Integrating Formal Description Techniques

Bernhard Schätz and Franz Huber*

Fakultät für Informatik, Technische Universität München,
Arcisstraße 21, 80333 München
Email: {schaetz|huberf}@in.tum.de

Abstract. Using graphical description techniques for formal system development has become a common approach in many tools. Often multiple description techniques are used to represent different views of the same system, only together forming a complete specification. Here, the question of the integration of those description techniques and views becomes a major issue, raising questions of consistency and completeness. In this paper, we present an approach to ensuring conceptual and semantic consistency, influenced by experience gained from a first implementation of the AUTOFOCUS tool prototype. Finally, we show how this approach forms the basis for the definition of specification modules.

1 Introduction

Using multiple description techniques has become a common approach for the tool-based system development. Prominent examples are SDL-based tools (e.g., ObjectGeode [19], SDT [18]) and automata-based approaches (e.g., ObjecTime [17]). Here, the specification of a system is spread out over several documents, each one describing a certain view of the system, like its structure, its behavior, its data types, or some of its sample runs. Only by combining those views we obtain the complete system specification. However, while this structuring mechanism makes specifications more readable and manageable, it also poses a major problem: inconsistencies may arise, for example by

- conflicts between the external and internal interface of a system or component,
- conflicts between the behavior of a system and of its combined subsystems, or
- conflicts between the specified behavior and given sample runs of a system.

To form a reasonable specification, inconsistencies between those views must be avoided. Thus a tool should support detecting and fixing those inconsistencies. In other words, for the usability of a tool supporting a view-based specification method, the integration of those views is a prime requisite. This article describes an approach towards this integration within the tool prototype AUTOFOCUS.

* This work was carried out within the sub-project A6 of "Sonderforschungsbereich 342 (Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen)" and the project SysLab, sponsored by the German Research Community (DFG) under the Leibniz program and by Siemens-Nixdorf.

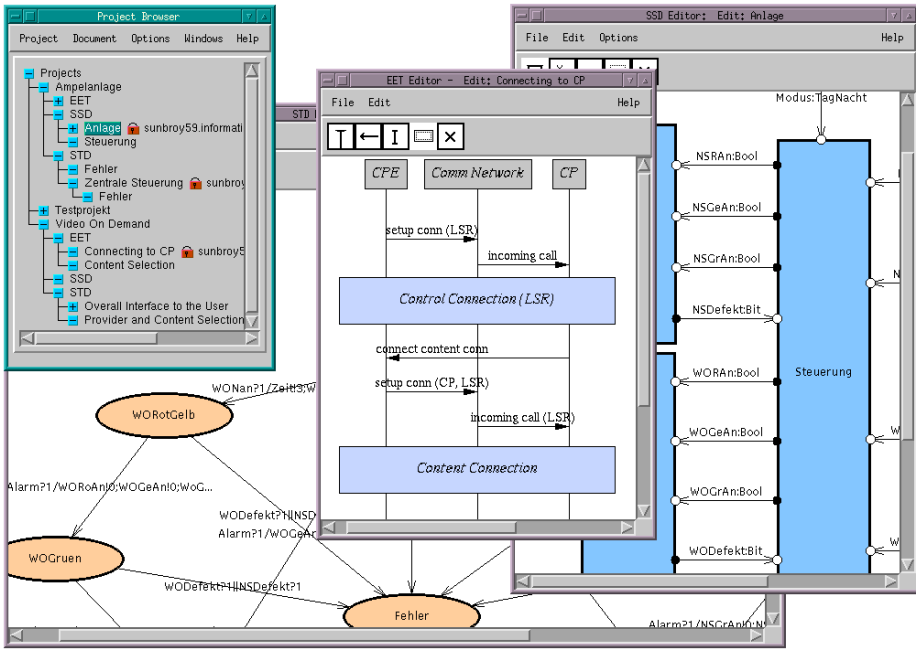

Fig. 1. The AUTOFOCUS Client Application – Project Browser and Editors

The remainder gives a short introduction of the AUTOFOCUS description techniques. Section 2 explains our notion of integrated formalisms, sections 3 and 4 sketch the idea of a conceptual model to base those formalisms on. Section 5 illustrates the capabilities of a conceptual model for the reuse of. Finally, section 6 concludes the approach with a short summary and outlook. As mentioned throughout the article, part of this introduced approach is already implemented in the current version of AUTOFOCUS, the remainder describes work in progress.

1.1 View-based Systems Development

To support the development of distributed systems, AUTOFOCUS [11] does not aim at capturing a complete system within a single formalism. Instead, different views of a system are each specified using an appropriate notation. In the AUTOFOCUS approach, a distributed system is characterized from several points of view, as

- the structure of a system including its components and channels,
- the behavioral description of the system as a whole or of one of its components,
- the data processed by the system and transmitted across the channels, and
- the interaction of the components and the environment via message exchange.

In general, only a description including all views forms a complete picture of the system. Thus, AUTOFOCUS offers multiple description techniques: system structure diagrams (SSDs), state transition diagrams (STDs), data type definitions (DTDs), component data declarations (CDDs), and extended event traces (EETs), covering all those aspects. Like the hierarchical concepts of the underlying theory FOCUS [4], each

description technique allows to model on different levels of detail, where, for example, components can be either atomic or consist of sub-components themselves.

