# Consistent Graphical Specification of Distributed Systems*

Franz Huber, Bernhard Schätz, Geralf Einert

Fakultät für Informatik,
Technische Universität München,
Arcisstraße 21,
80333 München
Email: huberf@informatik.tu-muenchen.de,
schaetz@informatik.tu-muenchen.de,
einert@informatik.tu-muenchen.de

**Abstract**: The widely accepted possible benefits of formal methods on the one hand and their minor use compared to informal or graphical description techniques on the other hand have repeatedly lead to the claim that formal methods should be put to a more indirect or transparent use. We show how such an indirect approach can be incorporated in a CASE tool prototype by basing it upon formally defined hierarchical description techniques. We demonstrate the immediate benefits by introducing consistency notions gained from the formalization. Additionally, we show how the formalization can be used to apply automated property validation. Finally, we discuss some further techniques that could be based on the underlying formalization.

## 1 Distributed Systems Development

Development of distributed systems has become a main objective of software engineering today. Intelligent networks providing multimedia-related services, distributed information systems like for instance car rental booking systems, or embedded controller systems used, for example, in avionics systems or industrial production lines: all of these are examples of distributed systems.

The increasing complexity of those applications, in particular the complex interactions between the components of such systems, make their development complicated and error prone. Intuitive graphical formalisms to specify and develop such systems, provided by a number of development tools, are already in widespread use in industry. Many of these notations and tools, however, lack a precise interpretation in the sense of a formal semantics. Thus, in a number of cases the interpretation of certain properties of a modeled system is unclear or even ambiguous. Such ambiguously specified system properties easily lead to insecure systems where, for example, behavior under certain conditions may be unpredictable or dependent on factors not accounted for by the developers. Situations like these are intolerable especially in the development of safety-critical systems like those, upon which human lives are dependent, or mission-critical applications for companies. In such applications, the correctness of a system in relation to the system specification is crucial. Nonetheless,

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

many systems that are in use today bear inconsistencies that are undetected until particular situations arise, where the effects often are disastrous [9].

### 1.1 Applicable Formal Methods

Despite the benefits that can be gained by using formal techniques, their use in large-scale industrial systems development is still quite uncommon. As has repeatedly been stated, the reasons for this fact are mainly due to the complexity and clumsiness of the mathematical formalisms used there. Development strictly adhering to such formalisms is too time-consuming for industrial systems development and requires too much time for developers to get acquainted with.

As a consequence, it has often been proposed to combine the advantages of both approaches, intuitive graphical notations on the one hand, and mathematically precise formalisms on the other hand, in development tools encapsulating as much as possible of the mathematical formalisms under the hood of pragmatical graphical notations. AUTOFOCUS, the tool presented in this paper, tries to reach this goal in selected areas: AUTOFOCUS uses a formal method for the development of distributed systems called FOCUS [2] as the underlying basis. FOCUS describes distributed systems as collections of components that are spatially or conceptually/logically distributed. The components communicate, that is, exchange messages, over communication channels. Components may be hierarchically refined by networks of subcomponents in the development process. The behavior of these components is specified using mathematical concepts like stream processing functions or traces. AUTOFOCUS uses a set of practice-oriented, mainly graphical notations, embedded into FOCUS, to specify different aspects of these concepts.

The following section briefly introduces the AUTOFOCUS tool giving a short description of general aspects. Afterwards, we outline the AUTOFOCUS description techniques, characterizing some aspects of their relationships to the underlying semantical concepts of FOCUS. Subsequently, the notion of syntactical consistency of development documents is introduced. In this section, a classification of consistency conditions according to their use and implementation is given, followed by a description of how consistency is controlled during the process of systems development and how inconsistencies, which are inevitable under certain circumstances, are treated. After that we show how the mathematical basis of AUTOFOCUS, provided by the FOCUS method, is used to verify system properties exceeding syntactical consistency. The paper is concluded by an outlook to the future development of the AUTOFOCUS tool.

## 2 The AUTOFOCUS Tool

This section briefly describes important aspects of the current implementation of the AUTOFOCUS tool. For a further description of its architecture and implementation, we refer the reader to [6].

### 2.1 Distributed Client/Server Architecture

Because of their complexity distributed systems are generally developed in teams by several developers at the same time, often using different computer platforms. Therefore, AUTOFOCUS is implemented as a client/server system with a central repository where all development documents are stored. An arbitrary number of clients can ac-

cess these documents over a network connection. Thus system developers can use the specification documents simultaneously. By implementing the clients in Java, AUTOFOCUS can be used on most of the usual operating system platforms.

### 2.2 Version Management

Specifications are repeatedly revised, especially in the early phases of development. Therefore, the possibility to use version control of single documents as well as of whole projects is absolutely necessary. Tool support should allow to rule out inconsistencies (other than the ones mentioned above) caused by team members working on the same specification documents. Therefore, the AUTOFOCUS repository offers version control of both documents and projects as well as locking mechanisms for documents based on the usual pattern of one write access and multiple read accesses.

### 2.3 Graphical User Interface

On the client side, the complete functionality of AUTOFOCUS is accessible in a graphical user interface, parts of which are shown in figure 1. A project browser provides access to the development projects in the repository as well as to the associated document hierarchies grouped by document classes. For each of the graphical description techniques that will be introduced in the following section AUTOFOCUS provides a graphical editor. These editors, which support the hierarchy concepts of FOCUS, are shown in figure 1 as well.

All editors use an identical user interface concept with mouse-based user interactions to facilitate editing development documents. To use AUTOFOCUS diagrams in common word processors for documentation purposes the diagrams can be exported into encapsulated PostScript graphics files.

## 3 AUTOFOCUS - The Description Techniques

### 3.1 View-based Systems Development

AUTOFOCUS, like many tools and methods that are in practical use, does not aim at capturing a complete system description within a single formalism. Instead, different views of a system are each specified using an appropriate notation called *description technique*.

