the constraint specifies that its source element is a simple state by demanding that its `subMachine` association (see the relator metamodel, Figure 4.5) is empty. Simultaneously, the constraint requires that the link's target element is a composite state by demanding that the same association is not empty.

A developer declaring a refinement relationship between two state machines by creating a relation model as an instance of this type, is required to create links from the states in the source model that are refined to complex states in the target model. The type's correctness rule enforces the semantics of our naive refinement by imposing a constraint on the these links that ensures that simple states get refined to complex ones. Performing validation of the relationship model with respect to its type enables the developer to determine whether the target model is indeed a proper refinement of the source model. We describe a similar validation scenario in more detail in Section 5.1.3.

4.4 Relation types revisited

The above discussion can also be viewed from a different perspective. The three characteristics we listed in Section 4.1 can be understood to define a plane. On the one axis lie the various purposes for which we conceive that models can be related (1st characteristic). On this axis we can put the various abstract relation types. On an orthogonal axis lie the various mappings that can exist between model types (2nd characteristic). On this axis we can put the various generic mapping types. At the points of the plane where the two meet, we have concrete relation types, which must be defined with respect to some well-formedness rules over the respective mapping type (3rd characteristic).

We can visualize such a grid of relation type hierarchies and mapping types, as shown

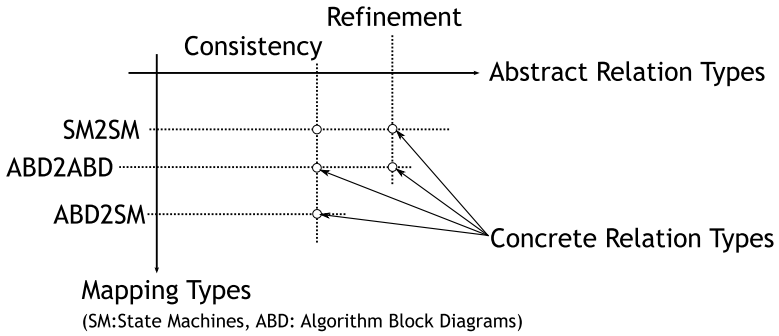


Figure 4.7: Relation types and mapping types (the ordering carries no meaning).

in Figure 4.7. Relation type hierarchies start from an abstract relation type, not associated with any particular application scenario. Mapping types can be created for any set of models that can be related in any arbitrary way. At the intersection of these two, we can create concrete relation types, i.e. relation types with respect to particular models, by adding relation type-specific constraints to the generic mappings.

We can even envision hierarchies of mapping types, which can include mappings between abstract model types, in which case relation types can be thought of as having a polymorphic character.

4.5 Model management with relation types

The mechanism for declaring relation types that we described so far can be employed in a macromodeling environment to achieve various model management tasks. Making the intended role of relations between particular types of models explicit in such an environment can enhance our ability for managing models in a variety of ways.

From one point of view, macromodels built with the typed relationships that we described in this chapter convey rich semantics. This semantic weight, along with the structuring potency of macromodeling can improve our ability to capture and comprehend how a set of models constitutes a system (or why it does not).

At the same time, by employing the relation types' semantics, which have been de-

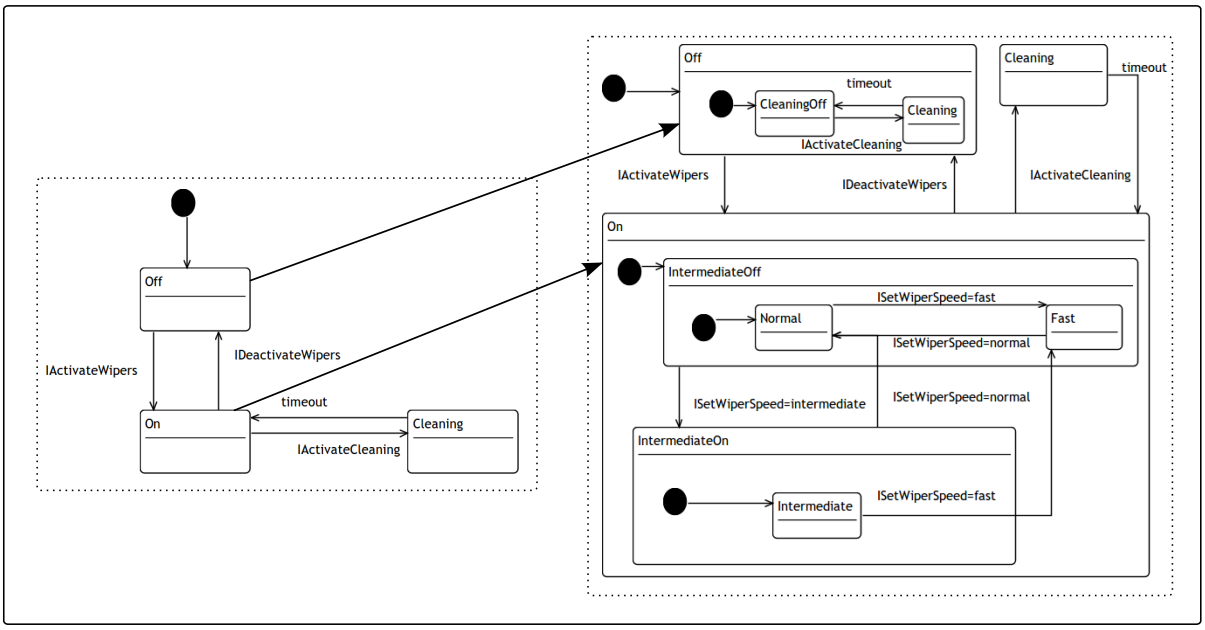


Figure 4.8: Declaring a refinement relationship (instance of the `RefinementSM-SM` relation type) for the WipersStory state machine shown in Figure 2.5.

fined in terms of metamodel constraints, we can do consistency checking of macromodels. Relationships can be verified as instances of their relation types, and the result of this verification can be used to draw conclusions regarding whether the related models are properly defined, revealing potential inconsistencies and so on. Drawing from the WipersStory scenario, we could for example use the `RefinementSM-SM` relation type to verify that two versions of the controller statemachine constitute a valid refinement.

As shown in Figure 4.8, an instance of the relation type `RefinementSM-SM` is declared for two state machines. Simple states of the source model of the relationship, representing the earlier version, are mapped to composite states of the target model which represents a subsequent version of the statemachine. This mapping is achieved by appropriate `Link` elements.

Verifying the relationship model in Figure 4.8 entails checking it for conformance to the `SM2SM` relator metamodel and the relevant relation type correctness constraints. In the case of this instance of the relation type, the check for conformance to the relator

metamodel succeeds as both constituent models are valid state machines, and all `Link` elements are between `State` elements, as specified by the relator metamodel. Additionally, validation of the relation type OCL constraint (see Figure 4.6) also succeeds as for both `Link` elements the source `State` elements are simple states while the target `State` elements are composite states.

In general, apart from checking consistency, relation types can be used to achieve greater versatility of expression. As relationship models are typed, we can create signatures for operators that act upon them to create new relationship models. In this manner, new relationships can be inferred, which can represent non-evident relationships between previously unrelated aspects of the overall system. The types of the input and output relationship models can be used to define the signature of such operators, effectively opening the space for creating an algebra of operators. Additionally, the declared type of the output can be used to verify it, and the result can reveal possibly hidden inconsistencies.

Furthermore, this conceptual framework can be employed as a development guide for the software system at hand. Relations between existing and planned or imagined models can be employed to capture the structure-to-be, while at the same time, as demonstrated in [40], certain models can be automatically inferred using model finders such as Kodkod.

In the following chapter we attempt to demonstrate some of these use cases.

Chapter 5

Analysis

Relationship types, when employed in a macromodelling context, can facilitate the carrying out of various model management tasks. In the following we present two cases where relationships can be used to guide such tasks.

In particular, in Section 5.1 we demonstrate how our relation-centric approach can be employed in order to check the consistency of sets of models, showing this with examples from the WipersStory example, described in Section 2.2. Additionally, in Section 5.2, we discuss composition operators and and demonstrate the composition of typed relation models, again with an example from WipersStory.

For simplicity, we limit ourselves to examples of relationships between (at most) two models, however, the same approach can be generalized to higher arities. Also for compactness of presentation, some diagrams in this chapter are presented in a simplified form, with obvious elements such as labels omitted.

5.1 Consistency checking

In a development process that involves a significant amount of models, as is the case in the automotive domain, it is imperative to ensure that the various models are consistent with each other. Inconsistencies can arise from a variety of sources, as models

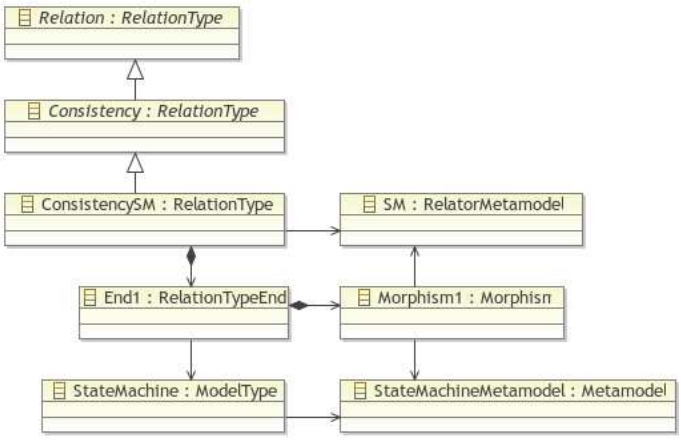


Figure 5.1: Declaring the unary consistency relation type for state machines.

are produced in a variety of stages in the development process, in different versions and to represent a system from different perspectives. By employing the relation-based approach we described in Chapter 4, developers can explicitly describe why and how various model types can be related. Consistency checking is therefore achieved by verifying that the relation type semantic rules are satisfied for the relationship between the models of interest.

The first step is to define an abstract relation type `Consistency` as a specialization of the generic `Relation` type, as was done with the `Refinement` abstract relation type, shown in Figure 4.3. This abstract relation type is therefore conceived as defining the family of relation types which represent the situation where some models are declared as being related with the intent of saying that they are supposed to be consistent with each other.

5.1.1 Unary consistency

In order to demonstrate the expressiveness of our approach, we begin at a simple level where we consider the unary consistency relation type. The semantics of this relation type is that a model is consistent with its declared type. Verifying a relationship conforming

to this type is equivalent to checking a model's conformity to its metamodel.

For a given model type, e.g., for state machines, the relator metamodel of the relation type `ConsistencySM` coincides with the metamodel of the model type. Figure 5.1 shows the declaration of `ConsistencySM`, following the same rationale as that of Figure 4.4, only in this case there exists only one `RelationTypeEnd` element.

