Domain Modeling and Domain Engineering: Key Tasks in Requirements Engineering

Manfred Broy

Abstract

Requirements engineering is an essential part of software and systems development. Besides the elicitation, analysis, and specification of the intrinsic system requirements as a basis for these activities, it also involves the elicitation, analysis, and specification of the information about the application domain (also called problem domain or domain for short: includes terminology, concepts, and rules). The result of this activity is an elaborated domain model, which is a model of the relevant parts of the application domain.

Roughly speaking, a domain model for a system or software development task comprises the following parts:

- The domain ontology rules, laws, terminology, and notions describing the relevant terms giving an ontology/taxonomy of the domain and specific rules and principles
  - Concepts, data types, and functions
  - Rules and laws
- The *context model,* which describes the general properties of the system's environment. This includes the operational context such as software systems, physical systems, and actors, encompassing users in the environment, properties of the physical environment in case of cyber-physical systems, as well as the wider business and technological context.

  These aspects can be captured by adequate data models.

The domain model collects all the information about the problem domain that must be known and understood to allow capturing requirements for the system, specifying them, implementing and verifying the system. The detailed system requirements, however, are not part of the domain model, but they are based upon it.

M. Broy (✉)
Institut für Informatik, Technische Universität München, 80290 Munich, Germany
e-mail: broy@in.tum.de

J. Münch and K. Schmid (eds.), *Perspectives on the Future of Software Engineering*,
DOI 10.1007/978-3-642-37395-4_2, © Springer-Verlag Berlin Heidelberg 2013

Ultimately, the domain model is a collection of knowledge about the application domain at an adequate level of abstraction—including the use of modeling techniques where useful.

1 Introduction

For developing software and software-intensive systems, a variety of different categories of knowledge is required, including know-how about system and software development processes and methodology, software and hardware technology, and, last but not least, knowledge about the application domain (also called the problem domain). The knowledge about the application domain is captured in a process of domain engineering that develops domain models. The importance of domain modeling has been recognized in the early 1990s (see [1]).

A *domain model* is a conceptual model capturing the topics related to a specific problem domain. Domain theory and domain modeling comprise several aspects. One is the theory of the domain itself. For instance, in domains there exist a number of notions, insights, and rules, which are important when developing domain-specific software. An example would be how to calculate interest in a banking application or how to calculate certain routes in navigation, or speed and acceleration values, in an automotive application Often these tasks require deep insights into the application fields and some understanding of their domain theories and experience.

In domain modeling, we develop different forms of ontologies and taxonomies to capture and document domain knowledge. We may use very logical ontologies related rather to domain-specific terminology and observations (often documented by data dictionaries, glossaries, or meta-models) or very technical ontologies, where we look at technical and physical terms and phenomena.

To capture domain models we have to use quite different kinds of modeling techniques to represent the different parts of domain models. For instance, in cases of domains where we can refer to well-developed theories as in physics, chemistry, or in certain fields of engineering, we can more or less take over these models and domain theories, as they have been worked out in the domain over decades, and import them into the knowledge used in software engineering. At any rate, in many cases it is advisable to import parts of the theory and terminology into the specific models and techniques of software engineering.

If we are dealing with an application domain where no systematic and comprehensive domain models exist yet, we need to work out an adequate domain theory during software and system development. We may only have to represent it in terms of software engineering specific models.

2 Structuring Domain Information

A domain model gathers all the information about a domain as needed to understand and formulate the requirements on a particular application system to be developed in software and systems engineering (see [2, 3]). Domain knowledge and domain properties, in contrast to requirements, cannot be chosen freely (see [4]), but have to reflect facts about the application domain and the operational context of the System or Software under Construction (SuC). This is in contrast to requirements that can be chosen freely according to the stakeholder needs. Domain modeling is part of problem solving and software engineering to develop conceptual models of domains of interest (often referred to as "problem domain") which describes the various notions, entities, their attributes and relationships, plus the constraints that govern the integrity of the model elements comprising that problem domain.