A distributed system can be characterized from several points of view, as

- the structure of a system including its components and the communication paths between them providing both a component interface specification and topological information,
- 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 communication paths, and
- the interaction of the components and the system environment via message exchange.

Only a description including all these views forms a complete picture of the system. Therefore, AUTOFOCUS offers five different description techniques: system structure diagrams (SSDs), state transition diagrams (STDs), data type definitions (DTDs) as well as component data declarations (CDDs), and extended event traces (EETs),

covering all the above aspects. Conforming with the hierarchical concepts of FOCUS, each of the graphical description techniques allows to model on different levels of granularity, supporting a top-down approach where, for example, components or behavioral modules can be either atomic or consist of sub-components or sub-modules themselves.

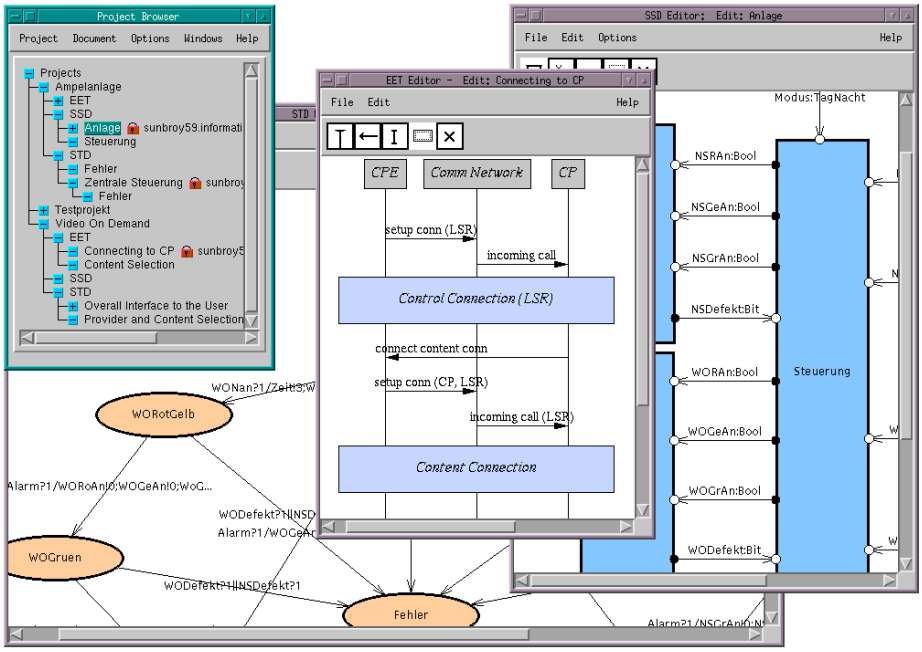


Figure 1: The AUTOFOCUS Client Application – Project Browser and Editors

### 3.2 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 (also called diagrams). Combined, these documents provide a complete characterization of a system in its current development status. Access and version control is done on the document level of granularity in the repository, which keeps track of the complete version history of every document. For a changed document, the user may choose whether it should be stored by default, simply incrementing the version number, or if it should be saved under a version number explicitly given by the user. In order to reuse documents, a document may be referenced by more than one project. Projects are subject to version control as well: in our approach, versions of a project are collections of specific versions of development documents.

**Hierarchical Documents**

As already mentioned, all graphical AUTOFOCUS description techniques share the concept of hierarchy. Both system structure diagrams and state transition diagrams -

which are essentially graphs - as well as extended event traces allow hierarchical refinement. In a system structure diagram, a system component may be viewed as a conceptual unit made up of a network of sub-components specified in another structure diagram document. In the same way, a state in a state transition diagram can be characterized by another state transition diagram document refining this state on a more detailed level. In extended event trace diagrams, so-called "boxes" are introduced as an abbreviating notation for parts of system runs specified in different event trace diagrams.

**Integrated Documents**

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

### 3.3 System Structure Diagrams (SSDs)

System structure diagrams describe static aspects of a distributed system, viewing it as a network of interconnected components that exchange messages over directed channels. Each component has a unique identifier and a set of input and output ports to which the channels are attached. Channels are defined by identifiers and data types describing the sets of messages that may be sent across them. Thus system structure diagrams provide both the topological view of a distributed system and the signature (the syntactic interface, given by the set of ports) of each individual component. As remarked above, components may be hierarchically refined by networks of subcomponents. In that case, the document specifying such a sub-network has the same set of communication ports as the higher-level component that this refined view belongs to. Graphically, as shown in figure 1, system structure diagrams are represented as graphs, where rectangular vertices symbolize components and arrow-shaped edges stand for channels. Both of them are annotated with their identifiers and, in the case of channels, also with their data types. Component ports are visualized as small hollow or filled circles, depending on whether they are input or output ports.

This graphical notation similar to data flow networks is a direct representation of the semantical concepts of a distributed system used in FOCUS. A textual representation to define such component networks, omitting the graphical context of SSDs, is given by the *Agent Network Description Language* defined in [14]. ANDL can easily be transformed into notations suitable for verification of system properties like, for example, the HOLCF package for the *Isabelle* theorem prover (see also section 5.3).

### 3.4 State Transition Diagrams (STDs)

State transition diagrams are extended finite automata similar to the concepts introduced in [4]. They are used to describe the dynamic aspects, that is, the behavior, of a distributed system or of its components. Each system component can be associated with an STD consisting of states and transitions between them. Each transition has a set of annotations: a pre- and a 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 that are read from or written to the input and

output ports of the component. The notation used to specify the input and output patterns is similar to CSP. For hierarchical refinement of states in STDs, we use a concept similar to the one used in the SSDs. Graphically, automata are represented as graphs with labeled ovals as states and arrows as transitions. Figure 1 shows an example of an AUTOFOCUS state transition diagram.

