Specification Modules for Methodical System Development

Franz Huber, Bernhard Schätz*
Fakultät für Informatik, Technische Universität München
Arcisstraße 21, D-80333 München, Germany
Email: {huberf|schaetz}@informatik.tu-muenchen.de

Abstract: We show how an alternative approach to document-oriented specification can ease the system design process. By identifying the modeling concepts needed for a complete system specification in an underlying conceptual model a specification can be interpreted as one single well-formed information complex avoiding inconsistencies as found in document oriented approaches. Besides a more intuitive development and treatment of specifications, an immediate benefit of a design process based on a conceptual model is the possibility to define specification modules that can be used as building blocks to assemble complex specifications.

1 Introduction

From a developer's perspective a major benefit of tool-supported system development should be an efficient management of specifications and a modular approach to development and reuse. Such modularity can be best achieved if tools allow developers to deal with the actual modeling concepts provided by a method instead of artifacts such as documents, which are still a basic modeling concept in many tools. Documents are basically an arbitrarily chosen grouping of descriptions or description parts providing partial information on specific views of a system. Maintaining the contained information, which is often spread throughout several of them and often bears many redundancies, is a laborious task both from the perspective of methodical specification development and from the *tool developer's* perspective. Even further, with respect to reuse of system descriptions this approach is not satisfactory since it is not always clear which specific documents are affected or need to be considered when reusing a particular system specification. From a tool-oriented perspective, a meta-model oriented approach, based on a conceptual model describing the elements of a method, has been recognized as an adequate way to provide an efficient internal structure to manage and store system specifications. Here, description techniques are regarded as visual representations of parts of the conceptual model.

In this paper, we show that such a formally based conceptual model can be useful not only for the implementation of efficient storage systems for CASE tools but as well in a methodical sense. Using a subset of a real development method for distributed systems, closely related to the concepts of the FOCUS methodology ([BDD+92], [BHS98]) and the AUTOFOCUS tool [HSS96], we first describe a simplified version of such a conceptual model capturing the method's basic modeling elements (Section 3). These elements—components, ports, channels, data types and data elements, control states, and the like—which are actually *specifications* or *descriptions* of them, are the terms in which developers characterize systems. Concrete syntactical notations, mostly on a graphical, partly on a textual basis, visually represent them. These notations, together with the represented modeling concepts and the relationships between them, make up *description techniques* in our approach. A specification can then be interpreted as a graph consisting of vertices of different types—the types of modeling elements from the conceptual model—and of edges pertaining to the relationships in

* The authors of this paper were funded by the sub-project A6 of the DFG Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen" and the "Forschungsverbund Software Engineering (FORSOFT)".

the conceptual model. The conceptual model describes which graphs, from the universe of all possible graphs of modeling elements, are valid, well-formed specifications. We give a more detailed characterization of the well-formedness, consistency, and completeness of specifications in Section 5. There, we also show how to define *specification modules* that can be reused in other specifications and how to apply these specification modules on the basis of the mathematical model of colored graphs. A short conclusion with an outlook to further work ends the paper.

2 Example: The Production Cell

Throughout this paper, we will use parts of the fault-tolerant production cell described in [Lötz96] as a running example. The production cell—as shown in Figure 1—processes metal blanks using two presses. Using its first arm, a robot picks up a piece from a rotary table fed by the feed belt and places it into one of two available presses. Using its second arm, processed pieces are picked out of the presses and transported to a deposit belt.

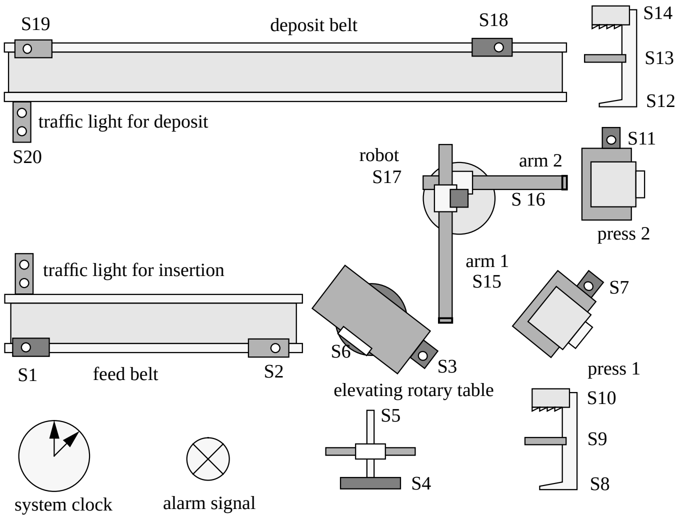

Figure 1. The Fault Tolerant Production Cell

Besides the actuation units (belts, arms, presses, etc.), the system has several sensory units signaling the state of the robot arms, presses, tables, blanks on a belt, and the like.