More specialized unary consistency relation types can be used to verify that a model is in fact conformant to a subtype of a model type. As discussed in Section 3.1, for a given model type, subtypes are constructed by injecting additional constraints to its metamodel. In view of our relation-centric approach, unary consistency relation types can be used to verify whether models conform to desired subtypes, if they are defined with correctness type constraints corresponding to the subtypes' constraints.

Considering the case of state machines, we assume the general state machine type, associated with the metamodel given in the Appendix Figure A.3, without the OCL invariant limiting the number of outgoing transitions, shown in the same figure. The automotive domain-specific statemachine type, which was discussed in Section 2.3.3, is a subtype of this generic state machine type, with the addition of the OCL invariant.

In order to enable checking whether a particular state machine conforms to the automotive domain-specific subtype, we define the unary relation type `ConsistencyAutoSM` by injecting the OCL invariant of Appendix Figure A.3 into the relator metamodel of the `ConsistencySM` relation type. Checking if an arbitrary state machine conforms to the automotive domain state machine type is accomplished by creating an instance of the `ConsistencyAutoSM` type and validating it, as discussed in Section 5.1.3.

5.1.2 Consistency between pairs of models

Moving to less simple cases, we look into consistency relationships between two (or more) models. We note that various *kinds* of consistency can exist between models. This reflects the fact that, when checking the consistency of a set of models, we are usually

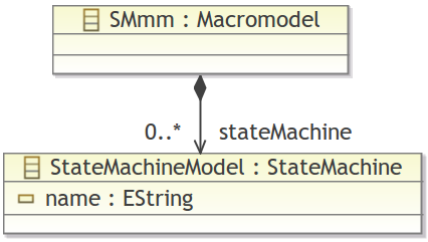


Figure 5.2: Metamodel for `SMmm` macromodels.

doing so with a particular intent in mind. Reflecting the 1st characteristic of relation types discussed in Section 4.1, we are inclined to organize these more subtle intentional approaches by further elaborating the Consistency relation type hierarchy with abstract relation types, extending the root `Consistency` abstract relation type.

For example, in WipersStory, we can examine the case where we are interested in making sure that Interfaces from the Signal Diagram (see Section 2.3.2) are used consistently across models. We therefore define the abstract relation type `IConsistency`, i.e., "interface consistency". A different kind of consistency is making sure that whenever an artifact needs to be elaborated by another model, that model exists. We therefore define the abstract relation type `DConsistency`, i.e., "declaration consistency".

We proceed to create concrete relation types from these abstract ones. In particular, we define relation type `IConsistencySD-ABD`, to capture interface consistency relations between signals and algorithm block diagrams. The relation type is directional, in the sense that its semantics indicate that all interfaces in the signals diagram are employed in the algorithm block diagram. The opposite relation type `IConsistencyABD-SD`, which indicates that all interfaces in an algorithm block diagram have been declared in a signals diagram, can be defined by reversing the start and end association links of the relator metamodel `Link` class and similarly reversing the semantic constraints.

We also define the type `DConsistencyABD-SMmm` to capture declaration consistency relations between algorithm block diagrams and sets of state machines. We can represent sets of state machines by using macromodels (see Section 3.2.2), and in particular the

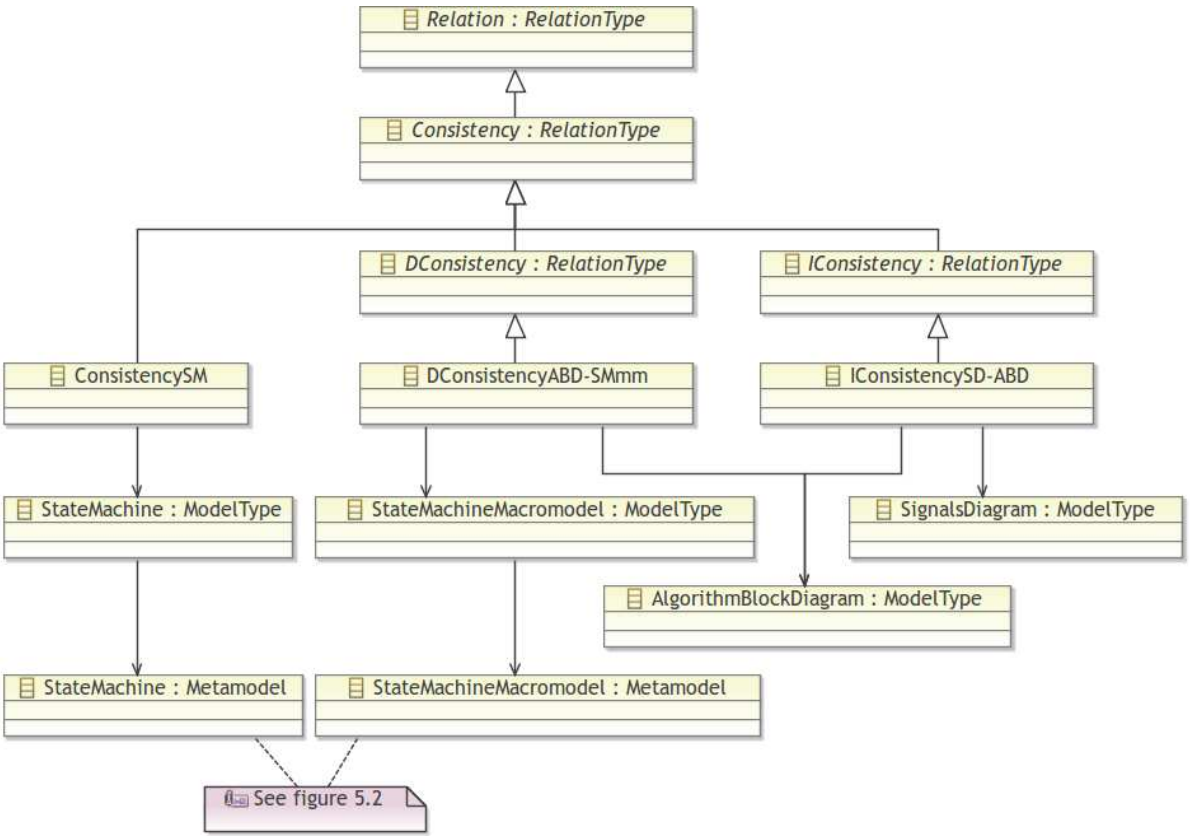
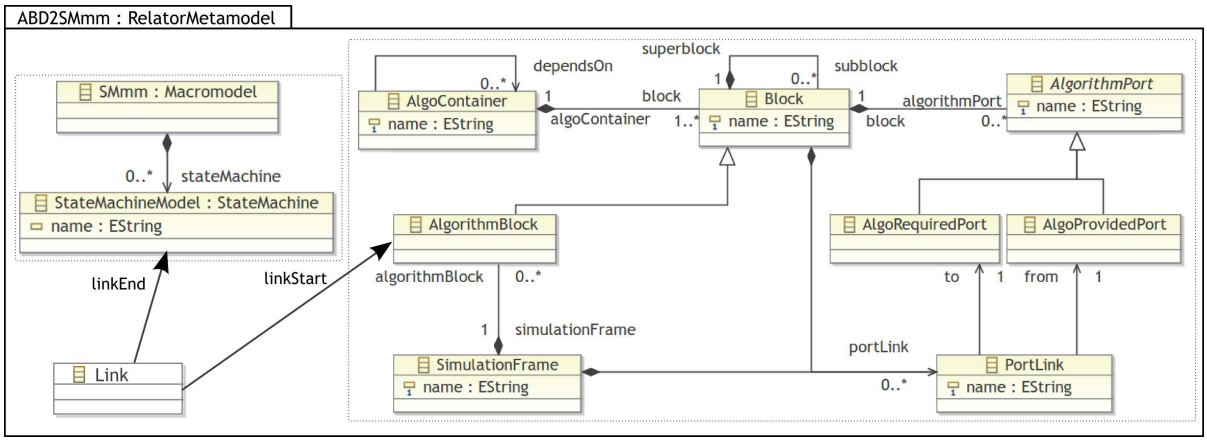


Figure 5.3: Declarations of the various consistency relation types.



context l : Link **inv**:

AlgorithmBlock.allInstances()−>forAll(

a | StateMachineModel.allInstances()−>exists(

s | a.name=s.name and l.start=a and l.end=b))

Figure 5.4: `DConsistencyABD-SMmm` relator metamodel and relation type constraints.

macromodel model type `SMmm`, seen in Figure 5.2, which is simply a model whose elements represent individual state machines. The overall picture for all relation type definitions can be seen in Figure 5.3, where each relation type is shown at its respective place in the relation type hierarchy, associated with its endpoint model types.

For each of the two relation types that we defined, we also explicate the necessary well-formedness rules, as OCL constraints over their relator metamodels.

In particular, the relator metamodel for the `IConsistencySD-ABD` relation type, seen in Figure 5.5, consists of copies of the signals diagram and algorithm block diagram metamodels. As algorithm block diagrams only allow the declaration of ports, we indirectly link interfaces by linking their respective ports. Thus the relator metamodel specifies that links can be established between `AlgorithmPort` elements.

To explicate the semantics of the interface consistency relation type, we take advantage of the fact that for a signal to be employed in the algorithm block diagram, at least one port must be declared as associated with it in the signals diagram. Therefore, the