1.1.1 Document Oriented Description

In AUTOFOCUS, a project, representing a system under development, consists of a number of documents that are representations of views using the description techniques introduced above. Thus each description technique is mapped to a corresponding class of documents ("diagrams"). Combined, these documents provide a complete characterization of a system in its current development status.

1.1.2 Hierarchical Documents

All graphical AUTOFOCUS description techniques share the concept of hierarchy. SSDs, STDs and EETs allow hierarchical decomposition. In an SSD, a system component may be viewed as a conceptual unit of sub-components specified in another SSD. In the same way, a state in an STD can be characterized by another STD document describing this state on a more detailed level. In EETs, so-called "boxes" are introduced as an abbreviating notation for partial runs specified in different EETs.

1.1.3 Integrated Documents

From the user's point of view, the documents of a development project are integrated, both vertically along the refinement hierarchies and horizontally along the relationships between documents of different kinds. For instance, an STD can be associated with a component in an SSD denoting that this STD specifies the behavior of the component. Along relationships like these, quick and intuitive navigation mechanisms between the documents are available.

1.1.4 System Structure Diagrams (SSDs)

System structure diagrams describe a distributed system as a network of components exchanging messages over directed channels. Each component has a set of input and output ports to which the channels are attached. Channels have associated data types describing the sets of messages sent across them. Components can be hierarchically refined by networks of sub-components. Then, the complete sub-network has the same set of communication ports as the higher-level component this refined view belongs to. Graphically, as in Fig. 1, SSDs are represented with boxes as components and arrows for channels. Both are annotated with identifiers and, in the case of channels, also with their data types. Input and output ports are visualized as small hollow and filled circles, respectively.

1.1.5 State Transition Diagrams (STDs)

State transition diagrams are extended finite automata similar to the concepts introduced in [8]. They are used to describe the behavior of a system or component. Each component can be linked to an STD consisting of states and transitions between them. Each transition has a set of annotations: a pre- and post-condition, encoded as predicates over the data state of the component satisfied before and after the transition, and a set of input and output patterns describing the messages read from or written to the

input and output ports. For hierarchical refinement of states in STDs, we use a concept similar to the SSD case. Graphically, automata are represented as graphs with labeled ovals as states and arrows as transitions. Fig. 1 shows an example of an AUTOFOCUS state transition diagram.

1.1.6 Datatype Definitions (DTDs)

The types of the data processed by a distributed system are defined in a textual notation. We use basic types and data type constructors similar to those found in the functional programming language Gofer [14]. The data types defined here may be referenced from within other development documents, for example, as channel data types in SSDs.

1.1.7 Component Data Declaration (CDDs)

Additionally to receiving and sending messages, components generally store information locally to process those messages. For this purpose, local variables may be defined for each component by associating a component data declaration to it. A CDD simply consists of a set of variable identifiers and their associated types as defined in the DTD of the system, plus a possible initial value. Those variables locally defined for a component may be addressed in the definition of the STD of this component in the input and output patterns as well as in the pre- and post-conditions.

1.1.8 Extended Event Traces (EETs)

Extended event traces (cf. [16]) describe sample system runs from a component-based view. As shown in Fig. 1, we use a notation similar to the ITU-standardized message sequence charts MSC'96 (ITU Z.120, [13]). Using “boxes” EETs, support hierarchy and specify variants of behavior. Indicators can be used to define optional or repeatable parts of an EET. From a methodological point of view they are used in the early stages of systems development to specify the functionality of a system on a sample basis as well as system behavior in error situations. Later in the development process, the system specifications given by SSDs, STDs, and DTDs can be checked against the EETs, whether they fulfill the properties specified in them.

2 Integration of Formalisms

We use the term *integration of description formalisms* to express the influence of the description formalisms on each other. To judge the integration we must answer the question “How well do the formalisms play together to form a reasonable description of the system to be developed?” In other words, “What are the necessary side conditions on the formalisms to form a consistent system specification?” Here, we use “consistency” in a rather general interpretation (cf. [10]) to express several forms of conditions like:

- **Document Interface Correctness:** If a document is hierarchically embedded into another one, these documents must have compatible interfaces (components in SSDs, states in STDs, or boxes in EETs).

- **Definedness:** If a document makes use of objects not defined in the document itself, those objects must be defined in a corresponding document (like channel types in SSDs or STDs).
- **Inter-View Consistency:** If two or more formalisms describe the same aspect of a system, the descriptions must be consistent (like SSDs and STDs with EETs).
- **Completeness:** All necessary documents of a project have to be present.

From a methodological point of view we distinguish two kinds of conditions:

- **Conceptual Consistency Conditions** can be defined solely in terms of the description technique concepts. Examples are the interface consistency, the definedness conditions or the well-typedness of specifications.
- **Semantical Consistency Conditions** can only be defined using semantical notions. Examples are the refinement of a system including its behavioral description by an implementation through a set of sub-components including their behavioral description; or the compatibility of a sample EET of a system with the behavioral description of is sub-components.