3 The Conceptual Model

This section introduces a simplified version of the conceptual model for describing distributed systems in our method. Its instances are abstract specifications describing systems. Developers create and manipulate them by means of concrete notations, which provide views upon them. Possibly even several different graphical or textual views on the same parts of the model can be offered. Some of the notations used have been introduced in [HSS96], and an example will be given in Section 4.

From a description technique-oriented point of view the elements in the conceptual model make up the essence (or the abstract syntax) of the information given by the notations used, quite similar to abstract syntax trees generated by parsers for common programming languages. In programming languages, however, source code “documents” are the important modeling concept and the syntax tree is just generated by the parser, in general unnoticeable from the user’s perspective. Many software engineering tools offer a similar approach, treating system descriptions as—at most loosely related—documents of different kinds. In our methodology, the abstract model is the central concept, both from the tool developer's and from the methodologist's point of view. Developers deal directly with the elements of the abstract model without encapsulation in artificial constructs such as documents. The modeling elements of our example method are shown in Figure 2 in a UML-style notation. For a more detailed description of the modeling concepts we again refer to [HSS96].

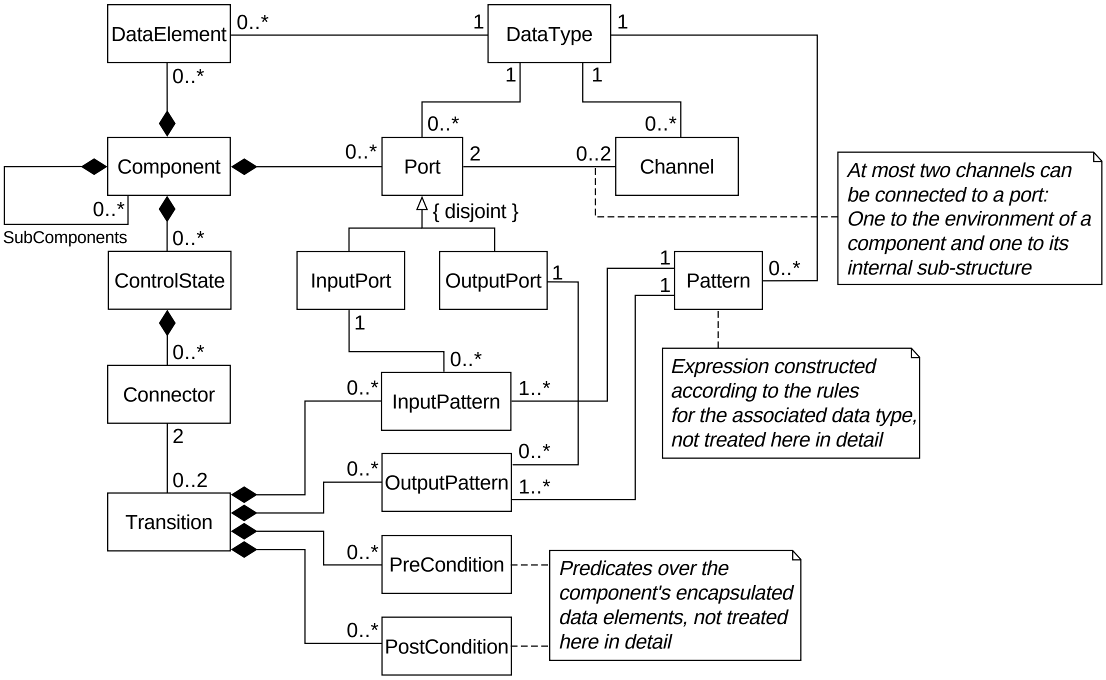
DataElement
0..*
1
DataType
1
0..*
1
1
0..*
0..*
Component
0..*
Port
2
0..2
Channel
0..*
{ disjoint }
SubComponents
0..*
ControlState
InputPort
OutputPort
1
1
1
Pattern
0..*
1
0..*
0..*
Connector
0..*
InputPattern
1..*
2
0..*
0..*
0..2
OutputPattern
1..*
Transition
0..*
PreCondition
0..*
PostCondition
At most two channels can be connected to a port: One to the environment of a component and one to its internal sub-structure
Expression constructed according to the rules for the associated data type, not treated here in detail
Predicates over the component's encapsulated data elements, not treated here in detail

Figure 2. Simplified Conceptual Model

Subsequently, we briefly describe the elements in the conceptual model.