Semantically, STDs are flat, thus hierarchy in this case is only a mechanism of visual representation. They represent system or component behaviors given by stream processing functions or traces. A mapping to stream processing functions can be accomplished based on the concepts from [3].

### 3.5 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 [8] for this purpose. The data types defined here may be referenced from within other development documents, for example, as channel data types in SSDs.

In order to use such data type definitions for verification of system properties they can be transformed into HOLCF using the HOLCF domain construct introduced in [15]. Subsets of Gofer data type definitions can be efficiently transformed into model checker input format.

### 3.6 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.

### 3.7 Extended Event Traces (EETs)

Extended event traces are used to describe exemplary system runs from a component-based view. As shown in figure 1, we use a notation similar to the ITU-standardized message sequence charts (MSCs) with some core concepts taken from MSC'96 (ITU Z.120, [7]). As well as the other graphical AUTOFOCUS notations, EETs support hierarchy. Using so-called boxes a number of sub-EETs can be nested to specify variants of behavior in parts of an EET. Additionally, indicators can be used to define optional or repeatable parts of an EET. A complete description of EETs can be found in [13].

EETs carry somewhat redundant information in relationship to the other AUTOFOCUS notations. From the methodological point of view, they are intended to be used in the very early stages of systems development to specify elementary functionality of a system on an exemplary basis. Additionally, EETs can be used to specify system behavior in error situations as well. 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.

Further applications of EETs consist in the visualization of model checking and simulation results. For instance, witnesses obtained by model checking state transition diagrams can be visualized using EETs.

## 4 Consistency of Descriptions

If a specification exceeds the toy world size, the contained information in general is spread across several documents, like in different modules or libraries of a large programming package. A large number of errors arises out of the fact that those information pieces are created separately and thus do not automatically fit together. Here, simple checks based on the abstract syntax of the description techniques can already be an enormous help. Since AUTOFOCUS uses different classes of documents as well as hierarchically organized document structures, it becomes even more important to make sure that the information spread out over those documents is consistent.

Therefore, consistency checks are offered to ensure that the produced documents fit together. Consistency includes several different classes of syntactical correctness criteria like

- **Grammatical Correctness:** The corresponding document obeys the syntactical rules for textual documents or the graph-grammatical rules for graphical documents.
- **Document Interface Correctness:** If a document is embedded into another document according to the hierarchical concepts introduced before, those documents must have compatible interfaces to each other (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 (channel types in SSDs or STDs, for example).
- **Type Correctness:** The type of an object assigned and the type of the object it is assigned to must coincide (channels and ports in SSDs, or channel values and channel types in STDs, for example).
- **Completeness:** All necessary documents of a project have to be present.

Those syntactical conditions can be split in two classes according to their definition and treatment:

- **Intra-document Conditions:** Conditions that can be checked using only information found in the document itself.
- **Inter-document Conditions:** Conditions that can only be checked by using two or more documents at the same time.

In the following two sections we give some examples for each class using SSD documents.

### 4.1 Intra-document Consistency

Intra-document consistency basically corresponds to the syntactical and semantical analysis performed during the parsing of program code. In general, the grammatical correctness and the type correctness are checked here. We give some simple consistency conditions for an SSD:

- Each component has a non-empty name.

- Each port has a non-empty name, a non-empty type and a direction (input or output).
- Each channel has a non-empty name and a non-empty type.
- Each port is bound to a channel.
- Each channel is bound to one port per direction with the same type.

Note that no notion of a document is used here. These conditions can easily be formalized using typed first order predicate logic with equality if we introduce appropriate individuals for the elementary objects like identifiers, components, channels, ports, or directions, and appropriate functions like *name_of* , *type_of*, or *direction_of*.

Thus, the last two conditions may be formalized as

$$\forall con : Connector.(\exists chan : Channel.channel\_of(con) = chan)$$

$$\forall chan : Channel.\exists icon, ocon : Connector.(inconnector\_of(chan) = icon \wedge outconnector\_of(chan) = ocon \wedge type\_of(icon) = type\_of(chan) \wedge type\_of(ocon) = type\_of(chan))$$

### 4.2 Inter-document Consistency

For inter-document consistency, we use checks that are performed in a similar way during the linking process of a program code. Primary checks for this case are the document interface correctness, the definedness and the completeness. Like in the case of the above intra-document conditions we define conditions for SSDs that need more than one document to be checked:

- Each sub-document has a defined corresponding component in a different super-document
- For each port of a sub-document bound to the environment there exists a single port bound to the corresponding component of the super-document having the same name, same type and opposite direction.
- If a component has a defined sub-document, then for each port of the component there exists a single port in this sub-document, having the same name, type and the opposite direction.

Again, these conditions can be formalized as above. Note that now, however, we have to add documents as individuals and corresponding functions to the language.

$$\forall comp: Component, doc: Document.(subdocument\_of(comp) = doc \Rightarrow (\forall father: Port.(component\_of(father) = comp \Rightarrow (\exists son: Port.(document\_of(son) = doc \wedge name\_of(father) = name\_of(son) \wedge type\_of(father) = type\_of(son) \wedge direction\_of(father) \neq direction\_of(son) \wedge \forall port: Port.((name\_of(port) = name\_of(son) \wedge document\_of(port) = doc) \Rightarrow port = son))))))$$

### 4.3 Consistency and Development

During the development process of a system, its specification documents are in fact inconsistent most of the time. In view-based systems development, these views, although describing different aspects of a system, are not independent of each other. For instance, a structural description in an SSD uses the data types defined in DTDs

to identify the data transmitted on channels. Transitions in STDs use port names from related SSDs to identify the data read (written) when the transition is performed.