Most developers expect the first class of conditions to generally hold during the development process. Luckily, there are simple mechanisms to check their validity (cf. Subsection 2.1). The second class is quite the opposite: very complex mechanisms are needed to validate those semantical consistency conditions (cf. Subsections 2.2.1 and 2.2.2), if possible at all. Since those conditions are quite complex, however, developers generally do not expect them to hold throughout the development process.

The distinction between conceptual and semantical consistency conditions plays an important role with the introduction of the conceptual model as described in section 3. There we introduce a notion of a specification based on a general conceptual model for AUTOFOCUS instead of a collection of AUTOFOCUS description techniques.

2.1 Conceptual Consistencies

Conceptual consistency conditions generally are considered to hold invariantly during the development process. However, mainly due to its originally document-oriented approach, AUTOFOCUS so far does not strictly enforce conceptual consistency throughout the development process. It rather offers developers the possibility to check the violation of conceptual consistency conditions and locate those elements of the specification causing these violations. With AUTOFOCUS, conditions can be formalized using a consistency condition description language based on a typed first-order logic. The language and the user interface are described in [10] and [6]. Experiences with AUTOFOCUS and this form of conceptual consistency conditions, however, suggest using a more rigorous approach, as discussed in section 2.3

2.2 Semantic Consistencies

Integration of views on the semantical level is more complicated. Those consistency conditions can only be expressed by proof obligations defined using the semantical basis. Thus a formal semantics of the description techniques and a sufficiently powerful proof-mechanism are needed. Providing a semantical basis for intuitive graphical

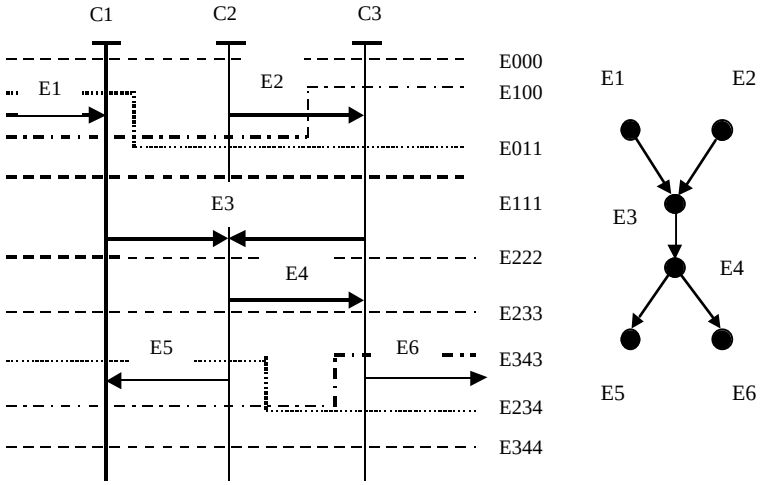

Fig. 2. Reachable Cuts of an EET and its Partial Order Trace

description techniques is becoming more and more state of the art (cf. StateChart, Rhapsody [12], ObjectGeode, SDT [18]). However, strong verification support, especially automatic support, is commonly found only in more mathematically oriented description techniques (e.g., Atelier B [1], FDR [7]). For a strictly integrated use of those user-friendly description techniques it is not sufficient to support a translation into a format suitable for a prover. For an integrated approach we furthermore require a tool to support the user on the level of those description techniques.

2.2.1 Hierarchical Consistencies

In the AUTOFOCUS approach only one form of hierarchical consistency on the semantic level is required: behavioral refinement of systems. Consistency violation may occur if a component is assigned behavior using a corresponding STD; furthermore a substructure is defined for this component via an SSD with STDs for all the components of this SSD. The AUTOFOCUS approach requires the behavior of the refined component (substructure components and their behaviors) to be a refinement of the behavior originally associated with the component. To check the validity of the refinement and thus the semantical consistency AUTOFOCUS offers an automatic check based on the relational μ calculus model checker μcke [3]. The formal basis for this check can be found in [10].

2.2.2 Inter-View Consistencies

To ensure semantic inter-view consistency in AUTOFOCUS, again, only one form of check is required. Since the combination of SSDs and STDs on the one hand and EETs on the other hand characterize system behavior, we require sample runs of the EETs to be a legal behavior as described by the SSD/STD combination. Again, we use μcke to check the consistency of the SSD/STD and the EET view.

As mentioned in Section 1.1.8, EETs are interpreted as positive or negative requirements in the form of sample runs. Formally, an EET is interpreted as a relation between two system states, relating a state before the execution of the EET to a state after the execution of all EET events. Thus, its μ formalization makes use of the product state and product transition relation of all components of the described system as described in [10]. Thus, an EET is interpreted as a state transition diagram of the complete system. *Reachable cuts* are used as the states of this diagram, defined using the independence expressed by the EETs. Events are considered independent except in two cases:

- All send and receive events of one component (represented as start or end point of arrows on a single component axis) are causally dependent in the downward direction to represent the passing of time.

Send and receive events of a single message event are causally dependent.[1] Fig. 2 shows the different reachable cuts of an EET consisting of components C1, C2 and C3.[2] Since a reachable cut marks a possible intermediate state of an EET, we define a *reachable cut* by the following rules:

- The start state of an EET (before any event) is a reachable cut.
- If a reachable cut is followed by a set of independent actions, all states reached by the execution of any subset of this set are reachable cuts.