Components are the basic elements. They are units encapsulating *data*, *structure*, and *behavior*, communicating with their environment. They are reactive; that is, they respond to stimuli from the environment according to their defined behavior. Components can be hierarchically structured, i.e., consist of a set of communicating sub-components.

Data types define data structures used by components. Data types are described in terms of product and summation types.

Data are encapsulated by a component and provide a means to store "persistent" state information inside a component. They are realized by typed state variables.

Ports are a component's means of communicating with its environment. Components read data on input ports and send data on output ports. Ports are named and typed, allowing only specific kinds of values to be sent/received on them.

Channels are directed, named, and typed. They connect component ports. They define the communication structure, i.e., the topology, of a distributed system.

Control States and **Transitions** between their entry and exit points—called **Connectors** here—define the flow of control within a component. Transitions carry four kinds of annotations that determine their firing conditions,

- pre-conditions and post-conditions, which are predicates over the data elements in the component that have to be fulfilled before and after the transition, respectively, as well as

- input and output patterns, determining which values have to be available on the component's input ports to execute the transition and which values are then written to the output ports.

Input and output values are given by patterns, like in functional programming languages, using the constructor terms available for the respective data types.

Behavior describes how a component reacts to input from its environment. This reaction is a result of the inputs from the environment and of the component's internal state. The latter can be partitioned into two separate aspects: the state of the *data* encapsulated by the component and the state of *control*, given by the component's internal state-based control flow description.

With respect to an underlying formal model, the elements in the conceptual model can be regarded as abstractions of both the formal model and the concrete notations used to describe them. Thus, the conceptual model represents the common denominator of both the description techniques and a formal model.

Viewing specifications as graphs, as sketched in Section 1, it is obviously possible to construct a multitude of such graphs when using only the elements and relationships in the conceptual model, leaving aside the arities given for the relationships. Then, of course, most of the possible graphs will not conform to the conceptual model.[1] In this respect, the conceptual model acts as a requirement specification for well-formedness (see Subsection 5.1.1), discriminating well-formed from ill-formed specifications.

4 Description Techniques – an Example

Using the structural decomposition of a component as an example, we briefly sketch informally our understanding of the term *description technique* in the context of this paper.

A description technique consists of

1. a (sub-)set of modeling elements given in the conceptual model,
2. a concrete syntax—a graphical or textual notation—describing these elements, and
3. a set of rules how the concrete syntax is mapped to the modeling elements and vice-versa.

As shown in Figure 2, components may be structurally decomposed into networks of sub-components. Such a network can be graphically described by a system structure diagram (SSD, [HSS96]). An example is given in Figure 3, which shows a very simplified network of two components in our production cell, the system clock and the production cell controller component with a subset of their communication channels.

For this description technique, the relevant modeling elements are Component, InputPort, OutputPort, Channel, and DataType as well as their associations (see Figure 2). Graphical representations of these elements are rectangles for components, hollow and filled circles for input and output ports, and directed arcs for channels. All of these graphical elements—as well as their counterparts in the conceptual model—have a name to designate them (invisible in case of the ports). Data types are represented by their names as strings. Both channels and ports have a data type assigned, again invisible in the case of ports.

[1] Although well-formedness is considered an invariant in the development process (see Subsection 5.1.1), allowing ill-formed specifications can be reasonable for some, mostly internal, operations on specifications invisible to the user (see Subsection 5.4.2).