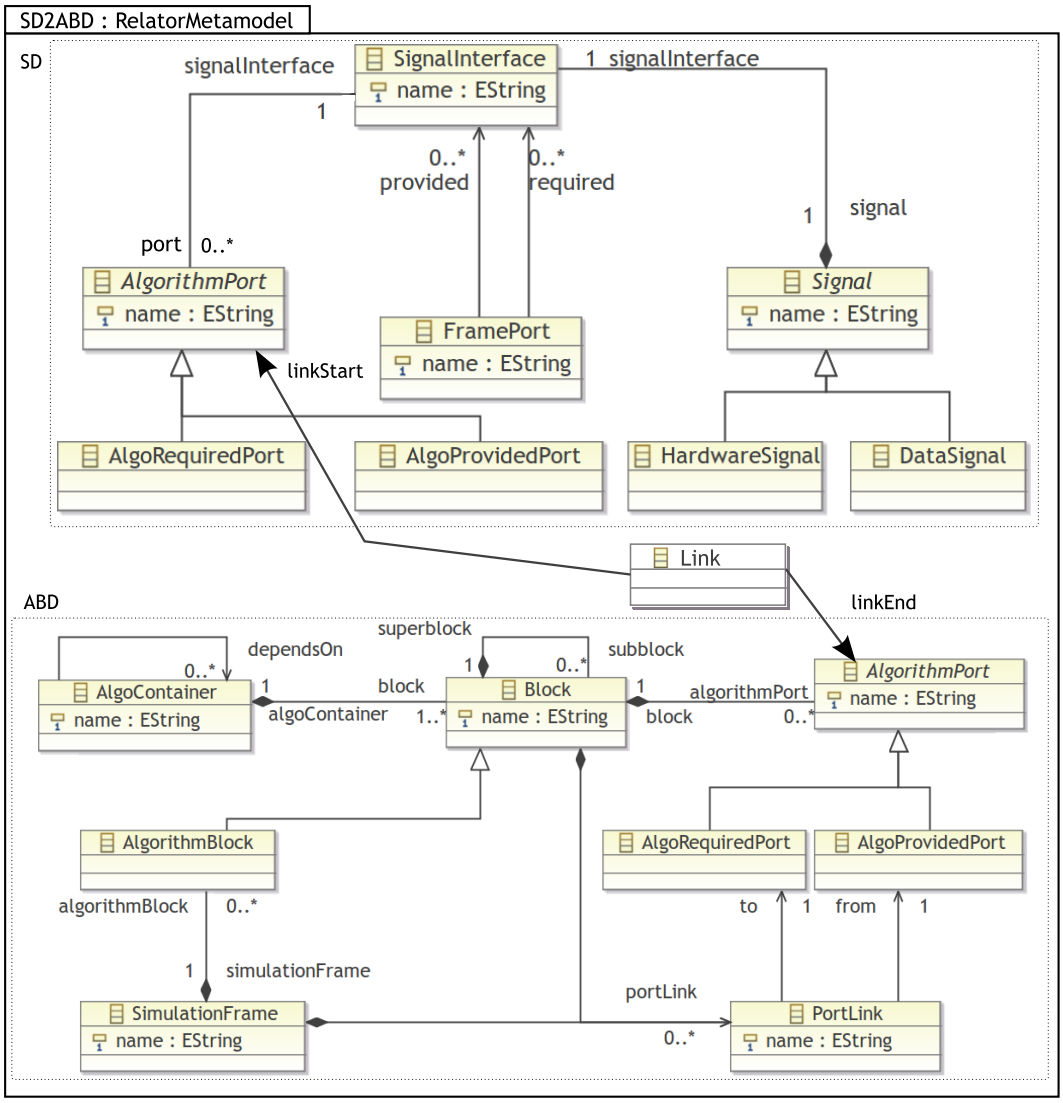
accompanying OCL constraint requires that in relator models conforming to this relation type, every interface in the signals diagram must be associated with at least one port, to reflect the requirement that all declared interfaces are utilized in the algorithm block diagram. Additionally, for every port defined in the signals diagram the OCL invariant requires that there exists a port of the same name in the algorithm block diagram that is associated with it via a `Link` element.

Similarly, the relator metamodel for the `DConsistencyABD-SMmm`, seen in Figure 5.4, consists of copies of the metamodels for the algorithm block diagram and state machine macromodel model types. Links can be established between algorithm block and statemachine elements of the two models. The OCL invariant requires that for every algorithm block element in the algorithm block diagram there exists a state machine of the same name, associated to it via a `Link` element. (Note that the invariant does not preclude two algorithm blocks from being linked to the same state machine.)

5.1.3 Application

Having defined the various relationships, we can now apply them on our WipersStory example. A relator model conforming to the `DConsistencyABD-SMmm` relation type can be instantiated for the algorithm diagram and the set of state machines created by the subcontractor for the wipers system. The diagram seen in Figure 5.6 represents this relator model. As can be seen in the diagram, the algorithm blocks of the WipersStory algorithm block diagram are linked with state machine model elements of the macromodel containing the set of all state machines created by the subcontractor company of our example.

Similarly, a relator model conforming to the `IConsistencySD-ABD` relation type, can be instantiated for the signals and algorithm diagrams created by the subcontractor in WipersStory. The diagram seen in Figure 5.7 represents this relator model. As can be seen in the diagram, the port declarations from the signals diagram are linked with



context l : Link **inv**:

SD!SignalInterface.allInstances()−>forAll(

s | s−>count(port) >= 1 and

ABD!AlgorithmPort.allInstances()−>exists(

a | a.name=s.port.name and l.start=a and l.end=s

))

Figure 5.5: `IConsistencySD-ABD` relator metamodel and relation type constraints.

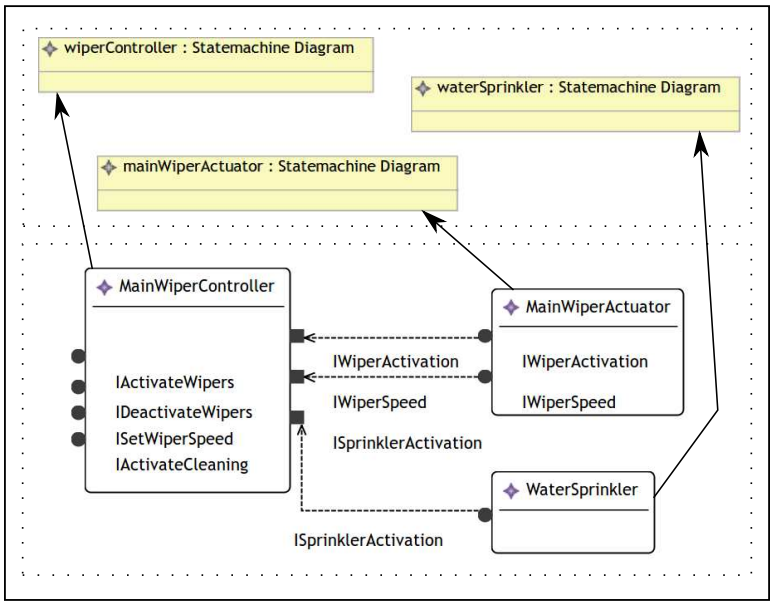


Figure 5.6: Instance of the `DConsistencyABD-SMmm` relation in WipersStory.

ports in the algorithm block diagram. For signal declarations corresponding to signal communications internal to the component, there exist pairs of ports, which are linked with corresponding port elements in the algorithm block diagrams. Conversely, for signal declarations that refer to communications external to the component, there exist single port declarations which are linked to ports of the algorithm block diagram at the points where the relevant interfaces are exposed.

In order to check the kinds of consistency that were discussed above between these models, we need to validate these two relator models as instances of their respective concrete relation type metamodels and constraints. This can be accomplished with an appropriate tool that supports the validation of models, and therefore relator models, with respect to their declared type (see also Chapter 6).

In our example the validation of the declaration consistency relationship produces an error as there exists a naming inconsistency between the `MainWiperController` algorithm block and its corresponding state machine, named `wiperController`. A similar error would have been produced if for some of the algorithm blocks there wasn't a link

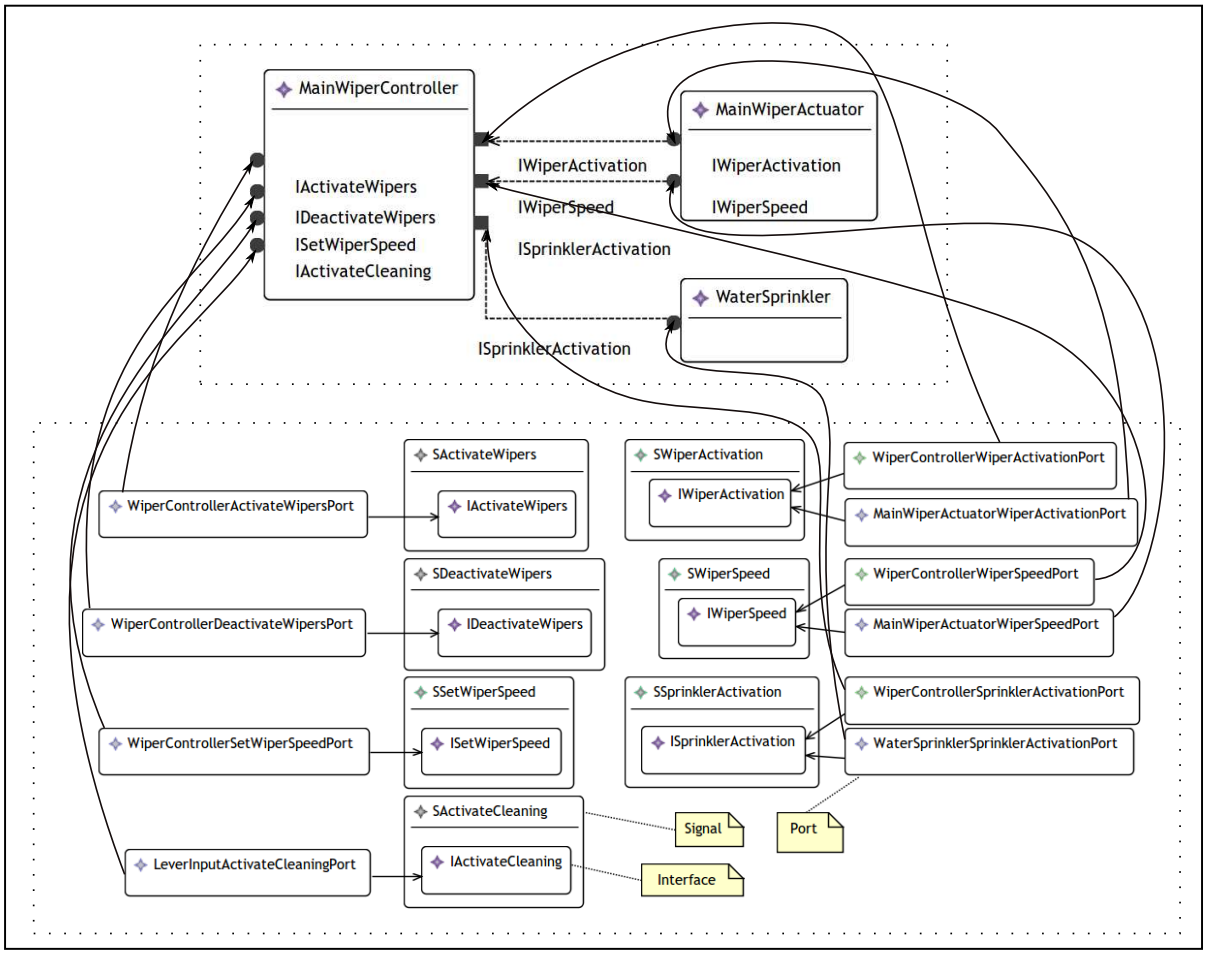


Figure 5.7: Instance of the `IConsistencySD-ABD` relation in WipersStory.

to a state machine in the macromodel. If this inconsistency is corrected, subsequent validation attempts will be successful, as all algorithm blocks will be linked with a corresponding state machine of a proper name, and therefore the purpose of the relating the two models with a declaration consistency relationship type will have been achieved.

Similarly, validation of the interface consistency relator model will produce no errors. This will happen because, one the one hand, all interface declarations are associated with at least one port declaration. On the other hand, for all ports defined in the signals diagram there exists a link to a port declaration in the algorithm block diagram with the appropriate name.[1] Hence, the purpose of relating the two models with an interface consistency relationship will also have been achieved.