Quite naturally, in a typical development scenario, these descriptions are incomplete for most of the time in a development process: SSDs are drawn before all necessary data type definitions are finished; behavior is specified for components for which the interface definition has not yet been completed; the revision of the refined structure (network of sub-components) of a system component requires a change in the interface of the component, thus rendering the component's interface temporarily inconsistent with that of its refined structure, and many more situations like these. Having many developers working on one system specification independently usually even worsens these circumstances producing more inconsistencies.

Consequently, in our view, inconsistency in the development is a natural and inevitable phenomenon that should be treated accordingly by the tool used.

**User Controlled Inconsistencies**

From the observations stated above, we conclude that a tool should give its users, that is, the developers working with it, control over when to check the system descriptions for consistency. The tool should not try to enforce consistency of descriptions automatically. Instead, developers should be able to decide when to have the tool perform appropriate checks for consistency. This approach, in our view, conforms much more to the way in which human developers work.

Many CASE tools that are commercially available are not being used just for this reason: developers feel too restricted when working with them, as they have to change their habits of working only to fulfill consistencies enforced by the tool. Similar experiences were made with syntax-driven editors for programming languages like Pascal, or C++. These editors, although intended to serve as an aid for programmers, have quickly proven to be more of an impediment than a help as they are usually too restrictive in their use.

**Definition of Consistency Conditions**

In AUTOFOCUS, the actual conditions upon which consistency checks are based, are defined using a declarative textual notation, similar to first order predicate logic with a simple type system. This approach, in contrast to hard-coding the consistency conditions in the AUTOFOCUS implementation language Java, which would be more efficient, provides a clear and simple way for developers to extend the set of consistency conditions if needed. In particular, no program code of the tool has to be modified or added. Thus it is unnecessary to know the internal program structure of AUTOFOCUS in order to write new consistency checks or to modify existing ones. The textual documents containing the consistency conditions stored in the repository of AUTOFOCUS also hold further information, like an informal explanation about the consistency condition, and whether it is an intra- or an inter-document condition.

The basic language elements of the textual notation used for the consistency conditions are

- quantors (universal quantor and existential quantor),
- selectors to access properties of documents, and
- operators on logical expressions and the equational operator.

A number of examples for the concrete syntactical representation of such consistency conditions in AUTOFOCUS is given subsequently:

*Example 1: Intra-document condition*

```
forall c: Component . Name(c) != ""
```

This very simple intra-document consistency condition states that no component in the SSD document being checked may have an empty name. It only uses selectors referring to items within that document and is thus classified as intra-document condition.

Note that the universe of components that the quantor is bound to in this case is implicitly restricted to those belonging to the SSD document being checked.

*Example 2: Inter-document condition*

```
forall ea: Axis .
   exists sc: Component .
      name(ea) == name(sc)
```

This condition for component axes in EET documents refers to elements from external documents, namely SSD documents, and is thus an inter-document condition. It checks whether for each component axis in a given EET, a corresponding component (in an SSD) is existing.

*Example 3: Inter-document condition*

```
forall c: Component .
   (exists ssd: SSDDocument . refinement(c, ssd)) or
   (exists std: STDDocument . behavior(c, std))
```

This consistency condition relating components with SSD documents and STD documents asserts that each component must either be refined by an SSD document specifying its sub-structure or be related to an STD that specifies its behavior. This consistency condition is an example for a simple completeness property of a system specification.

**Integration of Inconsistency Treatment**

In AUTOFOCUS, developers can invoke consistency checks from within different environments: editors and the hierarchical project browser.

In editors, where just a single document is being edited, all intra-document consistency checks for the respective kind of document can be started. This provides a quick and handy way for developers to ensure a minimum level of consistency within one single document. Checks of that kind are started simply by selecting a menu command in the editor of the document.

In the project browser, developers have several options to perform consistency checks. By selecting a whole project, a global consistency check for all development documents of that particular project, using all available consistency conditions, is started. This includes all intra-document checks, which are each performed for every appropriate document. Partial consistency can be checked by selecting a special class of documents, for example, by selecting the SSD category, which means that all SSD documents will be checked applying all appropriate intra-document checks and all inter-document checks suitable for SSD documents. Figure 2 shows this case, where the command to perform the consistency checks was invoked with the SSD document category selected. All consistency checks currently defined for this document category are then lined up in a dialog, grouped by the categories they belong to. These checklists are generated at runtime, based on the selection in the project browser and on the available consistency checks. Usually, all checks in these lists are

to be executed, but in case only a selection of them should be carried out, individual ones can be deselected, thus performing only an arbitrary subset of all possible checks.

The tool then performs the selected checks on the appropriate documents. As a result, it displays a list of all the consistency conditions that have been found violated by at least one of the documents checked. By selecting one of these conditions, users may bring up a list of all documents violating it, and, from there, directly navigate to the individual documents, that is, open the documents using the corresponding editors with the items violating consistency already highlighted. For consistency checks invoked from within editors, this is obviously not necessary, as the editor is already open; here the components violating consistency are directly highlighted.

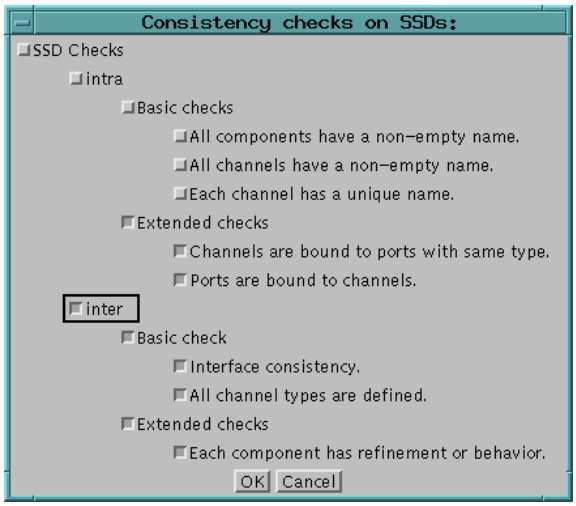


Figure 2: Sample List of Available Consistency Checks for SSDs

## 5 Verification of Specifications

After having introduced the description techniques and having defined several syntactical consistency conditions on them we now define their exact meaning. After giving an intuitive description we will introduce their semantics using μ calculus logic [10]. Since those descriptions can be automatically verified in a restricted range we will show how those checks can be performed using a μ calculus model checker like μ-cke [1].