As shown in Fig. 2, an EET can be equivalently expressed using a partial order trace (cf. [5]). Here, with independent E1 and E2, we obtain the reachable cuts:

- E000 is reachable by the first rule.
- E100 reachable from E000 through E1 by the second rule,
- E011 reachable from E000 through E2 by the second rule,
- E111 reachable from E100 through E2, from E011 through E1, or from E000 through simultaneously applying E1 and E2 by the second rule.

With E3 being dependent on E1 and E2, the next reachable cut is E222, and – similarly – E233. Analogously E5 and E6 lead to the cuts E343, E234, and E344.

For the μ formalization, we introduce a relation for each reachable state. A relation corresponds to executing pending events and reaching the associated cuts. For each execution of a set of events a clause is introduced reflecting the transfer of messages v_i on channels c_i and the reached cut EET_i. Furthermore, a clause is introduced with empty messages $c_i = nil$ representing a "nil round" of the system with no messages sent. Finally, all clauses are combined using disjunction:

$$\begin{aligned}\mu EET(s,s') \equiv\ & (s.c_1 = nil \wedge \ldots \wedge s.c_n = nil \wedge \exists t.(T(s,t) \wedge EET(t,s')) \vee \\ & (s.c_1 = v_{1,1} \wedge \ldots \wedge s.c_n = v_{1,n} \wedge \exists t.(T(s,t) \wedge EET_1(t,s')) \vee \\ & \vdots \\ & (s.c_1 = v_{1,k} \wedge \ldots \wedge s.c_n = v_{n,k} \wedge \exists t.(T(s,t) \wedge EET_k(t,s'))\end{aligned}$$

Here, the μ operator is used to define the relation as the least fixed point of this recursive definition. T corresponds to the transition relation of the system composed of C1, C2 and C3, formalized as product transition relation as in [10].

---

[1] Here send/receive events of a message are considered to happen simultaneously and are therefore interpreted as a single event.

[2] For the sake of brevity, we use labels E1, E2, ... instead of regular annotations.

To formalize hierarchic elements of an EET, a single relation is introduced for those elements. To formalize those sub-parts the above strategy is applied to them. Depending on the kind of hierarchical structuring mechanism (indicators/boxes) those sub-EETs are inserted in the relation of the embedding EET analogously to the execution of a single event:

$$EET(s,s') \equiv \exists t.(EET_b(s,t) \wedge EET'(t,s'))$$

with $EET_b(s,t)$ denoting the relation for the embedded EET constructed as described below, and $EET'(t,s')$ denoting the relation for the remainder of the embedding EET. The relation for an EET embedded using an indicator is constructed depending on the kind of the indicator (optional/repetitive/optional repetitive):

- **Optional**: $\mu EET_b(s,s') \equiv EET_i(s,s') \vee s = s'$
- **Repetitive**: $\mu EET(s,s') \equiv \exists t.EET_i(s,t) \wedge (s' = t \vee EET(t,s'))$
- **Optional repetitive**: $\mu EET(s,s') \equiv (s = s' \vee \exists t.EET_i(s,t) \wedge EET(t,s'))$

Here, $EET_i(s,s')$ denotes the relation for the indicated sub-EET. Similarly, boxes are formalized by a relation comprising the "boxed" EETs by simply forming a disjunction over them, thus allowing any of those EETs to substitute the box:

$$\mu EET_o(s,s') \equiv EET_1(s,s') \vee \ldots \vee EET_n(s,s'),$$

where $EET_1(s,s'),\ldots,EET_n(s,s')$ denote the relations for the boxed EETs.

Since the formalisation of an EET represents a property about the transition relation of the whole system, it can be used to express different positive and negative requirements about the system using an embedded clause as described in [2]. In case such a requirement does not hold for the system, the counter example generated of the model checker can be used to generate a counter example as discussed in [10].

2.3 Lessons Learned

AUTOFOCUS was originally implemented with a user controlled consistency mechanism (cf. [10]). Only syntactic consistency criteria of the description techniques where controlled automatically during the development process, all other conceptual consistency condition had to be initiated by the user. Experiences have shown that developers using AUTOFOCUS were willing to trade in a maximally flexible development process for an enforcement of conceptual consistency if offered a comfortable interface for the development of consistent specifications (see Section 5.5). Thus, the weak integration of description techniques in AUTOFOCUS is currently strengthened using a single conceptual model. This conceptual model and the resulting approach currently under implementation are described in Sections 3 and 5.