5.2 Composition

Another application of the relation-centric approach is to use composition of relationships to create new ones. The utility of such an application can be twofold. On the one hand, we can compose relationships in an exploratory manner, to investigate whether models are related in a particular way. On the other hand, we can compose relationships in order to infer implicit knowledge about non-evident relationships between models to potentially uncover previously unseen underlying inconsistencies. Moreover, composition of relationships can be used to automate the task of structuring a set of models, in the sense of explicating only necessary relationships and inferring the rest.

The discourse on consistency presented in the previous section is therefore of particular interest. By composing relationships to infer new ones, we can, for example, verify that a particular chain of relationships indicates a desired property, or that a particular sequence of transformations has a desired result. This perspective can be important in view of the software lifecycle, where a change occurs in some model and the need arises

[1] In the algorithm block diagram, the ports are labeled with the name of their respective interface, not their actual instance name.

for each $link1 : Link \in firstRelatorModel$

do for each $link2 : Link \in secondRelatorModel$

do if $link1.linkEnd = link2.linkStart$

then
$$\begin{cases} new\ link3 : Link \\ link3.linkStart := link1.linkStart \\ link3.linkEnd := link2.linkEnd \\ outputRelatorModel.add(link3) \end{cases}$$

Figure 5.8: Generic composition algorithm.

to verify whether relationships that held for its previous version still apply for the new version.

Our approach of explicating relation types provides grounding to start developing a theory of composition operators (and operators on model relationships in general). In parallel with the relation type hierarchy, various composition operators can be defined, each with its own signature. We can envision a composition operator hierarchy, with a generic composition operator with the signature

$$\texttt{Composition} : \texttt{Relation} \times \texttt{Relation} \rightarrow \texttt{Relation} \tag{5.1}$$

as the root element.

Such a generic composition operator can employ the simple algorithm presented in Figure 5.8. In particular, the composition algorithm follows each pair of links in the two related models for which the target element of the first link coincides with the source element of the second, and creates a new link from the source of the first link to the target of the second link in the new relator model.

We can create more specific composition operators for more particular relation types, that could potentially employ different algorithms. Such algorithms could take into account the particular types of the input and output models. The idea of creating relation type hierarchies, can be particularly useful in such cases, since the explicit definition of

relation types and their structuring in hierarchies provide the grounds to quickly identify the requirements for creating such composition algorithms. Type hierarchies additionally allow for flexible definition of composition operator signatures by taking advantage of subtype substitutability.

In general, each composition operator has a signature consisting of input relation types and a desired return relation type. The result of any relationship composition operator is always a relator model. This relator model must conform to the relator metamodel and type constraints of the relation type designated by the operator's signature. Successful validation of this output relator model indicates that it properly relates the endpoint models.

However, unless otherwise proven for the particular composition operator signature, it is not guaranteed that the output relator model will conform to the designated relation type with respect to models that it connects. In other words, the set of relation models conformant to relation types is not closed under composition operators, in the algebraic sense. This allows us to use the potential success or failure of the validation of the inferred relationship to draw conclusions about the consistency of the set of models under scrutiny.

5.2.1 Application

We demonstrate the composition operator framework with a simple example from WipersStory. As discussed in Section 2.2, the automotive company creates a UML Component Diagram specifying the windshield wipers component as a black box with interfaces. The subcontractor creates a signals diagram, an algorithm block diagram and a set of state machines.

Assuming that the evident relationship between the component diagram and the interface declarations in the signals diagram has been explicated, we look into the scenario where the automotive company is interested in establishing that all of the specified interfaces are also declared and implemented in the subcontractor's implementation.

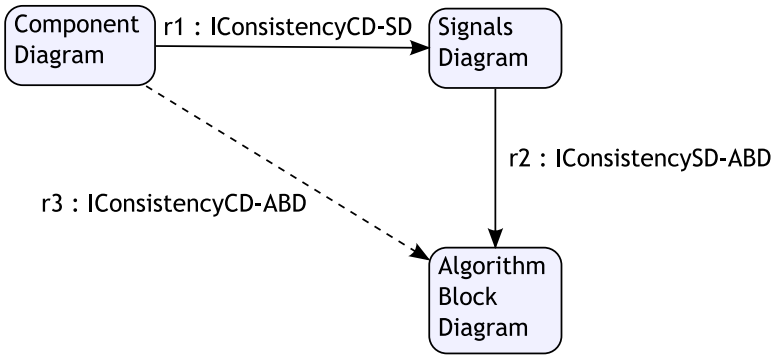


Figure 5.9: Overview of the WipersStory `IConsistency` composition scenario.

As seen in Figure 5.9, the appropriate concrete relation types need to be defined. In particular, apart from the `IConsistencySD-ABD` relation type, that was defined earlier (see Figure 5.5), the relation types `IConsistencyCD-SD` and `IConsistencyCD-ABD` are defined as sub-types of the `IConsistency` abstract relation type. Their relator metamodels can be found in the Appendix (Figures A.5 and A.6 respectively). The metamodels for these relation type follow the same logic as that for the `IConsistencySD-ABD` relation type, allowing the creation of `Link` elements between the elements in the endpoint metamodels representing interfaces, and requiring that all interfaces in the source endpoint model exist in the target endpoint model and are linked with `Link` elements.

For the particular scenario, the relator models r_1 conforming to `IConsistecyCD-SD` and r_2 conforming to `IConsistencySD-ABD` have also been instantiated. The diagram for r_1 can be seen in Figure 5.10. In the diagram, the interface elements in the component diagram are linked with the ports that correspond to interfaces with the same name in the signals diagram. The diagram for r_2 is depicted in Figure 5.7 and is discussed in Section 5.1.3.

The generic composition operator (5.1) can be directly used with the arguments being subtypes of the abstract relation type `Relation`, due to subtype substitutability. However, for demonstration purposes, we proceed to define a consistency composition operator, by specializing its arguments to create the new signature

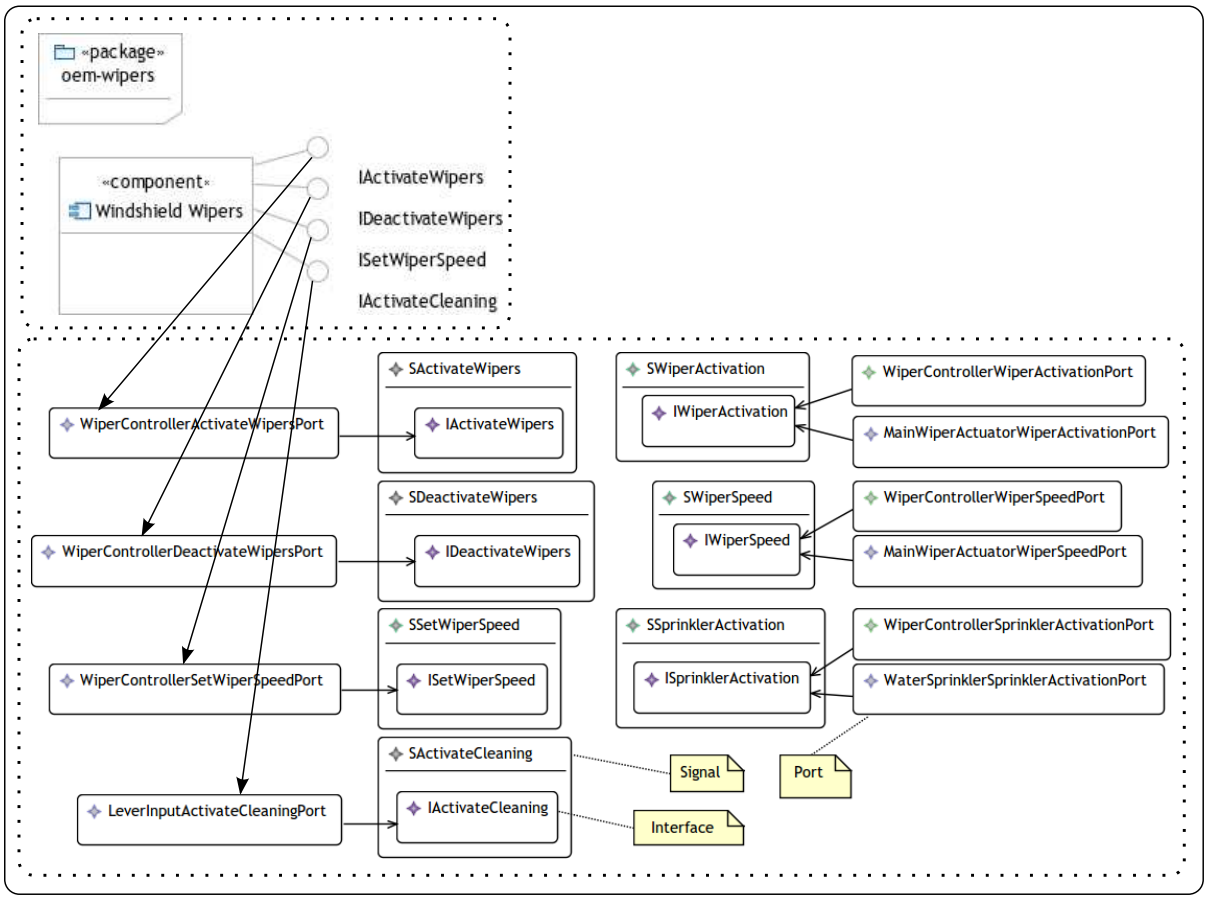


Figure 5.10: The r_1 relator model, an instance of `IConsistencyCD-SD`.

`Consistency` × `Consistency` → `Consistency`

This can be further specialized for specific use of the interface consistency abstract relation subtype:

`IConsistency` × `IConsistency` → `IConsistency`

If a complete, detailed, specialization of the composition operator is desirable, we can create a signature particularly for the desired types, to create the $\texttt{IComposition}_{CD-SD-ABD}$ operator, with the signature:

`IConsistecyCD-SD` × `IConsistencySD-ABD` → `IConsistencyCD-ABD`

This specialized operator could have an algorithm different than the generic composition algorithm shown in Figure 5.8, that synthesize the new relator model by referring to particular elements of the input relator metamodels, instead of generic `Link` elements.

For our example, however, the generic composition operator and its simple algorithm is sufficient. By executing the composition algorithm shown in Figure 5.8, we can compose the relator models r_1 and r_2. The resulting relator model r_3, shown in Figure 5.11, contains links from the interface elements in the component diagram to their corresponding ports in the algorithm block diagram.

As discussed earlier, it is not guaranteed that the resulting relator model of a composition operator will conform to the relation type designated in the composition operator's signature. Therefore, an additional step is required, in which the output relator model must be validated against the designated output relation type. In our example, r_3 must be validated against the metamodel of the `IConsistencyCD-ABD` relation type (see Appendix, Figure A.6).