2.1 Domain Models

We consider the following categories of information as part of the domain model:

- *Operational system context*: this comprises all the information about the system's environment such as surrounding systems, properties of users, or sensor input.
- *Application domain model*: general domain terminologies, basic notions, rules, and experiences of the application domain
- *Wider system context*: aspects of business, market, processes, technology, organisation, law, sociology, psychology

For these three categories, different forms and degrees of formalizations are advisable. In contrast to system requirements as captured in requirements engineering, the information captured in the domain model is generally not subject to design decisions, but has to be captured as given and valid properties of the problem world.

2.1.1 Domain Modeling and Requirements Engineering

A *domain model* identifies fundamental business- and application-specific entity types and relationships between them, including business processes.

In contrast to the system specification, where the properties of a system are structured and captured in terms of adequately chosen system models, domain models are often quite heterogeneous and diffuse and depend on the particularities of the problem domain. Thus, different modeling techniques should be applied for domain modeling.

Relevant domain information that domain models may include is information about

- Terminology and key notions and concepts
- Operational context: systems and users in the environment and their behavior in terms of interaction with the SuC, including business processes supported by the SuC.
- Application domain concepts and rules
- Business rules

Typically, different areas of information require different techniques for their representation and different kinds of modeling techniques.

Domain knowledge can therefore be roughly structured into the following categories:

- *Background Knowledge* is general knowledge about the application domain, its rules and principles, its terminology as well as its basic notions and concepts. This knowledge is useful for understanding and provides a rationale for decisions and specifications.
- *Operational Context Knowledge* is knowledge about the system's operational environment. It includes the reactions of systems that are part of the environment as well as of users. Formally this knowledge leads to assumptions about the environment, behavior of users and systems in the operational context, and thus about the expected input to the system (see [5]).
- *Direct domain knowledge* in the system refers to knowledge that directly influences the reaction of the system under development. Examples would be how quickly a system has to react to a crash sensor if an airbag is to be activated or how to calculate interest in banking applications (see [6] for an analysis of domain-specific know-how embedded in programs).

Of course, the borderlines between these categories of knowledge are not sharp. Often background knowledge is ultimately used as direct domain knowledge.

The parts and properties of the domain model directly influence the system specification.

- Data model: Typically, the data model for a system reflects terms and notions from the application domain (example: the term and notion of speed may become a data type).
- Operational context model assumptions: In system specifications, we typically find assumptions about the system's operational context (example: the speed cannot increase within one second by more than 10 km/h).

The operational context model describes systems and users in the environment of the system under consideration. For them the same modeling concepts (see Appendix) can be used as for the system under consideration. For the domain model, typical aspects of requirements engineering apply:

- There are different levels of abstraction that we can choose for the individual parts of the domain models.
- We may apply notions of refinement to domain models.
- There might be some uncertainty and disagreement about specific properties of a domain; as a result some information in the domain model may be invalid.

Therefore, domain models have to be validated just like requirements and system specifications.

2.1.2 Domain Knowledge in Context-Aware Systems

The category of context knowledge and direct domain knowledge may overlap in so-called context-aware systems. In this case, context and domain knowledge may be explicitly part of the data states of systems. This means that the system stores information about its operational context, its rules of behavior, as well as the actual

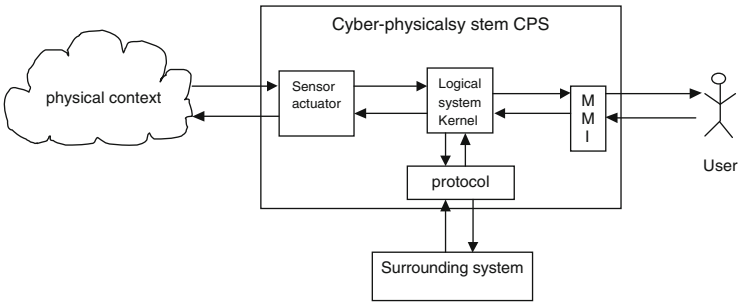

Fig. 1 Two system boundaries: the logical system kernel and the system boundary including sensors

state of its operational context. Then systems may react to specific input depending on the context information stored in the system.

In such cases, excerpts of the domain model are used directly as part of the system state model.