The verification based on the relational μ calculus described in this section is ongoing work currently being implemented, the verification based on interactive theorem proving at the end of the section only exists in fragments. For easier understanding, we describe the formalization in a non-optimized version which makes the basic concepts more obvious.

### 5.1 Informal Semantics

While the meaning of DTDs, CDDs, and SSDs is quite obvious and needs no further explanation, the exact meaning of the behavioral description techniques STDs and EETs was left open so far. In the following two sections we will give an informal description of the meaning of those techniques. These informal descriptions will be formalized accordingly using μ calculus logic.

**Meaning of STDs**

An STD characterizes the behavior of a system or a system component reacting on input received from its environment and producing output sent to the environment. Those reactions depend on the actual state of the component and influence the future behavior of the component by setting a new state. Since an STD describes an extended finite state machine by using variables local to the characterized component, the state of a component is defined both by the control state (that is, the state of the finite state machine) and the values of the local variables of the component. A new state of the component as a result of the reaction to input is a new control state and new values of the local variables.

As mentioned in section 3.4, input and output patterns are used to describe the messages read from the input ports and written to the output ports. Each pattern consists of one or more port patterns. An input port pattern is built from an input port identifier and a message type construct of a type matching the port type. A message type construct can be built using constants and data type constructors from the DTD of the system, variables defined in the CDD of the component or free variables. An input port pattern matches an actual message at this port, if the constants and the values of the defined variables match the corresponding values of the read message. The free variables are bound to the actual values as found in the message.[1] Thus, while the CDD defined variables are only read in an input pattern, the free variables get set during the matching process. Output port patterns are constructed correspondingly, the message written to the port is generated by simply assigning the current value of both defined and free variables to their identifiers.

In pre-conditions only defined variables may be used to define predicates over the current data state. In post-conditions, defined and free variables can be used to the define the predicates. Like in output port patterns, defined and free variables are bound to their current values. Additionally, for each defined variable $x$ a primed variable $x'$ can be used to address the value of the variable *after* the execution of the transition.

It is important to note that in our approach the input is read simultaneously from all input ports. Input patterns used in an STD are complete in the sense that unspecified combinations of input patterns are interpreted to result in an empty valued output and leave both control state and local variables unchanged. If no input pattern is defined for a certain port, the input pattern will only match if no message is received on this port. Analogously, output messages are produced simultaneously for all output ports. If no output pattern is defined for an output port, the value of the message written to this port is empty. As a consequence, the meaning of STDs is closer to

---

1 More formally, a match is considered successful if a data element equal to the actual message can be constructed according to the pattern using its type constructors, the constants, the CDD defined variables with their current assignment and the free variabales with an arbitrary assignment.

hardware oriented description mechanisms like Statecharts than abstract description languages like SDL. Since, furthermore, no implicit message buffering is done by the components, buffering must be explicitly introduced into the specifications if desired.

**Meaning of EETs**

For sake of brevity we will only discuss simple EETs without indicators or boxes. Those extensions, however, can be easily integrated in the formalization given later. Informally, the interpretation of an EET is given by the following statements:

- The system behavior is interpreted as the concurrent runs of its components, each run described by the incoming and outgoing messages of the respective component shown at the corresponding axis. By identifying which incoming message of the receiving component corresponds to which outgoing message of the sending component those single descriptions are merged into a complete system description. Note that this is a more general version than used in [13] since only a partial ordering of events is assumed here.
- The transmission of a message between components is interpreted to happen instantaneously, no delay is introduced during transmission.
- Only the sending and receiving of a message can be observed as an event using EETs.
- Two consequent events observed at a component take place either at the same time or with an unknown finite delay in between. During this delay no other events can be observed at this component.

### 5.2 Exploration and Model Checking

The above described model of computation for STDs was chosen for two reasons:

- A lot of embedded systems as mentioned above are described on hardware oriented levels. In such a setting, synchronized and unbuffered communication is the natural paradigm. Therefore, this paradigm is generally adopted in tools used in this area [5].
- If unbuffered communication is used, the space needed to describe the current system space is fixed during execution.[2] For such systems, in general, model checking techniques offer a reasonable possibility to automatically verify behavioral properties.

In the remainder of this section we describe how the informal semantics given above can be formalized to be used with a relational $\mu$ calculus model checker.

**Transformation of STDs**

Basically, STDs are formalized by describing all possible sequences of states of the corresponding machine starting in a initial state of the STD. The sequences include the messages consumed and produced. To characterize those sequences, we simply describe the transition relation generating the sequences. To formalize STDs and combinations of STDs via SSDs we have to give a $\mu$ calculus representation of channels, ports, the control state space, the variable state space, and the transitions.

Channels are formalized as variables shared between the two adjacent components of these channels addressed via ports. A channel will therefore only hold one value of the corresponding channel type at a time. Thus, messages are not buffered. If

---

2 Given fixed size data types.

an incoming message is not explicitly stored in a local variable by the receiving process, it will be lost after transmission. Since the lifetime of a value is restricted to a single step of the state machine, we need a special value to describe the absence of a message on a channel. Therefore each data type contains the special value *nil* representing such an absence.

The variable state space of an STD, that is, the collection of the local variables of the corresponding component as defined in its CDD, is formalized as the product of all variables.