Validating the r_3 relator model against its relation type be accomplished with relevant tooling, as discussed in Chapter 6. In the particular case at hand, validation would not produce any errors as all interfaces in the component diagram are linked to corresponding interface ports in the algorithm diagram of the appropriate name. It can therefore be safely said that for this particular composition the resulting model conforms to the

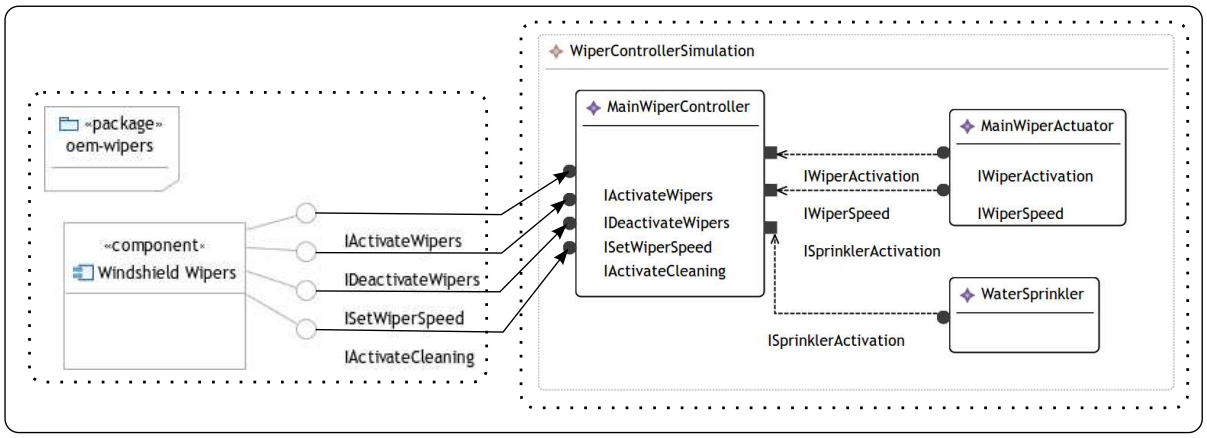


Figure 5.11: The inferred relator model r_3, an instance of `IConsistencyCD-ABD`.

designated `IConsistencyCD-ABD` relation type.

This result of the validation can be used by the automotive company to verify that the subcontractor's solution at least exposes all the necessary interfaces designated in the original specification. Moreover, by inspecting the resulting relator model, the automotive company can trace where the functionality for each interface is located in the algorithm block diagram.

In the case where the validation would produce errors, that would indicate either that some interface is missing from the set of interfaces exposed by the component or that there exits a naming inconsistency. Bug fixes could then be applied in the algorithm block diagram based on this error report in order to bring the related models in such a state that the various consistency relationships, including `IConsistencyCD-ABD`, hold between the various models of the wipers subsystem.

Chapter 6

Tool support

In Chapter 5, we demonstrated how our relation-centric approach can be employed in a macromodelling context to tackle various model management tasks, such as consistency checking, inferring implicit knowledge and automating the creation of composed relationships between models, whereas other applications of it can be envisioned, as was discussed in Section 4.5.

A software platform for practically applying the approach would have to fulfill a number of requirements. On a basic level, it should support macromodelling and be generic enough to be able to be deployed for various domains. For each particular domain, such as the automotive domain, it should provide the infrastructure to define model types, create structured collections of models conforming to these types and check the individual models against their type.

More particularly to our approach, it should provide for the definition of relation types, creation of instances of them for particular models and checking of the relationships against their types. And, with respect to Section 5.2, the framework should provide the infrastructure to define and run operators such as composition on relationships and enable the validation of the outputs.

With these in mind, we employed the *Model Management Tool Framework* (MMTF),

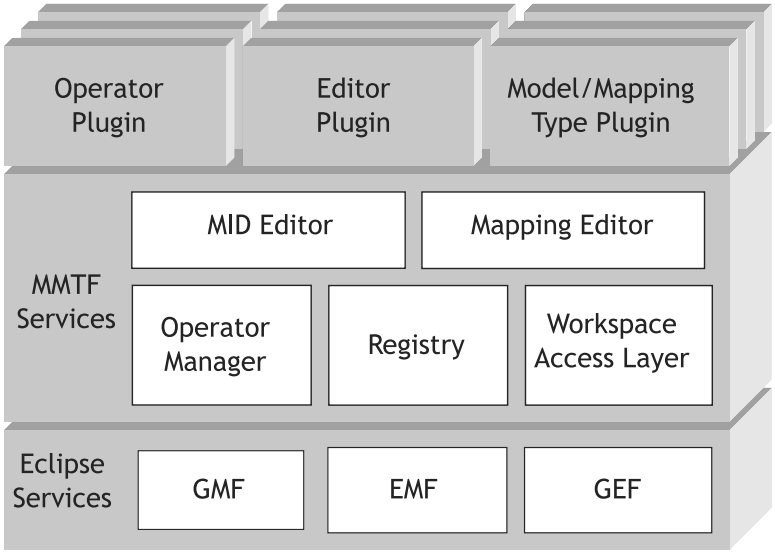


Figure 6.1: Overview of MMTF architecture.

introduced in [39]. MMTF is a macromodelling tool framework based on the Eclipse platform technologies[5], whose feature set satisfies the list of requirements discussed above.

More specifically, MMTF is a generic framework for macromodelling. It supports the creation of macromodels in the form of *Model Interconnection Diagrams* (MIDs). Moreover, it allows users to create particular deployments to cater to domain specific needs, by adding model type, relation type and operator plugins. In this vein we have created an automotive domain-specific deployment of MMTF, called *AutoMMTF* which contains plugins for the model types described in Section 2.3.

For the various model and relation types, MMTF can accept editor plugins created with the Eclipse Graphical Modeling Framework (GMF)[7]. This in turn means that all models and metamodels in MMTF are based on the Eclipse Modeling Framework (EMF)[4] technologies and therefore employ *Ecore* as their meta-metamodel.

MMTF supports creating subtypes of the model types for which full-fledged GMF plugins have been registered to it, by attaching to them constraints written in the Object Constraint Language (OCL)[33]. Models conforming both to full-fledged types, as well

as their to constrained subtypes can be verified using the EMF Validation framework. MMTF allows for validation of individual models, as well as for MIDs. In the case of MIDs, validation occurs recursively: the MID is considered valid if all its constituent element models (which can be themselves MIDs) and relationships are valid with respect to their particular type.

With respect to the specifics of our approach, MMTF supports creating relationships between models in the form of *mappings*. A generic mapping type, CommonMapping, is provided. Concrete relation types, as described in Section 4.1, can be implemented by creating constrained subtypes of CommonMapping, in the manner described in the previous paragraph.

Instances of the relation types, can be created in the MID Editor component of MMTF. The MID Editor allows users to create macromodels, by adding various typed models and creating mappings between them. A screenshot can be seen in Figure 6.2. Such relationship models, can in turn be verified with respect to their relation type, using MMTF's validation support.

Additionally, MMTF provides an infrastructure for declaring and running operators in the MID Editor. A composition operator, implementing the algorithm presented in Figure 5.8 is provided, alongside operators for a variety of uses, such as for name matching, inverting mappings etc. In MMTF, operators can be generic or tailored for specific model types. For example, for merging models, MMTF provides a generic structural merge operator alongside an operator specifically for the behavioral merge of state machines. Similarly, model type-specific composition operators, as discussed in Section 5.2, can be created. Additional composition operators, and operators in general, can be added to the framework via its plugin mechanism.

Tools comparable in purpose and functionality to MMTF can mainly be found in the context of the Epsilon [6] and AMMA [1] projects. As discussed in more detail in Chapter 7, these toolkits provide well-grounded support for a variety of model management

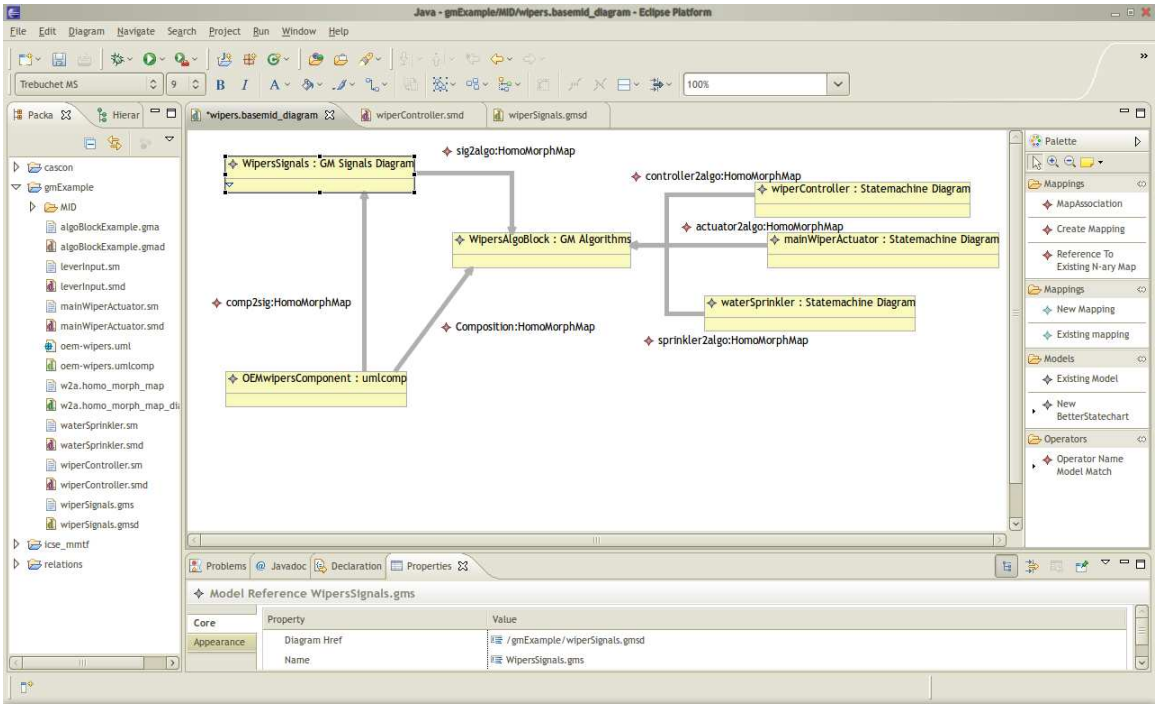


Figure 6.2: WipersStory in MMTF's Model Interconnection Diagram editor.

tasks, including good support for defining mappings between models, via the ATLAS Model Weaver(AMW) [2] in AMMA and the various languages built around EOL [28] in Epsilon.

However, there is no support defining explicit intention-specific relation types to which relationship models are required to conform. Moreover, to our best knowledge, other than sequential execution of transformations (which can be viewed as an implicit form of composition) these toolkits do not provide any support for composition of relationships. These tasks are made possible by MMTF via its mechanism that allows subtyping with constraints and its operator infrastructure.