2.2 Scoping: Choosing Boundaries for the System Under Development—Changing System Scope, Interface and Context

Another issue in domain modeling is related to system scoping. In many situations we can choose different scopes for systems under development. Actually, we may consider, in particular, embedded systems having an onion ring structure with a set of onion rings defining operational contexts. If we take the largest scope of the system, then we see the system with its physical surface as a user would see it (Fig. 1). Then we can choose various narrower and more technical scopes and views inside the system, such as the IT systems including sensors and actuators or its IT structure without the sensors and actuators, or just look at the software system itself or only at the CPUs. In every case, we get different scopes and hence different interfaces and relations to domain theories.

Sometimes different system boundaries are considered during system development. We then get systems with different operational context and different boundaries. This means that parts of the context information may become part of the system and vice versa.

We get a formula that describes the interface of the "outer" system and its behavior (for the definition of the operator $\otimes$, see the system model in the Appendix):

$$\mathrm{CPS} = \mathrm{S/A} \otimes \mathrm{LSK} \otimes \mathrm{PRC} \otimes \mathrm{MMI}$$

The inner system is described by the logical system kernel LSK, the outer one by CPS.

The architecture of the system including the behaviors of the actors in the context PHYC, USER, and SURSYS is described by (for the definition of the operator [×] see the system model in the Appendix):

$$\text{SYS} = \text{PHYC } [\times] \text{ USER } [\times] \text{ SURSYS } [\times] \text{ CPS}$$

In this case, we consider three different system scopes and two different contexts:

1. The empty context for SYS
2. The physical context, the user, surrounding system for the CPS
3. The (sensor/actuator ⊗ physical context), the (MMI ⊗ User) and the (protocol ⊗ surrounding-system) for the logical system kernel

The relationship between the interface behavior of the logical system LSK and the overall system CPS can be understood as vertical refinement (also called a layered system). The behavior of the context PHYC, USER, and SURSYS can be seen as assumptions (see [5], where assumptions are called promises).

2.3 Representing Domain Models

Domain models in terms of operational system contexts should not be modeled much differently than other system models. We can use classical "algebraic" data type specifications to model ontologies and domain-specific data models. Context systems including user behavior can be captured by classical system models but have to be enriched by human factor issues and user models.

Different levels of abstraction can be captured by functions relating levels of abstraction.

General domain knowledge requires different kinds of models, such as

- Ontologies introducing terminology, notions, and their relationships
- Algebraic specifications corresponding to interpreted ontologies

These specifications introduce theories as a basis of software specification and verification.

2.3.1 Domain Models and Their Formalization

Much attention in research has been devoted to so-called formal methods. Their goal is to formalize system properties as part of the specification and support development through formal refinement and verification steps. This form of formalization is justified by the fact that software can be seen as a formal artifact and that software development includes and explicitly or implicitly enforces a step towards formalization.

This is not true for the documentation of the problem domain. Large parts of the problem domain have not to be formalized, since they do not directly relate to software as a formal artifact. Only operational context information has to be formalized since it interacts directly with the software as a formal artifact.

2.3.2 The Integration Problem

Often, there is no homogeneous problem domain. Rather, there are several problem sub-domains with domain models perhaps represented in fairly different description formalisms. The different formalisms are difficult to integrate and to combine into a consistent, coherent problem domain.

2.3.3 The Translation Problem

Sometimes similar or related information is formulated in different languages, using different terms and different levels of abstractions. In these cases the different linguistic frameworks have to be related via translations.

2.3.4 Relating the Problem Domain and the Technical Level: User Input, Sensor Information, and Domain Knowledge

Typically, it is useful to speak about the problem domain independent of the technical implementation—at least in capturing system-level requirements. For instance, a requirement for a car may read as follows:

> In case of a crash, the airbag has to be fully activated within 200 msec.

At the technical level, this reads:

> If the sensor XYZ issues a signal CI, then the signal AA has to be issued within 150 msec.

These two requirements are formulated at completely different levels and have to related to each other.

A famous example is from the development of software for an airplane, where we may ask the question what it means that an "airplane is on the ground and moving fast". Being on the ground is certainly an important logical property in the problem domain of an airplane, which refers to its context, more precisely to the position of the airplane in its physical environment in which the airplane is located. If we are interested in finding out how to detect via sensors that an airplane is on the ground, we need a more technical description for this property. To grasp the property "airplane on ground and moving fast", there are two completely different views of an airplane. One view is a view reflected in a domain model where the plane is seen as part of larger systems. This way we can talk about the position of airplanes and particular aspects of the position, where one would be "the airplane is on the ground and moving fast". Another issue is how to detect and observe this property technically via sensors within the system. One possibility is to consider the torque on the wheels, meaning that if the wheels turn with a certain torque, we may conclude that the plane is on the ground moving at a certain speed.