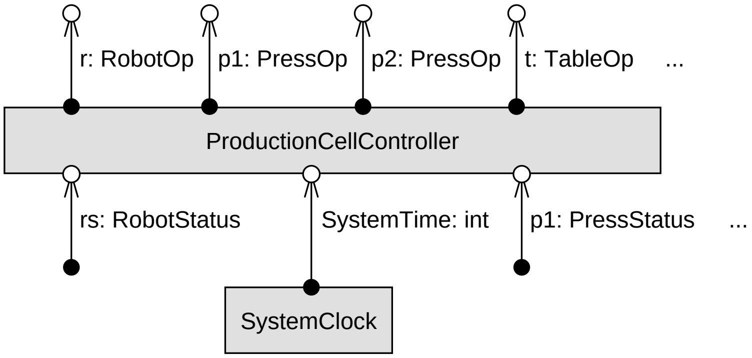

Figure 3. Component Network Description Using an SSD

Most of the mapping rules between the graphical representation and the modeling elements are elementary. For instance, it is quite obvious that the names and types of the modeling elements and their graphical representations must correspond; in case a developer changes a name, the name in the modeling element has to be changed accordingly. The relationships between modeling elements and syntactical elements are generally simple 1:1 relationships. There are, however, a number of more complex relationships. Adding, e.g., an additional communication path from the production cell controller to the environment (another channel with an unconnected destination port, see upper part of Figure 3) obviously has to generate the corresponding modeling elements and add the source port to the set of output ports of the controller. However, this operation also changes the port set of the *enclosing* component, since this view presents both its decomposition aspect and its interface aspect (given by the port set).

Additionally, a textual notation could be used as a concrete syntax to specify the same aspects of component networks, such as the language ANDL, defined in [SS95]. Both representations could then be used interchangeably for the same modeling elements.

Technically, such an update mechanism between modeling elements and their visual representations can be implemented according to a Model-View-Controller (MVC) pattern [GHJV94].

5 Specification Modules

The conceptual model introduces the terms and relations needed to describe the system specified by the system developer. In this section we show how those terms and relations are combined to form specifications. Furthermore, we show that this formulation of a specification naturally extends to the notion of specification modules and illustrate the definition and application of those specification modules.

In Subsection 5.1 we define when a description of a system using the conceptual model is considered a system specification. In Subsection 5.2 we show how specifications and specification modules are related and what it means to define an incomplete specification module. Finally, in Subsection 5.4 we demonstrate how specification modules are applied to support reuse of specifications.

5.1 Module Criteria

To support reuse of specifications or specification parts, a clear meaning of a specification has to be defined. Based on the conceptual model discussed above we introduce the notion of a *specification* of a system. A *specification*

1. is a *well-formed* description of one or more aspects of a system,

2. may fulfill additional *consistency conditions* (see [HSE97]),
3. does not necessarily need to be *complete*.

Like the conceptual model itself, a specification is an abstract concept and can have several concrete syntactical representations to manipulate it. The choice of the properties of the conceptual model determines the notion of a specification by defining the *well-formedness* and the *consistency* of a specification. The first describes invariant conditions of a specification, the latter conditions required only at certain development steps. Since the distinction between a well-formedness condition and a consistency condition depends on the definition of the conceptual model, this definition—as a methodological decision—influences the strictness of the design process. For example, the assignment of a data type to a port or channel may be considered a well-formedness condition as well as a consistency condition. In the first case the creation of a port or a channel is not possible without possibly defining and assigning an appropriate type. In the latter case, a type may be assigned at a later step in the design process.

5.1.1 Well-Formedness

Well-formedness conditions are invariant conditions that hold for specifications invariantly throughout the design process. Those invariances are defined by the conceptual model and typically include syntactic properties.

Examples are:

- Each specification entity (component, port, channel, data type, etc.) has a non-empty name.
- Each channel has two adjacent ports, an output port at its beginning, and an input port at its end, and an associated data type.
- Each transition has two adjacent connectors.

5.1.2 Consistency

Consistency conditions are defined as additional properties of the conceptual model that must hold for formally well-defined specifications, but may be violated throughout the design process. However, at certain steps in the design process, consistency of the specification is required. Typical steps are code generation, simulation, validation, verification, and specification module definition. Different consistency conditions may be required for different design steps. While code generation or simulation require completely defined data types, this is not necessary for verification or specification module definition.

Examples are:

- Each port, channel, etc. has a defined (i.e., non-empty) type.
- Port names are unique for each system component.

While the first condition is a necessary consistency condition for simulation or code generation, the second consistency condition is not required by any design step suggested in this paper; it may, however, be formulated and checked to support better readability or clarity of documents generated from the conceptual model.

5.1.3 Completeness

A third condition to be raised throughout the development process but not mentioned so far is the *completeness* of a specification. A specification is considered to be complete if all relevant objects of the specification are contained in the specification itself.