Chapter 7

Related Work

Our work is in part inspired by the calls made by researchers in the domain of automotive software engineering, an area where model-based techniques are becoming increasingly significant, for integrated, seamless model-driven approaches [18]. We approach the problem from a model management perspective, and thus work related to what we presented here comes mainly from that research area.

In the context of model management, our work is closely related to research on multi-model systems. In this context, we can identify two major schools of thought: the Epsilon project [6] and the Atlas Model Management Architecture [1]. Epsilon aims to build a family of task-specific languages centered around the Epsilon Object Language (EOL) [28] with the aim to manage models independently of their metamodel. The approach of the AMMA project on the other hand is centered around the concept of "megamodels", an environment that considers models, metamodels, tools, services and other related entities as a whole [15].

In the context of the Epsilon project, the different kinds of relationships between models are treated with regard to the particular task at hand. The Epsilon Comparison Language (ECL) is employed to automate the process of identifying matches between models for purposes such as differencing, versioning and others [26]. The Epsilon Trans-

formation Language (ETL) provides support for model transformations [30] which can be considered a class of model relation types, while the Epsilon Merging Language (EML) provides support for model merging [27]. Additionally, the Epsilon Validation Language (EVL) provides the capability to manage inconsistencies between models. In the same research context of detecting inconsistencies across models, Kolovos et al. have also identified some conceptual relation types, akin to our line of thought of identifying relation types with regard to their purpose. This work, also has the added benefit of tackling not only the problem of detecting inconsistencies, but also the issue of repairing them [29].

In the context of the AMMA project, transformations are the main focus in relating models. The Atlas Transformation Language (ATL) [25] is implemented as a realization of the Object Management Group's Query-View-Transformation (QVT) specification [34], a language for specifying transformations and relations between models. The AMMA platform further includes the Atlas Model Weaver (AMW), with which users can specify typed links between model elements [21]. In the context of AMW, models are related by creating weaving models conforming to weaving metamodels created for particular application domains. Work on managing consistency under the umbrella of AMMA has been conducted by using ATL to transform the models under scrutiny to "problem models" [14].

Compared to these approaches, our approach attempts to capture the various kinds of relations at a higher level of abstraction. Our conceptual framework is allows for the handling of relationships in an abstract and generic manner which can be extended to represent arbitrary kinds of relation types. In this sense, our approach can provide a platform for integrating the various perspectives present in these lines of work. Moreover, our approach to conceptualizing relationships as conforming to relation types, opens up the space to adopt type theoretic approaches to various problems, thus achieving a degree of formalism at a high level of abstraction.

More generally, the problem of consistency checking has been studied extensively.

Some indicative literature includes [38] where two models are merged into one which is then checked against consistency constraints, [24] which looks into the problem from the perspective of traceability, [19] which focuses on extracting constraints from model transformations, and many others. Finally, regarding the issue of composition of composition of model mappings, related literature includes a study of model operators in [11], while in a wider context, which includes model transformation, related work has been done with chains of ATL transformations [20]. As discussed earlier, our approach can facilitate bringing such particular model management application tasks on a unifying platform, which can encompass these as well as other tasks in an integrated way.

Chapter 8

Conclusion

8.1 Discussion and evaluation

We presented an approach to model management based on the notion of defining and explicating relation types and then using them in a macromodeling context. We demonstrated the approach with examples from the automotive domain, an area where the use of model-based techniques is increasingly emerging as an industry norm, and is therefore faced with significant model management problems.

Our approach entails the formation of relation type hierarchies, by declaring abstract types that capture the intent for a particular way to relate models. By additionally specifying particular application scenarios and correctness for these scenarios, we can create concrete relation types that specialize the abstract ones for particular model types.

In Section 2.4, we listed the following requirements for an approach to model management:

1. It should enable developers to reason about how a set of modeling artifacts is semantically structured.

2. It should provide the infrastructure for validating this structure to expose potential inconsistencies.

3. It should facilitate the inference of implicit knowledge about the ways that models are dependent in non-evident ways.

With respect to the first requirement, in Section 4.2 we define relation types as tied to particular intentions and in Section 4.3.1 we require that the semantics of concrete relation types are explicated in the form of relation type constraints. In a macromodeling context, this enables developers to create typed macromodels containing typed models and typed relationships among them. This creates a semantic structure that captures the interdependencies among the various models. Furthermore, as discussed in Section 3.2.2, macromodels can be nested. Applying the same principles, relationships can be established between macromodels, and therefore it is possible to create semantically structured hierarchies of models, where the semantics are captured by the types of models and the types of relationships between them.

What our approach does not address is the issue of macro-relationships, that is, nesting relationships between models from different metamodels in a relationship bundle at a higher level of abstraction. Such a feature would enable developers to express complex relationships between *collections* of models, as opposed to between individual models.

Regarding the second requirement, we discuss in Section 4.5 how the well-formedness constraints attached to any concrete relation type can be used to validate the proper application of the relation type. As discussed above, the semantic structure of a set of models is constructed by creating macromodels containing typed models and relationships between them. All relationships between models are concrete relationships which are tied to specific semantic constraints as defined in Section 4.3.1. Validation of the semantic structure of a set of models is therefore a matter of recursively validating the macromodel which captures it with respect to the model and relation types of the elements that it contains. Additionally, in Section 5.1, we define relation types specifically to capture checkable consistency dependencies between models. Consistency relation types are a

additional powerful means to verify the consistency between models in a structured set, as they provide a way to explicitly specify which models should be consistent and how.

A potential shortcoming of our approach with respect to this requirement is that as validation is based on examining OCL constraints on model elements, the input to the user could be too low level, compared to the high level of abstraction to which we aim. Additionally, our approach does not address the issue of proposing potential fixes for any inconsistencies uncovered by the validation process.

With respect to the third requirement, we describe in Section 5.2 how typed relationships can be employed to define composition operators. Such operators can in turn be used for the composition of relationships that have already been explicated in a macro-modeling context, to automatically infer new relationships. Such inferred relationships expose implicit knowledge about non-evident dependencies between models. Checking such a relationship model against the relation type specified by the signature of the composition operator enables uncovering potential inconsistencies between the related models.

An issue with this approach to uncovering implicit inconsistencies is that it can only uncover *expected* inconsistencies, i.e., inconsitencies checkable by constraints already explicated for the relation type designated as the output of a composition operator. Our approach does not tackle the issue of composing the constraints of the input relation types of a composition to infer those of its output relation type. Such a technique could potentially uncover non-evident types of inconsistencies, that cannot be detected by our approach so far.

8.2 Future work

On a basic level, our approach to declaring and defining relation types opens up the space for approaching model management in a more systematic manner. We have presented

a few applications of such an approach, but we can envision a much larger application domain. As sets of models are structured around a particular purpose, defining relation types to capture their semantic structuring can facilitate greater comprehension of sets complex sets, in a context that can potentially encompass the whole of the software lifecycle, from requirements to code generation and deployment. We intend to further investigate such uses including creating traceability links, defining transformations, managing variability and deferred design decisions, guiding development and representing evolution.

At the same time, we find that it might be cumbersome for developers to need to create concrete relation types for all possible models that they encounter. For each concrete relation type, relator metamodels need to be created and the particular semantics for it must be expressed as constraints on it. An issue that therefore could be useful to investigate more would be to see how to integrate our work with a more generic approach, possibly through a mechanism such as the notion of generics found in Java, or templates in C++. Such an approach could potentially allow developers to automate part of the process by capturing the abstract semantics at a higher level of abstraction.

Moreover, as abstract relation types are intended to convey an intent or purpose, it may make much sense to create taxonomies of such abstract relation types. This would entail the identification of specific properties and criteria, whose variance cross-cuts among the various relation types. These could then be used as building blocks for defining relation types, irrespective of the particular underlying model types. Combined with the ideas discussed in the previous paragraph, this could prove to be a powerful mechanism for quickly deriving concrete relation types.

Finally, as described in Section 5.2, composition operators can be declared with specific signatures. In the same section, we discussed how the output of such operators is not guaranteed to conform the the designated output relation type, and that the result of validating against its type can be used to draw conclusions e.g. with respect to con-

sistency. At the same time however, we feel that, at least for some input types, it could be possible to prove that the output will always conform to its designated type. Such a potential opens up the space for automating many model management tasks, and should therefore be further investigated.

Bibliography

[1] AMMA project home page: `http://atlanmod.emn.fr/AMMAROOT/`. Accessed April 2010.

[2] AMW project home page: `http://www.eclipse.org/gmt/amw/`. Accessed April 2010.

[3] AUTOSAR project home page: `http://www.autosar.org/`. Accessed April 2010.

[4] Eclipse Modeling Framework project page: `http://www.eclipse.org/modeling/emf/`. Accessed April 2010.

[5] Eclipse project home page: `http://www.eclipse.org/`. Accessed April 2010.

[6] Epsilon project home page: `http://www.eclipse.org/gmt/epsilon/`. Accessed April 2010.

[7] Graphical Modeling Framework project page: `http://www.eclipse.org/modeling/gmf/`. Accessed April 2010.

[8] L. Aceto, A. Ingólfsdóttir, K. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, August 2007.

[9] A. Albinet, S. Begoc, JL. Boulanger, O. Casse, I. Dal, H. Dubois, F. Lakhal, D. Louar, MA. Peraldi-Frati, Y. Sorel, et al. "The MeMVaTEx methodology: from requirements to models in automotive application design". In *Proceedings of the 4th European Congress Embedded Real-Time Software*, 2008.

[10] R. Baillargeon and R. Flores. "From algorithms to software – a practical approach to model-driven design". In *SAE Transactions Journal of Passenger Cars: Electronic and Electrical Systems*, pages 619–630, 2007.

[11] P. Bernstein. "Applying model management to classical meta data problems.". In *Proceedings of the Conference on Innovative Data Systems Research*, 2003.

[12] P. Bernstein, A. Halevy, and R. Pottinger. "A vision for management of complex models". *ACM SIGMOD Record*, 29(4):55–63, 2000.

[13] P. Bernstein and S. Melnik. "Model management 2.0: manipulating richer mappings". In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2007.

[14] J. Bézivin and F. Jouault. "Using ATL for checking models". In *Proceedings of the International Workshop on Graph and Model Transformation*, pages 69 – 81, 2006.

[15] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. "Modeling in the large and modeling in the small". In *Proceedings of the Model Driven Architecture, European MDA Workshop: Foundations and Applications*, pages 33–46. 2005.

[16] A. Bharadwaj, J. Choobineh, A. Lo, and B. Shetty. "Model management systems: a survey". *Annals of Operations Research*, 38(1):17–67, 1992.

[17] A. Boronat and J. Meseguer. "An algebraic semantics for MOF". In *Proceedings of Fundamental Approaches to Software Engineering, 11th International Conference*, pages 377–391, 2008.

[18] M. Broy. "Challenges in automotive software engineering". In *Proceedings of the International Conference on Software Engineering*, pages 33–42, 2006.