Sensors capture information about the operational context and its actual state. This information is used as input for the system. For instance, if a sensor measures the speed of a system or its geographical position, then this is operational context information. It is part of the domain model to relate certain sensor information to domain aspects. More precisely speaking, sensors deliver numerical values. To know that these numerical values represent speed with sufficient accuracy for a certain time slot is additional information with reference to the domain model.

In a more general view we can say that with the help of the domain model we interpret the sensor information in terms of the domain model and relate technical requirements to logical requirements (example: “If speed is greater than 20 km/h, the airbags are activated in case of a crash”, which is translated into a specification referring to certain sensors, their actual values, and attributes of the state; for details see [7]).

2.3.5 The Validation Problem

The information captured in the problem domain has to be validated. Invalid information in the problem domain leads to invalid assumptions in the requirements and may ultimately result in unsafe, insecure, or unreliable systems.

2.3.6 From Problem Domains to Assumptions in Specifications

The information captured in problem domains serves as assumptions for the requirements. This leads to a specific form of writing requirements specifications (see [5], where assumptions are called promises).

3 Modeling System Context as Part of the Domain Model

For the system model introduced in the appendix, the notion of scope and context is essential. A system has to be clearly separated from its context. The notion of context comprises everything that is not inside the system’s boundary. However, we are of course only interested in context aspects in connection with the system requirements that are relevant for the system, its properties, its behavior, and its requirements. In requirements engineering, we capture “facts” about the context—as far as they are relevant for the system under development. They serve as assumptions.

It is helpful to classify the elements of the context further. A straightforward characterization yields the following parts:

- Operational context (Fig. 2): systems, users, and physical environment with which the system under development interacts, possibly as part of business processes in the context
- Usage context: scale of usage, number of users, time (“duration”)
- Business context: marketing the system in use
- Development process context: development processes
- Execution context: hardware platform

These context elements have to be captured in sufficient detail during requirements engineering to the extent that they are needed for reflecting and documenting requirements. Strictly speaking, the properties of the context are not requirements but assumptions on which the requirements may rely (Fig. 2).

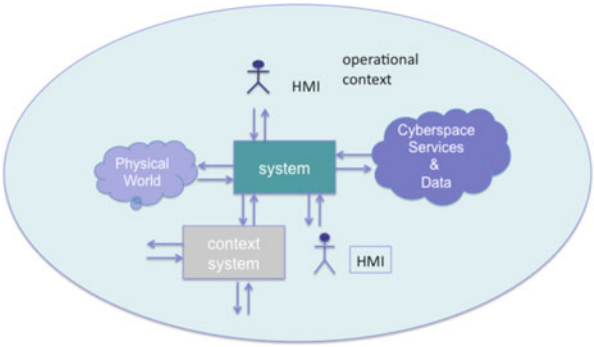

Fig. 2 System and its operational context

The operational context is also described by a syntactic interface and its behavior. There is a rich variety of context aspects:

Business Context: In this business view, we deal with issues of the system related to business and marketing. This may be the number of system instances sold, contract issues, price, and so on. This category may include questions about the cost and value of a system or parts of it.

Development Process Context: In this view, we deal with requirements addressing properties of the development process. Typical examples are the choice of the life cycle model or certain standards of certification.

Execution Context: Platform, runtime environment, and execution hardware—if not part of the system under development—are part of the domain model.

For these different forms of contexts, different modeling techniques are used.

4 Summary and Outlook

Domain modeling is a highly relevant field in software and systems development. Domain modeling must integrate domains, specific techniques for modeling, and modeling techniques and concepts from software and systems engineering. To a large extent, domain knowledge has to be represented by models from software and systems engineering in such a way that domain experts can still validate them and work with them.

At the same time, thorough understanding of a comprehensive problem domain is difficult and requires considerable effort. An example of such a domain approach is [8]. Today, software and systems development is done mainly by domain experts. Software and systems engineers are increasingly becoming domain experts at the same time. This is a very interesting development that has to be taken into account for the role models in software and systems engineering.