3 Conceptual Models

This section introduces a simplified version of a conceptual model for AUTOFOCUS. Here, specifications are *instances* of this conceptual model and describe systems in an integrated fashion. Developers create and manipulate them using concrete notations representing views upon them. Even different graphical or textual views on the same

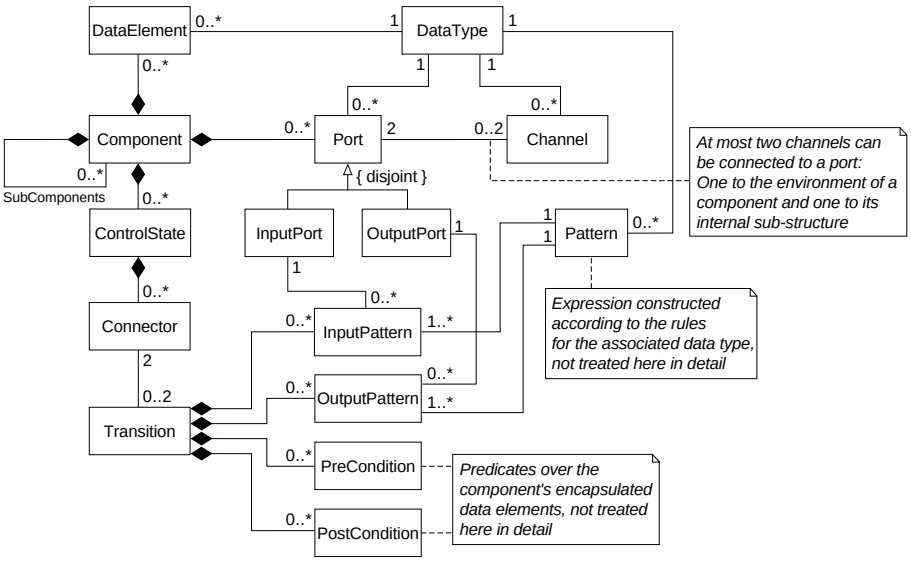

Fig. 3. Simplified Conceptual Model of AUTOFOCUS

parts of the model may be offered. The notations representing these views are the same used in the document-based initial version of AUTOFOCUS and were introduced in sections 1.1.4 through 1.1.8.

The elements in the conceptual model make up the essence of the notations used, like abstract syntax trees generated by parsers for programming languages. In programming languages, however, source code "documents" are the important modeling concept, the syntax tree is generated by the parser unnoticed by the user. Many software engineering tools offer a similar approach, treating system descriptions as—at most loosely related—documents of different kinds. In the model-based approach, 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 AUTOFOCUS are shown in Fig. 3 using a UML-style notation. For a more detailed description of the modeling concepts we refer to [11].

Subsequently, we describe the elements in the conceptual model, which are, of course, the concepts described by the notations introduced in Section 1.

- **Components** encapsulate *data*, *structure*, and *behavior*, communicating with their environment.
- **Data types** define data structures used by components.
- **Data** are encapsulated by a component and provide a means to store persistent state information inside a component, 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.
- **Channels** are directed, named, and typed. They connect component ports. They define the communication structure (topology) of a distributed system.

- **Control States** and **Transitions** between their entry and exit points—called **Connectors**—define the control flow of a component. Transitions carry four kinds of annotations determining their firing conditions,
  - pre-conditions and post-conditions, which are predicates over the data elements of the component to be fulfilled before and after the transition, respectively, and
  - input and output patterns, determining which values must be available on the component's input ports to fire the transition and which values are then written to the output ports.

The elements of the conceptual model can be regarded as abstractions of both the underlying formal model and their concrete notations. Thus, the conceptual model represents the common denominator of both the description techniques and a formal model.

Viewing specifications as graphs of specification elements, it is possible to construct a multitude of graphs using only instances of 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.[3] 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 Views and Description Techniques

How do developers develop system specifications using a conceptual model? In the model-based approach, a system specification is an *instance* of the conceptual model, i.e., a graph consisting of individual nodes, which are atomic modeling entities, and of arcs capturing their relationships. Such a model instance must obey the well-formedness conditions defined by the conceptual model; arbitrary graphs are not allowed since they do not represent well-formed specifications. As stated in Section 3, developers do not manipulate these "specification graphs" as a whole, but by picking only specific parts of it, which are of interest during particular development activities. These parts, usually closely related with each other, make up *views* of the system. For instance, the structural view in AUTOFOCUS considers only elements from the conceptual model describing the interface of components and their interconnection. The view on the control flow focuses on the state space of components and the transitions possible within the state space.

To manipulate elements of these views we represent them visually. In AUTOFOCUS we use the notations introduced in Sections 1.1.4 through 1.1.8. Although the notations used to represent modeling entities are the same as in the document-based approach, their purpose in the model-based approach differs substantially from a methodological point of view. In document-based development documents are *closed* modeling artifacts with no explicit references to modeling elements defined in a different context, outside a specific document. Only implicit references are defined, like references by equality of names of port elements or of names of variables. Only the

[3] 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).