The control state space of an STD is formalized by introducing an additional state variable containing the actual control state of the corresponding machine.

Additionally, a predicate $Init(s, x)$ on the control and variable state space is defined, characterizing initial configurations of the system as given by the initial state of the STD and the initial variable assignment in the CDD.

Transitions may be of the form

$$S_1 \xrightarrow{P(x);I(x,v)/O(x,v);C(x,x',v)} S_2$$

where

- $x$ is the set of local variables defined for the component described by the STD; the definition of these variables can be found in the CDD of the component.
- $v = \{v_1, \ldots, v_k\}$ is a set of variables $v_1, \ldots, v_k$ local to the definition of the transition; these are variables used in the transition definition without being defined as variables local to the STD.
- $P(x)$ is a predicate over the set of variables of the state transition diagram *x*. and the set of variables of the transition definition $v$. At the moment, only propositional logic plus equality on the individuals and individual variables are allowed to form those predicates.
- $I(x,v) = I_1(x,v);\ldots;I_m(x,v)$ is a list of input port patterns over the set of variables *x* where the pattern for port $i_i$ $I_i(x,v)$ may either be empty or $i_i ? c_i(x,v)$ with $c_i(x,v)$ being a data type constructor using the variables from *x*. and *v*.
- $O(x,v) = O_1(x,v);\ldots;O_n(x,v)$ is a list of output port patterns corresponding to the input patterns which may either be empty or $o_i ? d_j(x,v)$ with $d_j(x,v)$ being a data type constructor using the variables $x$ and *v*.
- $C(x, x', v)$ is a predicate over the variables of the state transition diagram and the transition definition similar to the pre-condition; additionally to the variables *x*, corresponding primed variables *x´* may be used to describe the values of the variables before and after the transition is executed.

For each transition of the above form a clause

$$\exists v_1, \ldots, v_k . s = S_1 \wedge P(x) \wedge i_1 = c_1(x,v) \wedge \ldots \wedge i_m = c_m(x,v) \wedge$$
$$t = S_2 \wedge o_1 = d_1(x,v) \wedge \ldots \wedge o_n = d_n(x,v) \wedge C(x,x',v)$$

is introduced. If an empty port pattern was used for port $i_i$ or $o_j$, $i_i = nil$ and $o_j = nil$, respectively, are used in the above clause.

The complete transition relation is defined as the disjunction of all the clauses introduced for the defined transitions plus clauses for all unspecified behaviors. Note that in case of conflicting transitions, that is, transitions with the same start state, pre-

condition and input pattern but differing output pattern, post-condition or end state, these conflicts are interpreted - as usual - to be solved nondeterministically: All computations picking one of those conflicting transitions are included in the semantics.

**Combination of SSDs/STDs**

If a system is described as a collection of several communicating components, the behavior of the system must be generated from the description of the behavior of the single components and the static aspects of intercommunication via the channels. As in the case of EETs, we assume communication to take place instantaneously without introducing delay. Therefore, this formalization is basically done by simply combining the description of the components and identifying the shared communication channels. More formally,

- the new control state space is the product of all the control state spaces of the components,
- the new variable state space is the product of all the variable state spaces of the components and the internal channels as the defined by the SSD,
- the new input and output channels are the external channels as defined by the SSD, and
- the set of initial states is the product of all the sets of initial states of the components.

Furthermore, the transition relation $R$ for the hierarchical system is defined to be the product of the single transition relations of each component.

**SSD/STD-Refinement**

One possible application of the above formalization and the model checking mechanism is the verification of refinement relations of SSDs together with the corresponding STDs. This becomes necessary if a component of an SSD has an associated sub-document, and for both the component itself and the components of the sub-document STDs are given to describe their behavior. Here, we have to show that the behavior of the complete hierarchical system described by the corresponding STD of the component is a more abstract version of the complete behavior of its components. The behavior of the complete system can be described as the combination of the behaviors of the individual components, abstracting from internal channels.

In FOCUS, on the requirements level we use trace equivalence as behavioral equivalence notion, and inclusion on trace sets as behavioral refinement notion. Since these notions are used in the *Isabelle* HOLCF-implementation of Focus, a corresponding notion of equivalence and refinement has to be defined for STDs using the relational μ calculus. Otherwise, the semantics given by the theorem prover *Isabelle* and by the model checker μ-cke would differ, yielding two different semantics for the same description techniques.

In general, bisimulation is the equivalence notion used on state-based description techniques and, in particular, in many model-checking based approaches. It cannot, however, be used in our case, since bisimulation and trace equivalence do not coincide here. Since we use STDs without a notion of fairness like, for example, fairness sets used in Buechi automata, we can apply the standard approach to show language inclusion of two given automata without any major complexity difference. Thus, to show that the set of traces $L(S_1)$ of STD $S_1$ is a subset of the corresponding set $L(S_2)$ of $S_2$, that is,

$$L(S_1) \subseteq L(S_2)$$

we show that

$$L(S_1) \cap \bar{L}(S_2) = \varnothing$$

where $\bar{L}(S_2)$denotes the complement of $S_2$ regarding the set of all traces. The complement automaton $\overline{S_2}$ of $S_2$ can be effectively constructed by adding a new state *f* to the set of states of $S_2$ and defining the transition relation $\overline{R}$ to be

$$\overline{R} = R \cup \{(s,i,o,f) \mid \forall t.\,(s,i,o,t) \notin R\} \cup \{(f,i,o,f) \mid i \in I \wedge o \in O\}$$

Additionally, we define a relation $Final_i$ characterising final states in both automata. As mentioned above, we do not use fairness conditions in our approach. Thus it suffices to compare finite prefixes of execution traces of $S_1$ and $\overline{S_2}$ instead of their infinite traces, and to show that no common finite prefixes of infinite execution traces of $S_1$ and $S_2$ exist. Therefore, every state of $S_1$ is a final state. For $\overline{S_2}$, only $f$ is a final state since only traces not possible in $S_2$ are considered.