The importance of domain modeling will grow even more for several reasons:

- Software systems will be related and integrated into their problem domains tighter and deeper—this cannot be achieved without comprehensive knowledge.
- The development of such software systems calls for deep integration of concepts and modeling techniques from the problem domain and from software and system engineering (see [9]).
- Domain know-how will become an ever greater asset that will be documented and reused and will be essential for the quality of software systems (see [10]).

Last but not least, the process of capturing domain knowledge will increase the knowledge in the problem domain field. The term "computational thinking" (see [11]) is an indication of such approaches where capturing problem domains using computer science methods is claimed to become a new form of scientific method.

A.1 Appendix: The System Model

We use a specific notion of discrete systems in this paper following [12] with the following characteristics and principles.

- A discrete *system* has a well-defined boundary that determines its *interface*.
- Everything outside the system boundary is called the system's *environment*. Those parts of the environment that are relevant for the system are called the system's *context*. Actors in the context that interact with the system, such as users, neighbored systems, or sensor and actors connected to the physical context are called the operational context.
- A system's interface indicates the steps through which the system interacts with its operational context. The syntactic interface defines the set of actions that can be performed in interaction with a system across its boundary. In our case, syntactic interfaces are defined by the set of input and output channels together with their types. The input channels define the input actions for a system, while the output channels define the output actions for a system.
- We distinguish between the *syntactic interface*, also called *static interface*, which describes the set of input and output actions that can take place across the system boundary, and the *interface behavior* (also called *dynamic interface*), which describes the system's *functionality*; the interface behavior is captured by the causal relationship between streams of actions captured in the input and output *histories*. This way we define a logical behavior as well as a probabilistic behavior for systems.
- The logical interface behavior of a system is described by means of logical expressions, called *interface assertions* or by *state machines,* or it can be further decomposed into *architectures*.
- A system has an *internal structure* and behavior ("glass box view"). This structure is described by its state space with state transitions and/or by its decomposition into sub-systems forming its architecture in case the system is decomposed into a number of subsystems that interact and also provide the

interaction with the system's context. The state machine and the architecture associated with a system is called its state view and its structural or architectural view, respectively.

- In a complementary way, the behavior of a system can be described by sets of *traces*, which are sets of scenarios of the input and output behavior of a system. We distinguish between finite and infinite scenarios.
- Moreover, systems operate in time. In our case, we use discrete time, which seems particularly adequate for discrete systems. Sub-systems operate concurrently within the architecture.

This gives a highly abstract and at the same time quite comprehensive model of a system. This model is formalized in the following by one specific modeling theory.

A.2 Data Models: Data Types

Data models define a set of data types and some basic functions for them. A *(data) type* T is a name for a data set for which a family of operations is usually available. Let TYPE be the set of all data types.

A.3 Interface Behavior

Systems have *syntactic interfaces* that are described by their sets of input and output channels attributed by the type of messages that are communicated over them. Channels are used to connect systems to allow transmitting messages between them. Formally, a channel is an identifier for a uni-directional communication link. A set of typed channels is a set of channels with types given for each of its channels.

Definition. Syntactic interface

Let I be the set of typed input channels and O be the set of typed output channels. The pair (I, O) characterizes the syntactic interface of a system. The *syntactic interface* is denoted by (I►O). □

Figure 3 shows the syntactic interface of a system F in a graphical representation as a data flow node with its syntactic interface consisting of the input channels x_1, ... of types T_1, ... and the output channels y_1, ... of types T'_1, ...

Definition. Timed Streams

Given a message set M of data elements of type T, we represent a *timed stream* s of type T by a mapping