Similar to the consistency of a specification, completeness is only required at certain steps of the development process like simulation, code generation or verification. As a matter of fact, completeness of a specification can be defined as a consistency condition and checked the same way (see [HSE97]). However, the incompleteness of a specification module can be used to define parameterized modules, and is thus treated as a separate property for methodological reasons: since an instantiation mechanism is needed for incomplete or parameterized specification modules, incomplete modules are distinguished form other forms of inconsistency. Subsection 5.2.2 treats this question in more detail.

5.2 Module Definition

Given the modeling concepts introduced above, the notion of a specification module can be introduced. Typical examples of specification modules needed for a system specification are:

- Data type specification modules: The data type definition part of a specification or a sub-part of it.
- System structure specification module: A specification module of a system as defined by a corresponding component possibly including its sub components.
- Behavioral specification module: The behavior assigned to a component or a subpart of it.

In our approach we will not distinguish between different classes of specification modules. Furthermore, a specification module is *not* distinguished from a specification in general. Therefore, basically every well-formed part of a specification is considered a specification module. Additionally, a well-formed part of a specification does not necessarily have to be complete. However, for a reasonable reuse of a specification module, it has to obey several consistency conditions as introduced in [HSE97]. This leads to a simple distinction of two different specification module concepts*:*

- *Complete specification module:* A specification module is considered to be *complete* if all referenced elements of the conceptual model (e.g., type definitions of used port types or local data, sub-components of a component) are contained in the module.
- *Parameterized specification module:* A specification module is considered to be *parameterized* if some referenced elements of the conceptual model are not included in the specification (e.g., incomplete type definitions of a component, undefined behavior of a component).

Specification modules are defined as well-formed specification parts possibly obeying additional consistency conditions. Therfore, specification modules can either be developed as in the case of a usual specification or reused form a larger specification by a selection process based on the conceptual model.

5.2.1 Complete Modules

The simplest form of a specification module concerning reuse is the one containing all relevant information. Since—unlike in the parameterized case—no more instantiating of the module is needed, all information of the module can simple be added o the target specification.

A simple example of a complete specification module is the press controller module consisting of

- the data types describing the actuatory and sensory data,
- the interface description of the controller consisting of typed input and output ports,
- the (empty) list of variables of the controller unit, and

- the behavioral description of the controller unit given by a state transition diagram using the typed port, component, and transition variables of the controller.

This module is complete since all entities referenced in this module (data types, ports, variables, etc) are also defined in this module.

5.2.2 Parameterized Modules

As a simple example we define a behavioral specification module as shown in Figure 4. The module, described using a state transition diagram, is used to cope with fault situations of the production cell units. Upon entering the module, an error report is generated. The unit is then brought to a defined state and stability of the unit is reported. Finally, upon receipt of a restart message, the unit is restarted in normal mode. Since all of the units of the production cell must support this kind of error treatment, it is useful to define a general fault correction module that can be instantiated for all components.