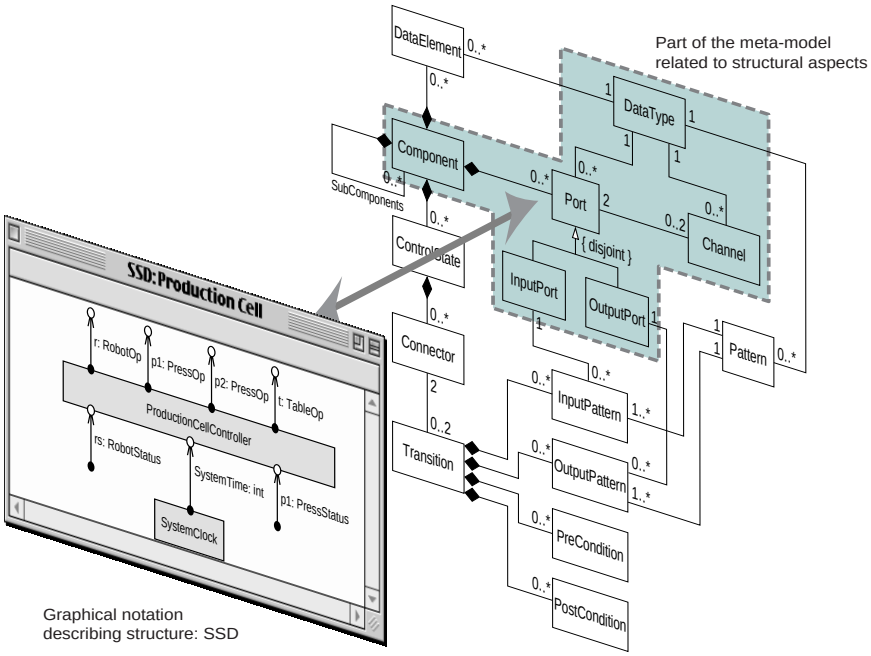

Fig. 4. Structural View of Conceptual Model and Notation Representing its Elements

assembly of all the individual documents and the resolution of these implicit references gives the complete view of the specification. In the model-based approach the specification of the system as a whole is incrementally constructed by adding new modeling elements to the specification. This complete specification thus really represents a *model* of the system, an abstraction of the complete system, which is the goal of the development process. The notations do not represent self-contained documents but a visualization of a clipping from the complete specification graph. This clipping *does* contain explicit references to other parts of the specification, as in Fig. 4, where the gray area in the conceptual model encompasses all elements carrying information about the structure of a system. Within this structural description, other information, such as the interface view of components (collection of ports of a component) and the data type view (necessary to describe both the component ports and the channels) is contained as well. Thus, ports and data types are *explicitly referenced* in the structural view. One possible graphical representation of the structural view of a system is given by SSDs as symbolized in Fig. 4. This sample SSD represents a part of a fault-tolerant production cell controller, which will also be used as an example in Section 5.2.2. We call this collection of a view and a notation together with their interrelationships a *description technique*. More formally, a description technique consists of

- a subset of modeling elements from the conceptual model and the relationships between them, which, together, define a specific view on the system,
- a concrete syntax (graphical or textual notation) representing these elements, and
- rules to map the concrete syntax to the modeling elements and vice-versa.

A description technique thus serves as a kind of peephole through which developers can see (and change) parts of a specification.

5 Specification Modules

The conceptual model introduces the terms and relations needed to describe the system specified by the developer. We show how those terms and relations are combined to form specifications. In Subsection 5.1 we define when a description of a system 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*

- is a *well-formed* description of one or more aspects of a system,
- may fulfill additional conceptual consistency conditions,
- does not necessarily need to be *complete.*

Like the conceptual model, a specification is an abstract concept and can have several concrete syntactical representations. The choice of the conceptual model determines the notion of a specification by defining the *well-formedness* and additional *conceptual consistency conditions* of a specification. The first describes invariant conceptual consistency 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 a port or a channel cannot be created without 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 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 reasonable specifications but may be violated throughout the design

process. At certain steps in the design process, consistency of the specification is required. Typical steps are code generation, verification, and specification module definition. Different consistency conditions may be required for different steps. While code generation or simulation require completely defined data types, this is not necessary for verification or specification module definition. Example conditions 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 formally; 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 called 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. Actually, completeness can be defined as a consistency condition and checked the same way (see [10]). 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 are:

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

In our approach a specification module is *not* distinguished from a specification. Thus, every well-formed part of a specification is considered a specification module. A well-formed part of a specification need not be complete. However, for a reasonable reuse of a specification module, it has to obey several consistency conditions. This leads to a simple distinction of two different specification module concepts*:*

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

Specification modules are well-formed specification parts possibly obeying additional consistency conditions. Therefore, specification modules can either be developed as

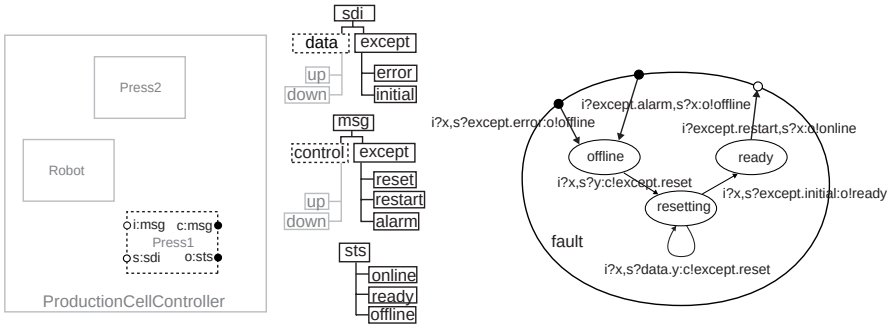

Fig. 5. Parameterized Error Treatment Module

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 is the one containing all relevant information. Since—unlike in the parameterized case—no instantiating of the module is needed, all information of the module can simple be added to the target specification. A simple example of a complete specification module is the press controller module of the production cell consisting of