$$s : IN\backslash\{0\} \rightarrow M^*$$

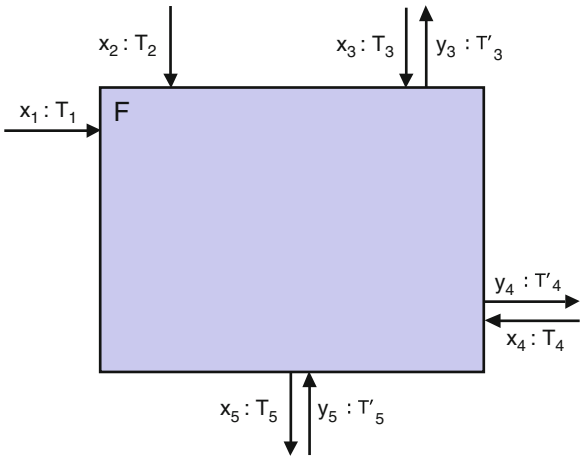

Fig. 3 Graphical representation of a system F as a data flow node

In a timed stream s, a sequence s(t) of messages is given for each time interval $t \in IN\backslash\{0\}$. In each time interval, an arbitrary, but finite number of messages may be communicated. By $(M^*)\infty$ we denote the set of timed streams. □

A (timed) channel history for a set of typed channels C assigns to each channel $c \in C$ a timed stream of messages communicated over that channel.

Definition. Channel history

Let C be a set of typed channels; a (total) *channel history* x is a mapping (let IM be the universe of all messages)

$$x : C \to \left(IN\backslash\{0\} \to IM^*\right)$$

such that x(c) is a timed stream of messages of the type of channel $c \in C$. $\vec{\mathbf{C}}$ denotes the set of all total channel histories for the channel set C. □

The behavior of a system with a syntactic interface (I►O) is defined by a mapping that maps the input histories in $\vec{I}$ onto output histories in $\vec{O}$. This way we get a functional model of a system interface behavior.

Definition. I/O-Behavior (see [13])

A causal mapping $F : \vec{I} \to \wp(\vec{O})$ is called an *I/O-behavior*. By IF[I►O] we denote the set of all (total and partial) I/O-behaviors with a syntactic interface (I►O) and by IF the set of all I/O-behaviors. □

Interface behaviors model system functionality. For systems we assume that their interface behavior is total. F behaviors may be deterministic (in this case, the set F(x) of output histories has at most one element for each input history x) or nondeterministic.

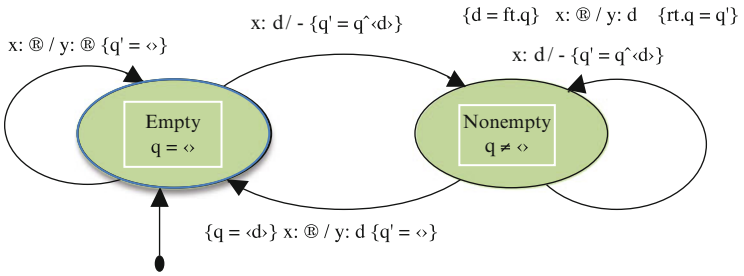

Fig. 4 A simple state machine—described by a state transition graph

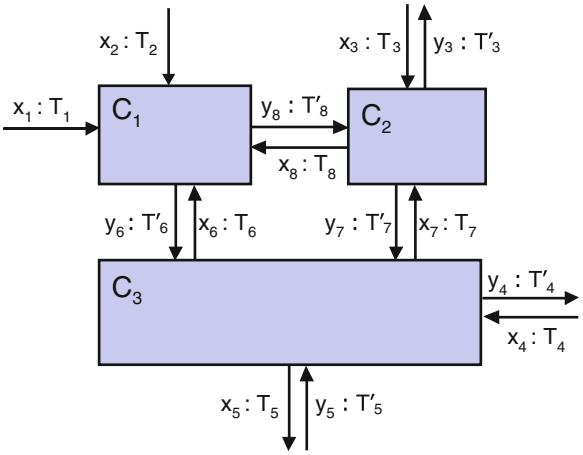

Fig. 5 A simple architecture—described by a data flow graph

A.4 State Machines by State Transition Functions

State machines with input and output describe system implementations in terms of states and state transitions. A state machine is defined by a state space and a state transition.

Definition. State Machine with Syntactic Interface (I►O)

Given a state space Σ, a state machine (Δ, Λ) with input and output according to the syntactic interface (I►O) consists of a set $\Lambda \subseteq \Sigma$ of initial states as well as of a nondeterministic state transition function □

$$\Delta : (\Sigma \times (I \to M^*)) \to \wp(\Sigma \times (O \to M^*))$$

For each state $\sigma \in \Sigma$ and each valuation a: $I \to M^*$ of the input channels in I by sequences of input messages, every pair $(\sigma', b) \in \Delta(\sigma, a)$ defines a successor state σ' and a valuation b: $O \to M^*$ of the output channels consisting of the sequences produced by the state transition (Fig. 5).