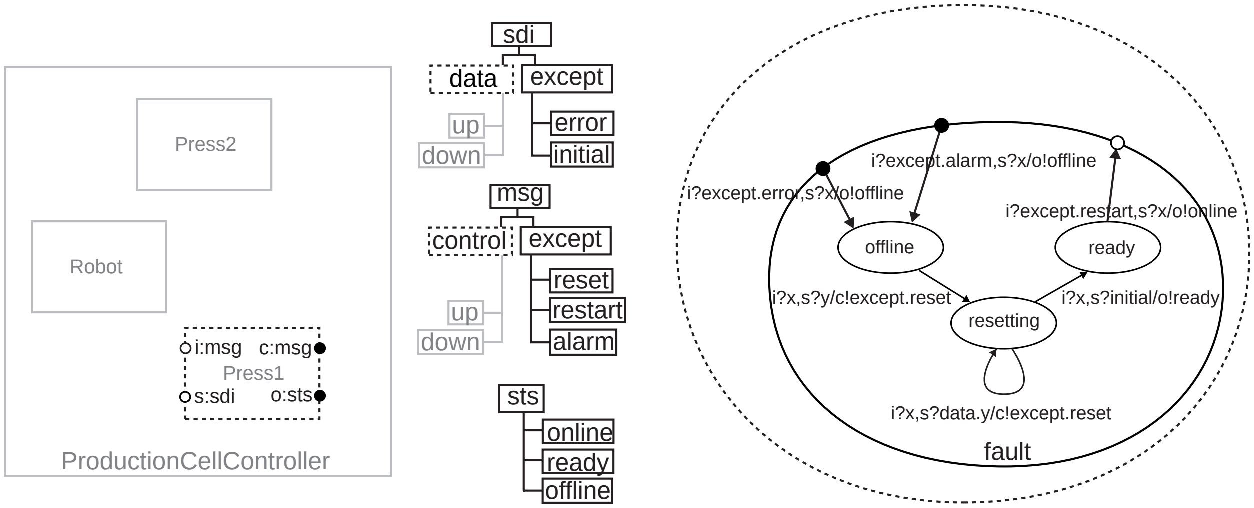

Figure 4. Parameterized Error Treatment Module

The defined specification module consists of three parts:

- The specification of the minimal interface that must be supplied by a system component intended to offer this fault recovery strategy. The interface is defined by the corresponding ports (i, o, c, s).
- The specification of the type of messages that are used to indicate the status of such a fault-correcting unit and to influence its behavior. Since the behavior is independent of the kind of unit (press, rotating table, robot arm, etc.), the unit-dependent part of the message types (data, control) is not specified by the specification module but defined as a specification module parameter to be instantiated upon use of the module.
- The specification of the behavior relevant for fault recovery. Only a part of a complete behavioral description of a production cell unit is described by giving the necessary states (offline, resetting, ready), and the corresponding transitions and entry- and exit-points. Thus applying the module only adds the fault-recovery routine to the target specification which has to be extended to a complete specification for the corresponding unit.

To apply a parameterized specification module to a target specification the parameters must be instantiated and the resulting module is added to the specification. In Section 5.3 we give a more precise definition of the concepts of specification modules and their elements; in Section 5.4 we show how to apply complete or parameterized modules to a target specification.

5.3 Mathematical Model

In the previous sections we gave an intuitive interpretation of the terms *specification* and *complete* and *parameterized specification module*. However, to introduce the application or reuse of specification modules we need a more precise definition of those terms. Therefore, we define a mathematical model for the above-introduced concepts.

In Subsection 5.3.1 we define the mathematical concept of specification modules and of their combination using the notion of colored graphs and typed binary relations. Based on this model we will introduce the necessary formal operations *union construction* and *renaming*, which will be used to define the module application in Section 5.4.

5.3.1 Model of Specification Modules

A specification is considered a graph with the specification elements of the conceptual model as nodes and the relations between these elements as the edges of the graph.[2] Since the conceptual model is typed (the elements of the conceptual model are elements of distinct classes like components, ports, channels, etc.), the graph is colored. Thus a specification or specification module can be described by a pair *(E,R)* with

- a collection of sets of elements $E = (E_1, E_2, \ldots, E_{m-1}, E_m)$
- a collection of binary relations $R = (R_1, R_2, \ldots, R_{n-1}, R_n)$

with $E_i \subseteq \mathcal{E}_i$ and $R_j \subseteq \mathcal{R}_j$, where $\mathcal{R}_j = \mathcal{E}_k \times \mathcal{E}_l$, as well as corresponding definitions for $\mathcal{E}$ and $\mathcal{R}$. The definition of $\mathcal{E}_i$ and $\mathcal{R}_j$ depends on the definition of the conceptual model as described in Section 3. In the AUTOFOCUS conceptual model, for example,

- the collection $\mathcal{E}$ contains the set of input ports $\mathcal{I}$, the set of system components $\mathcal{S}$, the set of channels $\mathcal{C}$ or the set of types $\mathcal{T}$, and
- the collection $\mathcal{R}$ contains the relation $\mathcal{SS}$ between a system component and its sub components, the relation $\mathcal{IC}$ between an input port and connected channel, $\mathcal{IT}$ between an input port and its associated type as well as $\mathcal{CT}$ between an channel and its associated type.

In general, R_i will not cover the complete range of sub relations of $\mathcal{R}_{\mathrm{i}}$. For example, if an input port is part of a specification module, it also has a defined type

$$\forall i : I. \exists t : T.(i,t) \in IT$$

- since a system component cannot be its own sub component, the sub component relation will not contain the identical relation

$$\forall x : S, y : S.(x,y) \in SS \Rightarrow x \neq y$$

- since it is not possible to connect one port to two input channels, the channel-input port relation will not contain two different channels for one port:

$$\forall i : I, c_1 : C, c_2 : C.(i,c_1) \in IC \wedge (i,c_2) \in IC \Rightarrow c_1 = c_2$$

[2] For reasons of simplicity we will only consider binary relations; relations of higher cardinality can be reduced to binary normal form.