- the data types describing the actuatory and sensory data,
- the interface description of the controller (typed input and output ports),
- the (empty) list of variables of the controller unit, and
- the behavioral description of the controller unit given by an STD 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 Fig. 5. The module is used to handle fault situations of the production cell units. Upon entering the module, an offline status message is issued. The unit is then brought to a defined state and stability of the unit is reported. Upon receipt of a restart message, the unit is restarted. Since all of the units of the production cell must support this kind of error treatment, it is useful to define a general fault module to be instantiated for all components. The defined specification module consists of three parts:

- The minimal interface to be supplied by a component intended to offer this fault recovery strategy. The interface is defined by the corresponding ports (i, o, c, s).
- The type of messages used to indicate the status of the unit or to influence its behavior. As the behavior is independent of the kind of unit (press, robot arm, etc.), the unit-dependent part of the message types (data, control) is not specified by the module but defined as a parameter to be instantiated upon use of the module.

- The behavior relevant for fault recovery. Only a part of a complete behavioral description of a unit is described by giving the necessary states (offline, resetting, ready), and the corresponding transitions and entry- and exit-points. Applying the module adds the fault-recovery routine to the target specification to be extended to a complete specification for the corresponding unit.

To apply a parameterized module to a target specification the parameters are 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 modules and their combination using the notion of colored graphs and typed binary relations.[4] 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.[5] Since the conceptual model is typed (the elements of the conceptual model are elements of distinct classes like components, ports, channels, etc.), the node set is partitioned. Thus a specification 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 $E_i \subseteq \mathrm{E}_i$ 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 definitions of $\mathcal{E}_i$ and $\mathcal{R}_j$ depend 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 type as well as $\mathcal{CT}$ between an channel and its type.

Generally R_i will not cover the complete range of sub relations of $\mathcal{R}_i$. As, for example,

- a system component cannot be its own sub component, the sub component relation is not reflexive

---

[4] See [15] for an elaborate treatment of using algebraic approaches to model specifications.

[5] For reasons of simplicity we will only consider binary relations; the extension to relations of higher cardinality is straight-forward.

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

- 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:

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

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 conditions described in Subsection 5.1. Some of those conditions above are typically described using arity-annotations of class 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 [10].

5.3.2 Operations on Specification Modules

To combine two modules (E, R) and (E', R'), the union $(E \cup E', R \cup R')$ is constructed with $E \cup E' = (E_1 \cup E_1', \ldots, E_m \cup E_m')$ and $R \cup R' = (R_1 \cup R_1', \ldots, R_n \cup R_n')$. *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 components, ports, types, states, etc.
- new relations between both old and new specification elements like adding a new port to an already existing system component.

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 considers 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, isomorphic mappings $M : \mathcal{E} \times \mathcal{R} \rightarrow \mathcal{E} \times \mathcal{R}$, $M_{\mathcal{E}_i} : \mathcal{E}_i \rightarrow \mathcal{E}_i$, and $M_{\mathcal{R}_j} : \mathcal{R}_j \rightarrow \mathcal{R}_j$ are defined with $M = (M_{\mathcal{E}_1} \times \ldots \times M_{\mathcal{E}_m}, M_{\mathcal{R}_1} \times \ldots \times M_{\mathcal{R}_n})$, 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. In Subsection 5.4.1 we outline 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. Having introduced a mathematical model for specification modules, it is obvious that this distinction is not a technical but a methodical one. To add a *complete* specification module we simply construct the union as defined above. Assuming disjoint sets of specification elements no further renaming is necessary.[6] 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 connect the ports of the module to the ports of the system.

To make use of a *parameterized* specification module it is necessary to instantiate its parameters before adding it to the specification. Therefore, the parameter elements must be renamed to elements of the target specification prior to the union construction. Like specification parameters in algebraic specification languages like SPECTRUM [9], the parameter elements are considered the interface of a module used to apply it to the target specification. Again, 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 algebraic specifications. 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 module) it 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 Fig. 5. 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 Fig. 5,
- 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.

Fig. 5 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] 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.

5.5 User Interface

Developers manipulate a specification as instance of the conceptual model using concrete notations, e.g., as in Sections 1.1.4 through 1.1.8, representing its elements and their structure. Users interacting with such notations need appropriate operations on the presented specification part. These operations must fulfill two criteria. First, they must not allow to create ill-formed specifications in terms of the conceptual model: Each operation must preserve conceptual consistency (cf. Sections 2 and 5.1). Second, they must be flexible and comfortable enough, so users do not regard them as too restrictive. Since notations are *visual* elements, interaction mechanisms developed in the GUI domain that have proven their user-friendliness are candidates to be considered here. For the AUTOFOCUS approach, the following two concepts are envisaged.

5.5.1 Drag and Drop

One paradigm to exchange information is "drag-and-drop", a mouse-based interaction scheme, which aims at several different purposes of information interchange. First, relocation of information can be accomplished by drag-and-drop: Grabbing an information element, a piece of text or a graphics element, for instance, and dragging it somewhere else results in removing (or copying) it from its source location and placing it in the target location.