A.5 Architecture

In the following, we assume that each system used in an architecture as a component has a unique identifier k. Let K be the set of identifiers for the components of an architecture.

Definition. Set of Composable Interfaces

A set of component names K with a finite set of interfaces (Ik►Ok) for each identifier $k \in K$ is called *composable* if the following propositions hold:

- The sets of input channels Ik, $k \in K$, are pairwise disjoint,
- The sets of output channels Ok, $k \in K$, are pairwise disjoint,
- The channels in $\{c \in Ik: k \in K\} \cap \{c \in Ok: k \in K\}$ have consistent channel types in $\{c \in Ik: k \in K\}$ and $\{c \in Ok: k \in K\}$. □

If channel names and types are not consistent for a set of systems to be used as components, we can simply rename the channels to make them consistent.

Definition. Syntactic Architecture

A syntactic architecture $A = (K, \xi)$ with the interface $(I_A \blacktriangleright O_A)$ is given by a set K of component names with composable syntactic interfaces $\xi(k) = (Ik \blacktriangleright Ok)$ for $k \in K$.

$I_A = \{c \in Ik: k \in K\} \setminus \{c \in Ok: k \in K\}$ denotes the set of *input* channels of the architecture,
$D_A = \{c \in Ok: k \in K\}$ denotes the set of *generated* channels of the architecture,
$O_A = D_A \setminus \{c \in Ik: k \in K\}$ denotes the set of *output* channels of the architecture,
$D_A \setminus O_A$ denotes the set of *internal* channels of the architecture
$C_A = \{c \in Ik: k \in K\} \cup \{c \in Ok: k \in K\}$ denotes the set of all channels

By $(I_A \blacktriangleright D_A)$ we denote the *syntactic internal interface* and by $(I_A \blacktriangleright O_A)$ we denote the *syntactic external inter*face of the architecture. □

A syntactic architecture forms a directed graph with its components as its nodes and its channels as directed arcs. The input channels in I_A are ingoing arcs and the output channels in O_A are outgoing arcs for that graph.

Definition. Interpreted Architecture

An interpreted architecture (K, ψ) for a syntactic architecture (K, ξ) associates an interface behavior $\psi(k) \in IF[Ik \blacktriangleright Ok]$, where $\xi(k) = (Ik \blacktriangleright Ok)$, with every component $k \in K$. □

An architecture can be specified by a syntactic architecture given by its set of sub-systems and their communication channels and an interface specification for each of its components.

For an interpreted architecture A with syntactic internal interface $(I_A \blacktriangleright D_A)$, we define the glass box interface behavior $[\times]\ A \in IF[I_A \blacktriangleright D_A]$ by the equation (let $\psi(k) = Fk$):