[19] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. "Verification and validation of declarative model-to-model transformations through invariants". *Journal of Systems and Software*, 83(2):283–302, 2010.

[20] M. Didonet Del Fabro, P. Albert, J. Bézivin, and F. Jouault. "Achieving rule interoperability using chains of model transformations". In *Proceedings of Theory and Practice of Model Transformations, Second International Conference*, pages 249–259, 2009.

[21] M. Didonet Del Fabro and F. Jouault. "Model transformation and weaving in the AMMA platform". In *Proceedings of the Workshop on Generative and Transformational Techniques in Software Engineering*, page 71, 2005.

[22] R. France and B. Rumpe. "Model-driven development of complex software: a research roadmap". In *Proceedings of the International Conference on Software Engineering 2007 Future of Software Engineering*, pages 37–54, 2007.

[23] J. Goguen and R. Burstall. "Institutions: abstract model theory for specification and programming". *Journal of the ACM*, 39(1):95–146, 1992.

[24] A. Goknil, I. Kurtev, K. van den Berg, and J. Veldhuis. "Semantics of trace relations in requirements models for consistency checking and inferencing". *Software and Systems Modeling*, 2010. To appear.

[25] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtevand P. Valduriez. "ATL: a QVT-like transformation language". In *Companion Proceedings of the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 719–720, 2006.

[26] D. Kolovos. "Establishing correspondences between models with the Epsilon Comparison Language". In *Proceedings of the European Conference on Modelling Foundations and Applications*, pages 146–157, 2009.

[27] D. Kolovos, R. Paige, and F. Polack. "Merging models with the Epsilon Merging Language (EML)". In *Proceedings of Model Driven Engineering Languages and Systems, 9th International Conference*, pages 215–229, 2006.

[28] D. Kolovos, R. Paige, and F. Polack. "The Epsilon Object Language (EOL)". In *Proceedings of the European Conference on Modelling Foundations and Applications*, pages 128–142, 2006.

[29] D. Kolovos, R. Paige, and F. Polack. "Detecting and repairing inconsistencies across heterogeneous models". In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 356–364, 2008.

[30] D. Kolovos, R. Paige, and F. Polack. "The Epsilon Transformation Language". In *Proceedings of the International Conference on Model Transformation*, pages 46–60, 2008.

[31] T. Kühne. "What is a model?". In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005.

[32] Object Management Group. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[33] Object Management Group. *Object Constraint Language OMG Available Specification Version 2.0*, 2006.

[34] Object Management Group. *Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification Version 1.0s*, 2008.

[35] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure Version 2.2*, 2009.

[36] A. Pretschner, M. Broy, I. Kruger, and T. Stauner. "Software engineering for automotive systems: a roadmap". In *Proceedings of the International Conference on Software Engineering 2007 Future of Software Engineering*, pages 55–71, 2007.

[37] G. Rushton and R. Baillargeon. "Model-driven product line software development process". In *SAE Transactions Journal of Passenger Cars: Electronic and Electrical Systems*, 2005.

[38] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. "Consistency checking of conceptual models via model merging". In *Proceedings of the International Requirements Engineering Conference*, pages 221–230, 2007.

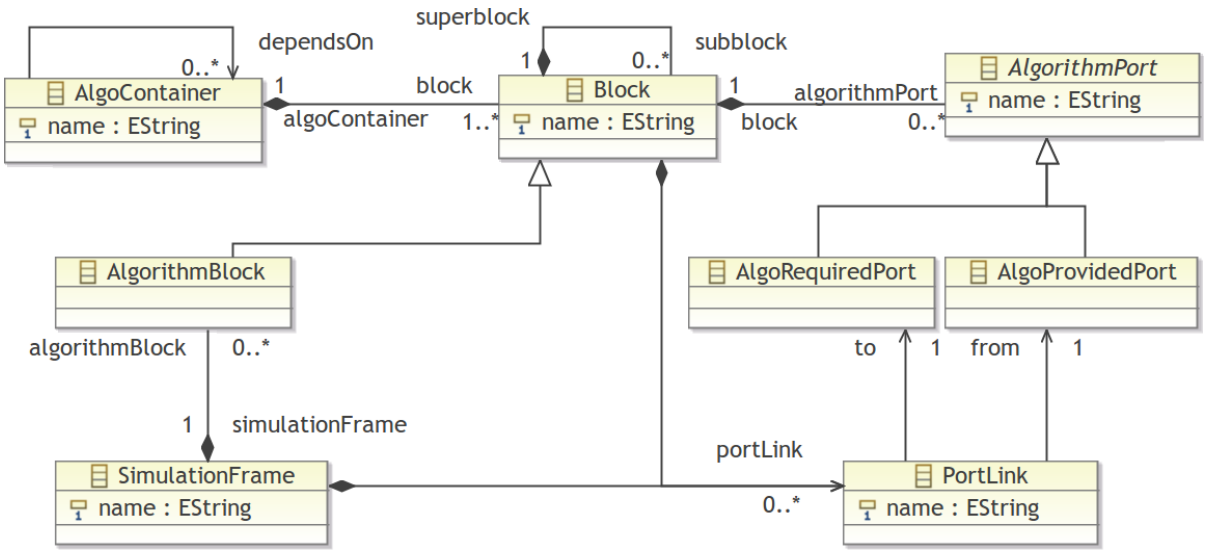
[39] R. Salay, M. Chechik, S. Easterbrook, Z. Diskin, P. McCormick, S. Nejati, M. Sabetzadeh, and P. Viriyakattiyaporn. "An Eclipse-based tool framework for software model management". In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications Workshop on Eclipse technology eXchange*, pages 55–59, 2007.

[40] R. Salay, J. Mylopoulos, and S. Easterbrook. "Using macromodels to manage collections of related models". In *Proceedings of the International Conference on Advanced Information Systems Engineering*, volume 5565, 2009.

[41] J. Steel and J. Jézéquel. "On model typing". *Software and System Modeling*, 6(4):401–413, 2007.

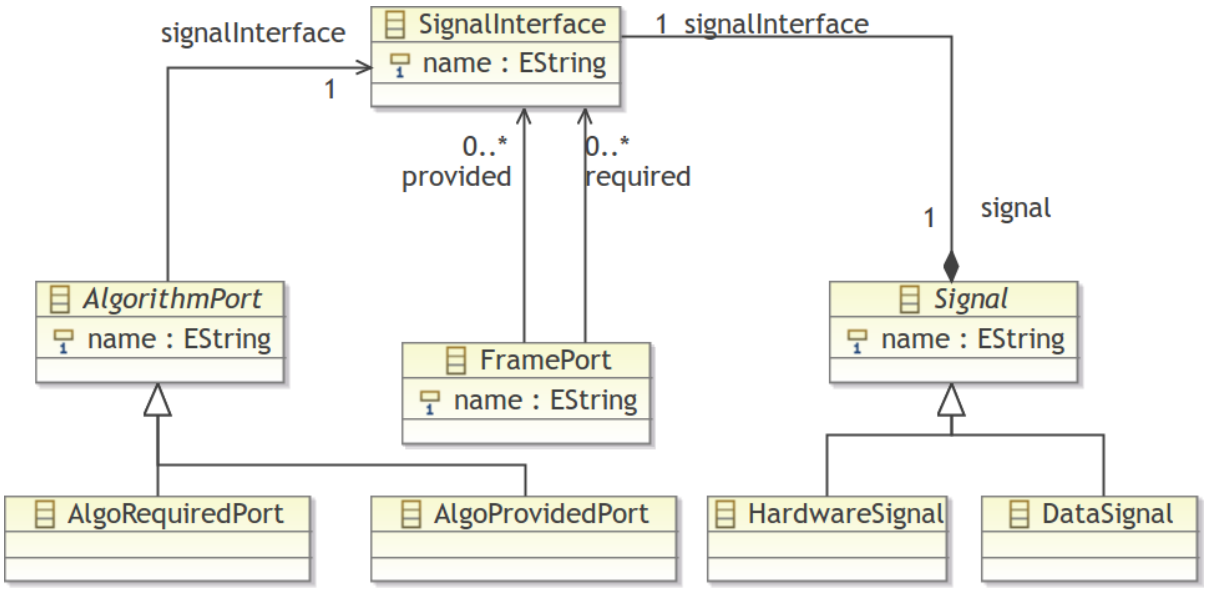
Appendix A

Additional Metamodels



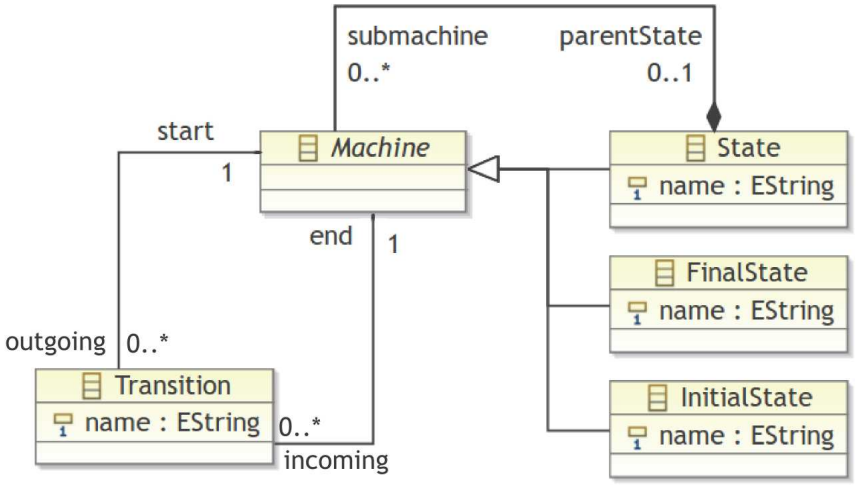
Algorithm Blocks are the basic elements for creating logical architectures of components. They can be decomposed internally in Block elements to describe distinct parts of behavior. They can also be organized in Simulation Frames, that contain scheduling information. Additionally, they can have a number of Ports for accessing Interfaces to Signals (see Figure A.2). Port links can be used to create connections between blocks' ports, following the required/provided pattern for interfaces.

Figure A.1: Metamodel for Algorithm Block Diagrams, discussed in Section 2.3.1.



For each Signal, defined as a hardware or data signal, an Interface element must be declared. Signal Interfaces are utilized by Algorithm Blocks (see Figure A.1) via required and provided port elements. Interfaces can also be bundled in Frame ports.

Figure A.2: Metamodel for Signals Diagrams, discussed in Section 2.3.2.