Thus, drag-and-drop can be used to move representations of specification elements around within their graphical context. These are, however, usually operations that change only layout information and are thus not semantically relevant.[7] With respect to the specification, drag-and-drop can create new associations between elements of specifications. Thus, by grabbing a specification element, for example a type defined in an DTD, in a given context and dragging it into a different context, like a port, a relationship between the two contexts, the definition of a port of this type, is created. The target context uses the specification element defined in the source context.

5.5.2 Contextual Menus

Contextual menus provide users with a range of possible operations that are applicable in a certain context. In graphical file managers selecting file or directory icons and activating the contextual menu presents a set of possible operations that can be carried out upon the selected files/directories such as deleting or changing properties. We use contextual menus again to help users establish relationships between specification elements, for example type constructors to build messages for EET events, and, to preserve the well-formedness of the specification, to make only *suitable* specification constructs available to users.

5.5.3 Example

Consider the use of ports as example. When defining an input pattern for an STD-transition, an input port has to be specified along with the value or pattern to be present at the port for the transition to fire. Assigning the port can be done by dragging one of the component's input ports (and only input ports) from the SSD of the com-

[7] In EETs the vertical layout of the messages is relevant and relocating them has a meaning.

ponent into the property sheet of the transition. Alternatively, in the property sheet a contextual menu can be used to specify the port. In this menu, only the input ports of the component are available. The data required at the input port can be specified again by drag and drop of a data type constructor element from the data type definition used by the port. Alternatively, all possible constructors of component variables (or available local transition variables) could be given in a contextual menu.

6 Conclusion

We discussed the need for an integration of description formalisms in tool-supported formal system development. We showed that the introduction of a conceptual model and the interpretation of description techniques as views on the model on the one hand and the integration of powerful proof tools on the other hand support a manageable conceptually and semantically consistent development process. The introduction of a conceptual model additionally allows the introduction of specification modules and eases the reuse of specifications. However, as discussed in Section 5.5, sufficient usability is a prime requisite for the success of such an approach.

Thus, while the introduced approach is consequently carrying further approaches found in state-of-the-art system development tools, finally its acceptance can only be affirmed after the introduced concepts are implemented in the current prototype.

7 References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Bechtel, R.: Einbettung des µ-Kalkül Model-Checkers µ-cke in AutoFocus (in English). Master's Thesis. Institut für Informatik, TU München (1999)
3. Biere, A.: Effiziente Modellprüfung des µ-Kalküls mit binären Entscheidungsdiagrammen. Ph.D.Thesis. Universität Karlsruhe (1997)
4. Broy, M., Dendorfer, C., Dederichs, F., Fuchs, M., Gritzner, T., Weber, R.: The Design of Distributed Systems – An Introduction to Focus. Technical Report TUM-I9225, Technische Universität München (1992)
5. Diekert, V., Rozenberg, G.: The Book of Traces. Singapore World Scientific (1995)
6. Einert, G.: Ein Framework zur Konsistenzprüfung von Spezifikationen im AutoFocus-Werkzeug. Master's Thesis. Institut für Informatik, TU München (1998)
7. Formal Systems (Europe) Ltd.: Failures-Divergence Refinement: FDR2 User Manual. Oxford (1997)
8. Grosu, R., Klein, C., Rumpe, B., Broy, M.: State Transition Diagrams. Technical Report TUM-I9630. Technische Universität München (1996)
9. Grosu, R., Nazareth, D.: The Specification Language SPECTRUM – Core Language Report V1.0. Technical Report TUM-I9429. Technische Universität München (1994)
10. Huber, F., Schätz, B., Einert, G.: Consistent Graphical Specification of Distributed Systems. In: Fitzgerald, J., Jones, C. B., Lucas, P. (eds.): Proceedings of FME '97, 4th International Symposium of Formal Methods Europe, Lecture Notes in Computer Science, Vol. 1313. Springer (1997)

11. Huber, F., Schätz, B., Spies, K.: AUTOFOCUS – Ein Werkzeugkonzept zur Beschreibung verteilter Systeme. In Herzog, U., Hermanns, H. (eds.): Formale Beschreibungstechniken für verteilte Systeme. Universität Erlangen - Nürnberg (1996) 165–174
12. i-Logix Inc.: Rhapsody Reference (1997)
13. International Telecommunication Union: ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU, Geneva (1996)
14. Jones, M. P.: An Introduction to Gofer. User's Manual (1993)
15. Paech, B.: Algebraic View Specification. In Wirsing, M., Nivat, M. (eds.): Proceedings of AMAST '96: Algebraic Methodology and Software Technology. Lecture Notes in Computer Science, Vol. 1101. Springer (1996) 444–457
16. Schätz, B., Hußmann, H., Broy, M.: Graphical Development of Consistent System Specifications. In Gaudel, M.-C., Woodcock, J. (eds.): FME '96: Industrial Benefit and Advances in Formal Methods. Lecture Notes in Computer Science, Vol. 1051. Springer (1996) 248–267
17. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. Wiley Professional Computing (1994)
18. Telelogic AB: SDT 3.1 Reference Manual. Telelogic AB (1996)
19. Verilog: ObjectGEODE Method Guidelines. Verilog (1997)