Furthermore, the collection of relations will not cover the complete range of possible sets of relations fulfilling those above conditions. For example, if a pattern is defined for an input port in a state transition diagram, both pattern and input port are of the same type

$$\forall i : I, p : P, t1 : T, t2 : T.((i,t1) \in IT \wedge (p,t2) \in PT \wedge (i,p) \in IP) \Rightarrow t1 = t2$$

Those additional conditions fulfilled by the pair *(E,R)* of a specification module represent the well-formedness or consistency conditions described in Subsection 5.1. Some of those conditions as described above are typically described using arity-annotations of E/R diagrams. Those conditions can be expressed in typed first-order predicate logic with equality and can thus be automatically checked by a consistency checker as described in [HSE97].

5.3.2 Operations on Specification Modules

To combine two specification modules $(E,R)$and (E',R') as described above, simply the union $(E \cup E', R \cup R')$ is constructed with

$$E \cup E' = (E_1 \cup E_1', E_2 \cup E_2', \ldots, E_m \cup E_m')$$
$$R \cup R' = (R_1 \cup R_1', R_2 \cup R_2', \ldots, R_n \cup R_n')$$

Note that in general well-formed but otherwise arbitrary specification modules are used in constructing the union. Especially, E and E' are neither required to be disjoint nor to be identical. Thus, the union of two specification modules can

- introduce new specification elements like new components, ports, channels, types, states, etc.
- introduce new associations to the relations between both old and new specification elements like adding a new port to an already existing system component, adding a collection of new sub-states to an already existing state, etc.

It is important to note, however, that the union construction of two well-formed or consistent non-disjoint specification modules in general will not lead to a well-formed or consistent specification. Subsection 5.4.2 will consider this aspect.

Finally, specification modules can be renamed prior to the union application to allow the identification of specification elements. Thus, parameterized specification modules can be applied to specifications. To rename specification modules, an isomorphic mapping $M : \mathcal{E} \times \mathcal{R} \to \mathcal{E} \times \mathcal{R}$ is defined with $M = (M_{\mathcal{E}_1} \times \ldots \times M_{\mathcal{E}_m}, M_{\mathcal{R}_1} \times \ldots \times M_{\mathcal{R}_n})$, $M_{\mathcal{E}_i} : \mathcal{E}_i \to \mathcal{E}_i$, and $M_{\mathcal{R}_j} : \mathcal{R}_j \to \mathcal{R}_j$ as well as

$$M_{\mathcal{R}_i}(R_i) = \{(M_{\mathcal{E}_j}(e1), M_{\mathcal{E}_k}(e2)) \mid (e1,e2) \in R_i \wedge R_i \subseteq \mathcal{E}_j \times \mathcal{E}_k\}$$

Based on the techniques of renaming and union construction we will describe how a specification module can be applied as a complete or parameterized module in the following section.

5.4 Module Application

Basically, the application of a specification module can be defined as an embedding operation on the conceptual model with additional mappings of common elements of the module and the specification. Therefore, in Subsection 5.4.1 we outline the renaming as the basic difference between the application of a complete and a parameterized specification module. Subsection 5.4.2 sketches how such a renaming mapping is used to define parameterized modules using the example of Subsection 5.2.2.

5.4.1 Complete and Parameterized Specification Modules

As mentioned in Subsection 5.2 we distinguish between parameterized and complete specification modules. However, having introduced a mathematical model for specification modules, it becomes obvious that this distinction is not a technical but only a methodical one. To add a *complete* specification module we simply construct the union of both as defined above. Assuming disjoint sets of specification elements no further renaming of the added specification module is necessary.[3]

For example, we can simply add the press controller module defined in Subsection 5.2.1 to the specification to add another press to the system. To make use of the controller module we then must connect the ports of the module to the ports of the system.