Finally, the emptiness of the intersection of the trace sets is simply checked by making sure that no computation of the product automaton of $S_1$ and $S_2$ will lead to a final state of the automaton, or - conversely - that by computing backwards no initial state can be reached from a final state. Therefore we define a relation *LeadsToFinal* to characterise states leading to a final state of the product automaton of $S_1$ and $\overline{S_2}$:

$$\begin{aligned}
&\mu\ LeadsToFinal(s_1,s_2) = \\
&\quad (Final_1(s_1) \wedge Final_2(s_2)) \vee \\
&\quad \exists i,o,t_1,t_2.\ R_1(s_1,i,o,t_1) \wedge \overline{R_2}(s_2,i,o,t_2) \wedge LeadsToFinal(t_1,t_2)
\end{aligned}$$

Here, μ characterises the operator for the smallest fixed point. Now, the check itself simply consists of

$$\forall s_1,s_2.\,Init_1(s_1) \wedge Init_2(s_2) \rightarrow \neg LeadsToFinal(s_1,s_2)$$

If this condition does not hold, that is, a trace of $S_1$is found not contained in $L(S_2)$, a counter example can be produced to demonstrate the mismatch.

**Transformation of EETs**

While STDs were basically formalized by giving a transition relation, we will formalize EETs as conditions on these transition relations. In order to demonstrate the core concept of this formalization we will first define the formalization only for EETs consisting of only one axis. After that, we will sketch how such single axis EETs can be combined to form general EETs.

Using the relational μ calculus, for each step within the corresponding EET we can define a relational μ calculus formula characterizing the described behavior. In case of the EET shown in Figure 3 we have to define the steps 1 through n+1 according to the following scheme:

$$\begin{aligned}
&\nu EET_{n+1}(s) = \exists i,o.(C(i,o) \wedge \exists t.(R(s,i,o,t) \wedge EET_{n+1}(t)) \\
&\mu EET_n(s) = \exists i,o.(P_n(i,o) \wedge \exists t.(R(s,i,o,t) \wedge EET_{n+1}(t))) \vee \\
&\quad (C(i,o) \wedge \exists t.(R(s,i,o,t) \wedge EET_n(t))
\end{aligned}$$

$$\vdots$$

$$\mu EET_i(s) = \exists i,o.(P_i(i,o) \wedge \exists t.(R(s,i,o,t) \wedge EET_{i+1}(t)))$$
$$\vee (C(i,o) \wedge \exists t.(R(s,i,o,t) \wedge EET_i(t))$$
$$\vdots$$
$$EET(s) = Init(s) \wedge EET_1(s)$$

Here, ν and μ characterize the greatest and least fixed points of the corresponding equation. The variables *s* and *t* contain the complete state space of the system, that is, both the variable and control state space. *C* describes a step of the component where no messages are sent or received between two observed events:

$$i_1?nil \wedge \ldots \wedge i_m?nil \wedge o_1!nil \wedge \ldots o_n!nil$$

Finally, $P_i$ describes the fact that the corresponding messages were sent or received while no events occurred at the other channels of the component:

$$i_1?v_1 \wedge \ldots \wedge i_m?v_m \wedge o_1!w_1 \wedge \ldots o_n!w_n$$

with $v_i$ and $w_j$ being the values received or sent, or *nil* otherwise.

By using a similar construction as in the case of the hierarchical SSDs described for the combination of SSDs and STDs, the combination of single component EETs to complete system EETs can be defined. The system state space is the product of the component state spaces, and the in-between states (defined by the $EET_i$ predicates) are the products of those states.

**STDs and EETs**

Note that EETs are defined as requirements for the transition relation of a system. This is due to the fact that the μ calculus definition of *EET* as given above makes use of the transition relation *R* as defined by the corresponding STD. They are not descriptions of a system in themselves. So we need not use some kind of language inclusion relation to verify the relationship between STDs and EETs as we have done in case of SSD/STD refinement.

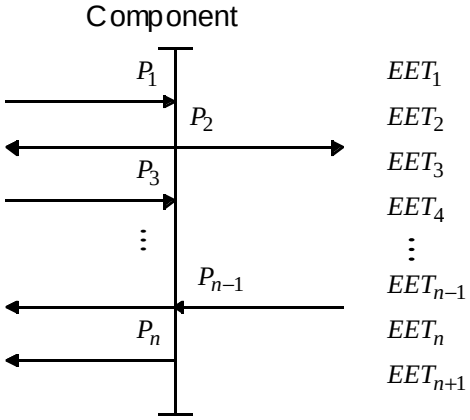


Figure 3: One Component EET

Instead, to check that an EET formalized as $EET(s)$ using a transition relation $R$ describes a possible run of the system with transition relation $R$ we just have to prove

$$\exists s.EET(s)$$

Thus, EETs can be seen as "use cases" describing a possible expected behavior of the transition relation of the system. Similarly, to prove that an EET describes a run not possible for the system, we have to prove

$$\forall s.\neg EET(s)$$

This, in turn, applies EETs as "negative use cases" describing behavior patterns not allowed for the transition relation of the system.

**Further Properties**

Besides from checking the behavioral consistency of hierarchical systems or STDs and EETs, the full expressiveness of the relational μ calculus can be used to verify additional properties. By formalizing properties like liveness of a component, deadlock freedom or similar properties as a relational μ calculus expression without using the graphical notations these properties may be verified automatically using the translation of the graphical specification. Since formalizing these properties using the relational μ calculus however requires quite some mathematical knowledge, easier specification front-ends to the μ calculus like CTL might be more appropriate in this framework.