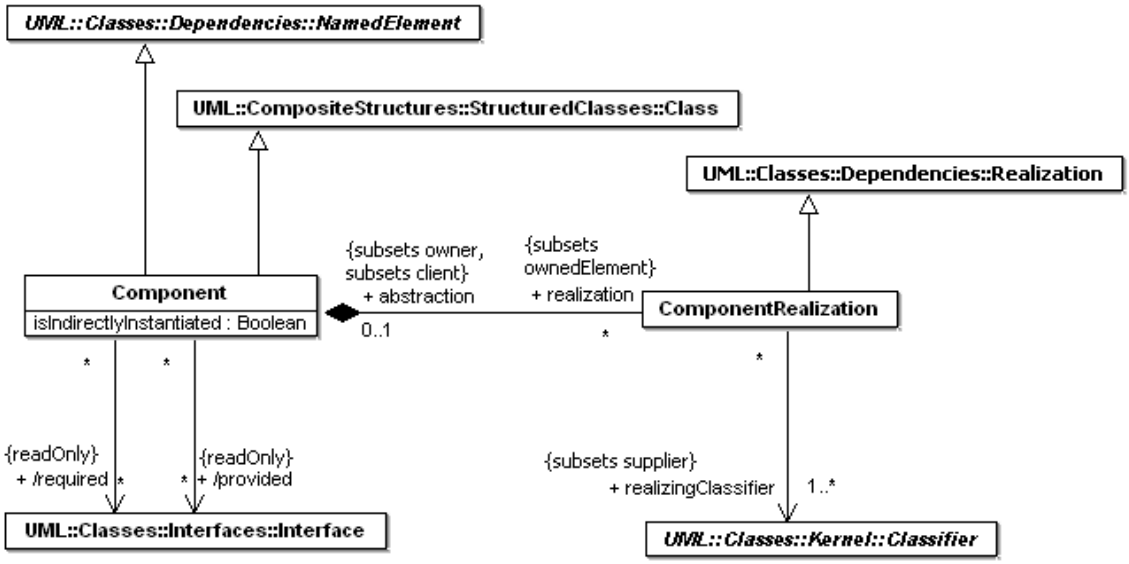


context Machine **inv**:

self−> count(outgoing) <= 1

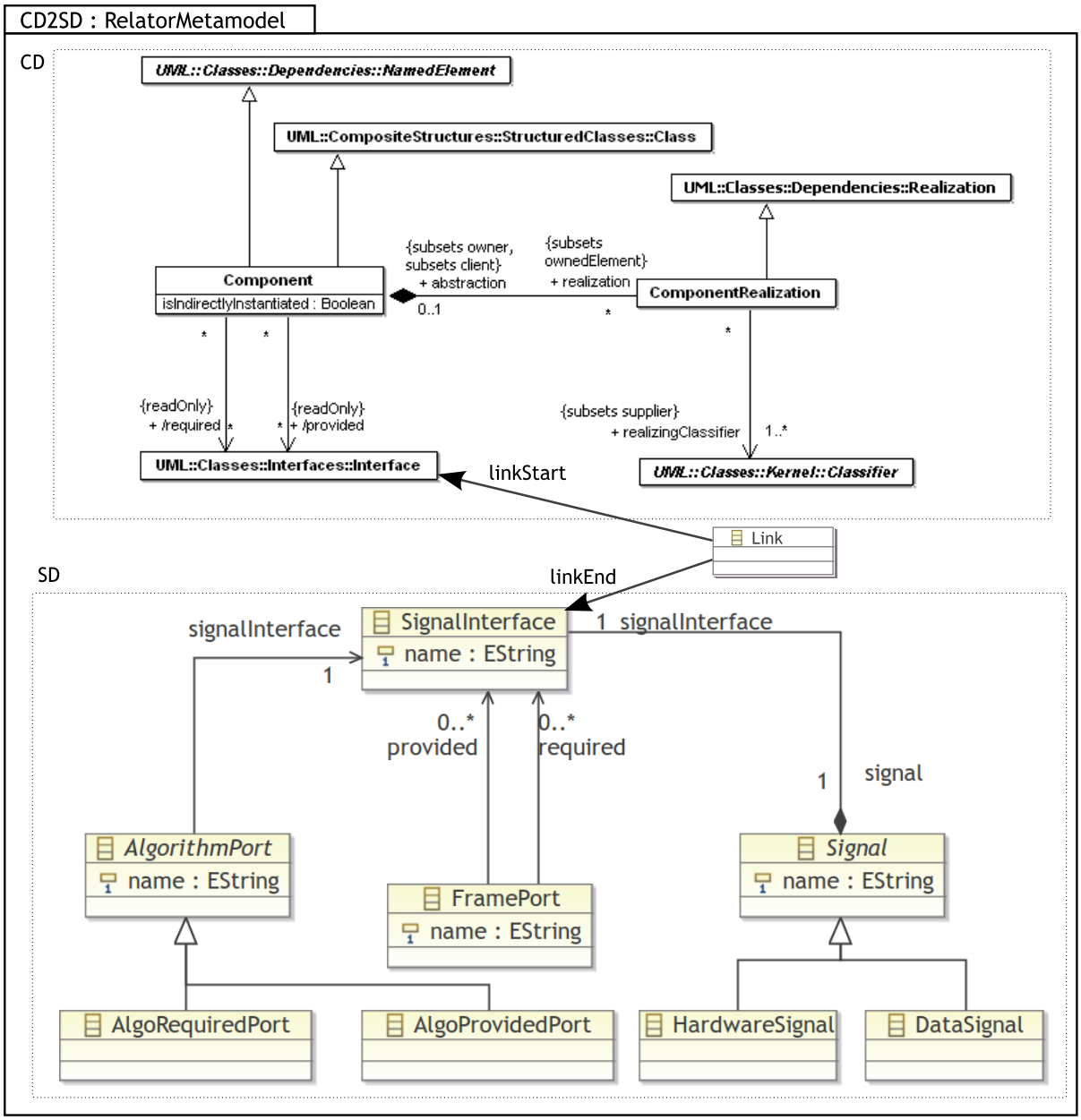
In general, a state machine consists of states and transitions. States can be simple or composite, and there exist initial and final states. For the domain specific state machines of the automotive domain, only one outgoing transition is allowed for each state.

Figure A.3: Metamodel for State Machines, discussed in Section 2.3.3.



On a basic level, UML Component Diagrams contain component elements which possibly expose a number of provided and required interfaces.

Figure A.4: Basic metamodel for UML Component Diagrams, as discussed in Section 2.3.4. Source: UML Specification[35], Figure 8.2.



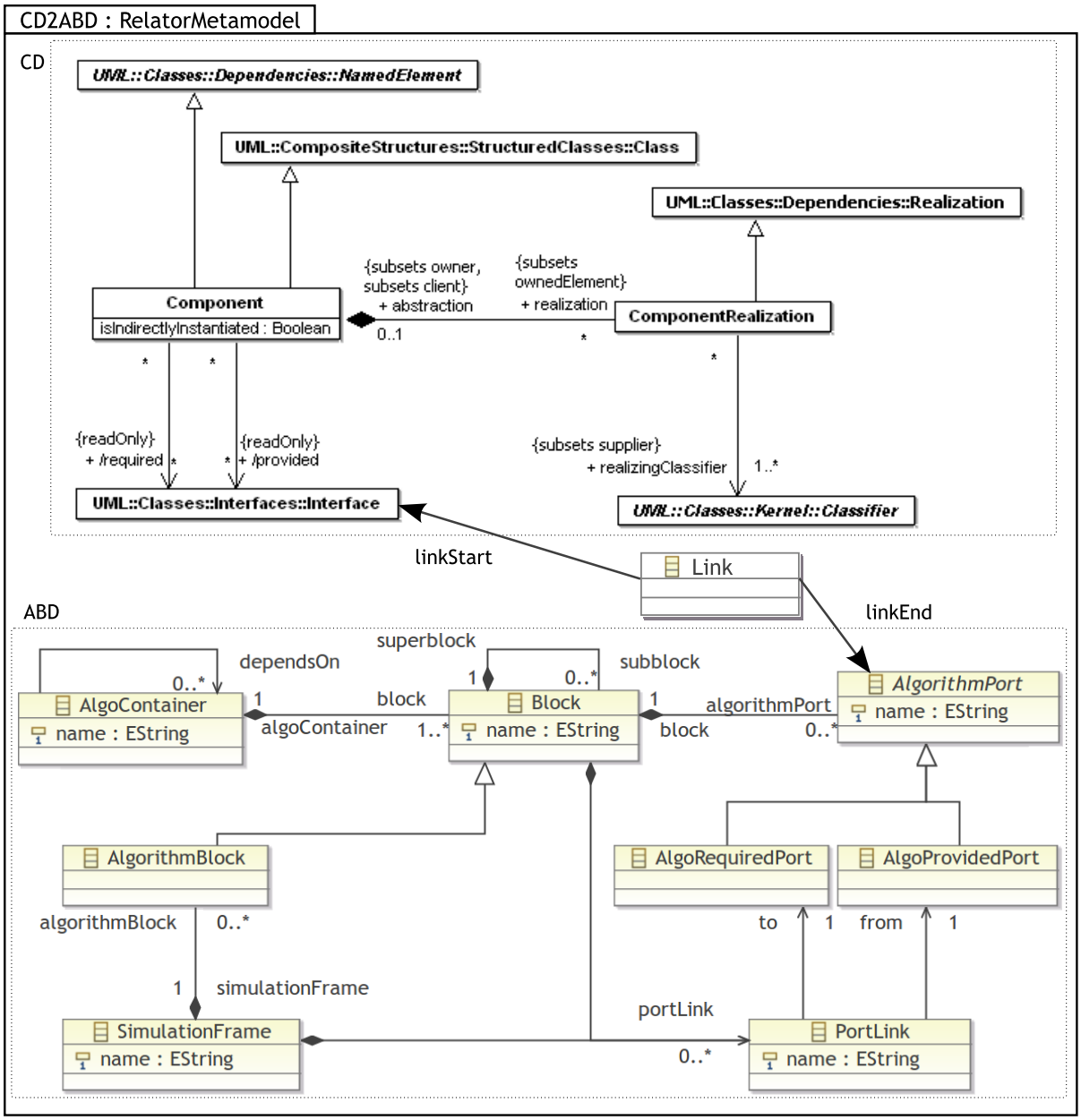
context l : Link **inv**:

CD!Interface.allInstances()−>forAll(i | SD!SignalInterface.allInstances()−>exists(

s | i.name=s.name and l.start=i and s.end=s))

In this interface consistency concrete relation type, for every Interface element in the Component Diagram there must exist a SignalInterface element in the Signals Diagram, and they must be connected via a Link element.

Figure A.5: `IConsistecyCD-SD` relator metamodel, discussed in Section 5.2.1.



context l : Link **inv**:

CD!Interface.allInstances()−>forAll(i | ABD!AlgorithmPort.allInstances()−>exists(

a | i.name=a.name and l.start=i and l.end=i))

In this interface consistency concrete relation type, for every Interface element in the Component Diagram there must exist a Port element in the Algorithm Block Diagram, and they must be connected via a Link element.

Figure A.6: `IConsistencyCD-ABD` relator metamodel, discussed in Section 5.2.1.