On the other hand, to make use of a parameterized specification module it is necessary to instantiate the parameters of the module before adding it to the system specification. Therefore, a renaming of the parameter elements of the specification module to elements of the target specification has to be applied prior to the union construction. The parameter elements of the specification can be considered the interface of a specification module that is used to apply the module to the target specification. Thus, module interfaces can be compared to specification parameters as found in algebraic specification languages like SPECTRUM [GN94]. However, even more sophisticated techniques are definable using the consistency condition mechanism since the mathematical framework does not distinguish between parameters or regular elements of a specification module.

Again, we consider the example of the press controller module defined in Subsection 5.2.2. The specification can be used as a behavioral specification module with type parameters (control and data), a system component parameter (Press1) and a state parameter (fault). To avoid the introduction of new types for the actuatory and sensory data we identify the types used in the press controller module with the types already defined in the system specification.

5.4.2 Module Instantiation

As mentioned in Subsection 5.4.1, specification modules can be compared to (parameterized) algebraic specifications. Thus, the combination of specification modules is similar to the combination of algebraic specifications: elements of the interface of the applied specification module are identified with elements of the specification (or specification module) the module is applied to. Thus, to apply a specification module, a mapping must be constructed to map the interface elements to elements of the same type in the target application. Furthermore, the resulting specification must again be well-formed.

To illustrate module application we consider the module introduced in Figure 4. Here, the mapping

- introduces new port elements (i,o,s,c), new type elements (msg, except, alarm, etc.), new state elements (offline, resetting, online), as well as the new transition elements and pattern elements found in Figure 4,
- identifies old and new elements like the system component Press1, the type elements data or control and the state fault, and therefore
- introduces new relations, like the component-port relation between Press1 and i, or the state-sub state relation between fault and offline.

[3] The disjointness condition might be relaxed to support the common use of predefined data types like bool or int as well as identifiers of specification elements.

Figure 4 shows the resulting specification after the mapping and the union construction including the newly introduced elements, the already defined elements (grayed out) and the identified elements (dashed).

6 Conclusion and Outlook

We have outlined an approach to specification development closely related to CASE tool design principles. Using the same approach—a conceptual model—as a basis both for tool development and for the methodological basis, a schematic way of dealing with specifications can be applied. In CASE tool design, the suggested, model- and view-based approach helps to avoid many redundancies and sources of inconsistency commonly found in document-based approaches. This results in major benefits with respect to a clear tool design. But advantages are to be expected from the methodological point of view as well. Working directly with the modeling concepts of a method—in our view—yields a more natural way of system development. Furthermore, we showed that a simple mathematical model can be used to define the necessary operations for the definition and the reuse of specification modules.

To validate the practicability of the suggested approach the conceptual model and the defined operations will be implemented in the formal development tool prototype AUTOFOCUS.

Acknowledgements

We thank Annette Lötzbeyer for letting us use the picture from Figure 1.

7 Bibliography

[BDD+92] M. Broy, C. Dendorfer, F. Dederichs, M. Fuchs, T. Gritzner, and R. Weber. The Design of Distributed Systems - An Introduction to Focus. Technical Report TUM-I9225, Technische Universität München, 1992.

[BHS98] M. Breitling, U. Hinkel, K. Spies. Formale Entwicklung verteilter reaktiver Systeme mit FOCUS. In this collection.

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Micro-Architectures for Reusable Object-Oriented Design. Addison-Wesley, 1994.

[GN94] R. Grosu, D. Nazareth. The Specification Language SPECTRUM – Core Language Report V1.0. Technical Report TUM-I9429. Institut für Informatik. Technische Universität München, 1994.

[HSE97] F. Huber, B. Schätz, G. Einert. Consistent Graphical Specification of Distributed Systems. In: J. Fitzgerald, C. B. Jones, P. Lucas (Eds.). Proceedings of FME ’97, 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science 1313, Springer, 1997.

[HSS96] F. Huber, B. Schätz, K. Spies. AutoFocus – Ein Werkzeugkonzept zur Beschreibung verteilter Systeme. pp. 165-174. In U. Herzog, H. Hermanns (Eds.). Formale Beschreibungstechniken für verteilte Systeme. Universität Erlangen - Nürnberg, 1996.

[Lötz96] A. Lötzbeyer. Task Description of a Fault-Tolerant Production Cell. Forschungszentrum Informatik, Karlsruhe. 1996. URI: http://www.fzi.de/prost/projects/korsys/korsys.html.

[SS95] B. Schätz, K. Spies. Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik. Technical Report TUM-I9529. Institut für Informatik. Technische Universität München, 1995.