**Witnesses**

A major advantage of model checking is the possibility of witness generation. If we try to prove the incorrect assumption that a component can be implemented by a subsystem of components, we get a counterexample describing a system run leading to a contradiction instead. This counterexample can be visualized using the EET description technique, thus showing why the refinement relation does not hold. Similarly, if we check whether an EET describes a legal system run, we can mark the point of failure where an event prescribed by the EET cannot be produced by the system.

### 5.3 Theorem Proving

In general, model checking of the described systems may not be feasible under all circumstances due to the state space being too large, even using sophisticated mechanisms to reduce the size of the internal representation like variable interleaving. In this case it may be necessary to apply interactive theorem proving to verify certain system properties. Therefore, in addition to the relational μ semantics a semantics suitable for interactive theorem proving must be given. We use the semantical model of stream processing functions defined in the FOCUS methodology [2]. By defining a corresponding semantics [12] on basis of the HOLCF [11] logics, we can use the *Isabelle* theorem prover as interactive verification tool. Of course, this semantics has to be consistent with the semantics introduced using the relational μ calculus. Basically, the semantics described in [3] can be used.

## 6 Future Work

AUTOFOCUS in its current state is the result of a number of student projects. It was started in a practical project course in software engineering in the 1996 summer term and has since then been enhanced to its current state.

Based on this status, a number of further extensions are currently planned or implemented. The present implementation of AUTOFOCUS is intended to be the core of

a complete tool-set for developing distributed systems. Extensions currently in work or under investigation are listed subsequently.

### 6.1 Code Generation

Code Generation is an important core functionality for various other functions, like prototyping, simulation, and more. Since transformation done by hand is not only laborious but also error-prone, it is an important feature in a formally-based approach. As we use an implementation-oriented description technique to describe system behavior, code generation becomes a simple process. In addition to the specialized generation mechanisms used in the simulation framework as mentioned in section 6.2, it is planned to implement a generic code generator that can be customized according to specific needs.

### 6.2 Simulation

Simulation, particularly in combination with sophisticated visualization tools, is a very important means for developers to gain a deeper understanding of how a developed system works. The area of simulation covers a wide range of applications, reaching from elementary animation of single diagrams, like STDs, visualizing the state transitions according to the inputs received, to concurrent simulation of several or all system components processing input and producing output simultaneously. In this context, the ability to generate protocols of simulation runs is desirable: we are planning to use EETs, showing the recorded communication history of selected components for this purpose.

### 6.3 Graphical Development Steps

Graphical development steps for the same description techniques as used here, guaranteeing consistent refining transformations of the specifications were already discussed in [13], using a slightly different semantics. Since the semantics introduced here is basically compatible with the original version, the same graphical transformation steps can safely be applied. Thus, enhancing the AUTOFOCUS tool with these mechanisms is a straight-forward step.

### 6.4 Reuse and Libraries

We intend to integrate library mechanisms into AUTOFOCUS, enabling developers to easily reuse documents that have been developed earlier, a functionality that is essential for industrial systems development. Thus, developers can develop libraries of reusable system components, structures, and behaviors that can be incorporated into new development projects.

## 7 Bibliography


[1] A. Biere. *Eine Methode zur μ-Kalkül-Modellprüfung*. Slides for the AKFM from 23.05.96, GI/ITG-Fachgespräch „Formale Beschreibungstechniken für verteilte Systeme“ (in German), 1996.

[2] 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.

[3] M. Fuchs and M. Mendler. *Functional Semantics for Delta-Delay VHDL based on Focus*. In: C. Delgado Kloos and P. Breuer (eds.). Formal Semantics for VHDL, Kluwer Academic Publishers, 1994, Chapter I, pp. 9 - 38.
[4] R. Grosu, C. Klein, B. Rumpe, and M. Broy. *State Transition Diagrams*. Technical Report TUM-I9630, Technische Universität München, 1996.
[5] D. Harel and A. Naamad. *The Statemate Semantics of Statecharts*. IEEE Transactions of Software Engineering Methods, 1996.
[6] F. Huber, B. Schätz, A. Schmidt, and K. Spies. *AutoFocus - A Tool for Distributed System Specification.* In: B. Jonsson and J. Parrow (eds.). Proceedings FTRTFT´96. Lecture Notes in Computer Science 1135, Springer, 1996, pp. 476-470.
[7] International Telecommunication Union. *Message Sequence Charts, 1996.* ITU-T Recommendation Z.129. Geneva, 1996.
[8] M. P. Jones. *Introduction to Gofer 2.20.* Technical Report, Yale University, 1991.
[9] J.-L. Lions et al. *Ariane 5 Flight 501 Failure.* ESA Press Release 33-96, Paris, 1996.
[10] D. Park. *Finitness is μ-ineffible*. Theoretical Computer Science 3(2), 1976, pp. 173-181.
[11] F. Regensburger. *HOLCF: Higher Order Logic of Computable Functions*. In: T. Schubert, P. Windley, and J. Alves-Foss (eds.). Higher Order Logic Theorem Proving and Its Application (HOL95), 1995, pp. 293-307.
[12] R. Sandner and Olaf Müller. *Theorem Prover Support for the Refinement of Stream Processing Functions*. Proc. 3rd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS´97). Lecture Notes in Computer Sience Vol. 1217. Springer, 1997, pp. 351-365.
[13] B. Schätz, H. Hußmann, and M. Broy. *Graphical Development of Consistent System Specifications*. In: J. Woodcock, M.-C. Gaudel (eds.). FME' 96. Lecture Notes in Computer Science Vol. 1051, Springer, 1996, pp. 248-267.
[14] B. Schätz and K. Spies. *Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik*. Technial Report TUM-I9529, Technische Universität München, 1995.
[15] D. von Oheimb. *Datentypspezifikationen in HOLCF*. Master´s Thesis, Technische Universität München, 1996.