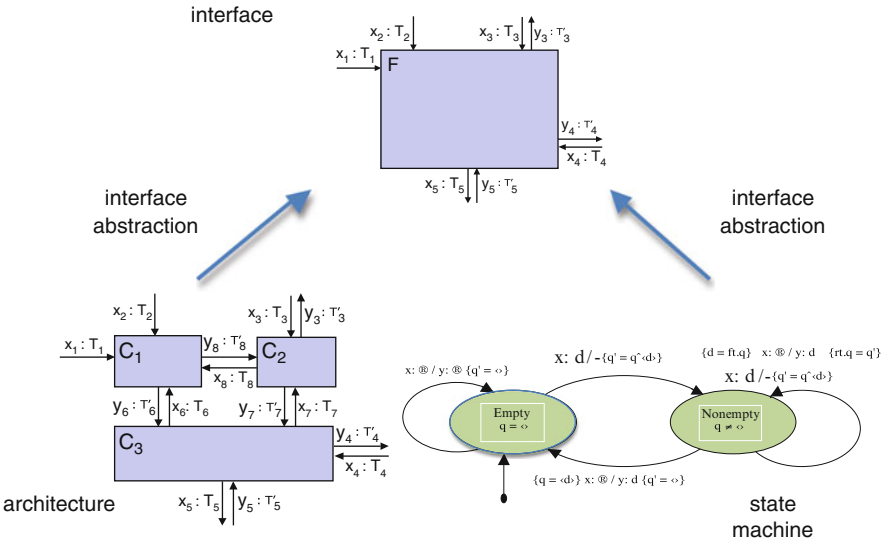

Fig. 6 Abstraction functions between modeling concepts

$$([\times]\ A)(x) = \Big\{ y \in \vec{\mathbf{D}}_A : \exists\ z \in \vec{\mathbf{C}}_A : x = z|I_A \wedge y = z|D_A \wedge \forall\ k\ \in K :\ z|O_k \in F_k(z|I_k) \Big\}$$

$[\times]$ A describes the behavior of the architecture A. For $[\times]\ \{F_1, F_2\}$ we also write $F_1\ [\times]\ F_2$.

In a black box view $\otimes$ A $\in$ IF$[I_A \blacktriangleright O_A]$ onto the architecture we hide internal channels

$$(\otimes A)(x) = \Big\{ y \in \vec{\mathbf{O}}_A :\ \exists\ z\ \in\ \vec{\mathbf{C}}_A :\ x = z|I_A \wedge y = z|O_A \wedge \forall\ k \in\ K :\ z|O_k \in F_k(z|I_k) \Big\}$$

$\otimes$ A describes the interface behavior of the architecture A. For $\otimes\ \{F_1, F_2\}$ we also write $F_1 \otimes F_2$ (Fig. 6).

A.6 Relating the Modeling Concepts

The three basic modeling concepts can be related as shown in Fig. 4. Through interface abstractions we can relate state machines and architectures to interfaces.

References

1. Batory, D., McAllester, D., Coglianese, L., Tracz, W.: Domain modeling in engineering of computer-based systems. In: 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems, Tucson (1995)
2. Gunter, C.A., Gunter, E.L., Jackson, M., Zave, P.: A reference model for requirements and specifications. IEEE Softw. **17**(3), 37–43 (2000)
3. Jackson, M.: Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley, Boston (2001)
4. Kofler, Th., Ratiu, D.: Towards a reusable unified basis for representing business domain knowledge and development artifacts in systems engineering. In: DE@ER2010, Workshop on Domain Engineering (2010)
5. Broy, M.: Towards a theory of architectural contracts:—schemes and patterns of assumption/promise based system specification. In: Broy, M., Leuxner, Ch., Hoare, T. (eds.) Software and Systems Safety—Specification and Verification. NATO Science for Peace and Security Series—D: Information and Communication Security, vol. 30, pp. 33–87. IOS Press, Fairfax
6. Ratiu, D.: Intentional meaning of programs. Dissertation, Technische Universität München, Fakultät für Informatik (2009)
7. Broy, M.: The logic of requirements – formalizing tracing, In: Schnieder, E., Tarnai, G. (eds.) Forms/Format 2012, Technische Universität Braunschweig, Beyrich Digital Service GmbH & Co. KG, pp. 2–4
8. Scholz, G., Scholz, G.: IT-Systeme für Verkehrsunternehmen. In: Informationstechnik im öffentlichen Personenverkehr. dpunkt.verlag, Heidelberg (2012)
9. Broy, M.: Functional safety based on a system reference model. In: Cant, T. (ed.) Australian System Safety Conference (ASSC 2012). Conferences in Research and Practice in Information Technology (CRPIT), vol. 145. Brisbane, 23–25 May 2012
10. Basili, V.R., Rombach, H.D.: Support for comprehensive reuse. Softw. Eng. J. **6**(5), 303–316 (1991)
11. Wing, J.M.: Computational thinking. Comm. ACM **49**(3), 33–35 (2006)
12. Broy, M.: Software and system modeling: structured multi-view modeling, specification, design and implementation. In: Hinchey, M., Coyle, L. (eds.) Conquering Complexity, pp. 309–372. Springer (2012)
13. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer, New York (2001)

http://www.springer.com/978-3-642-37394-7

Perspectives on the Future of Software Engineering
Essays in Honor of Dieter Rombach
Münch, J.; Schmid, K. (Eds.)
2013, XVI, 366 p., Hardcover
ISBN: 978-3-642-37394-7