Abstract—In this paper a method is presented to systematically integrate fault tolerance properties into the design of complex software systems. This is achieved by exploiting a formal specification of the system where the amount of necessary redundancy can be determined. The system description is based thereby on a combination of a predicate/transition net with regular expressions. The net model provides a formal—but nevertheless lucid—overview of the system behavior in general, supporting the correct understanding of potential concurrency in the system processes. Regular expressions are used to model the sequential behavior of single-system components in detail. Both model layers provide well-defined levels of error detection; the regular expressions enable the system designer to also determine and introduce redundancy to achieve error correction. The paper briefly outlines the methods we use to describe and analyze system behavior—fault detection and fault tolerance issues play a central role. The methods will be illustrated by a case study which contains a stepwise refined specification and analysis of a multistorey shelving system model which has been implemented following the method described here. The method described in this paper applies to any software system which is to be protected against the considered errors. It is not confined to control systems.

Index Terms—Software specification, Petri nets, predicate/transition nets, regular expressions, finite-state automata, fault tolerance, system modeling.

I. INTRODUCTION

Today, a growing spectrum of critical applications of software systems can be observed where an extremely high degree of reliability is required. Considerable efforts have been made to establish protection against simple hardware faults (e.g., single-bit stuck-at faults in memory) by means of appropriate hardware fault-tolerance techniques. Although such methods usually cover a large percentage of all potential error sources, for software parts several other potential error sources are still remaining. Generally spoken, errors in the code or input data of software system components may arise due to:

- erroneous data from the environment of the given system;
  e.g., errors caused by faulty behavior of other systems or user errors
- undetected hardware failures
- undetected design errors in the hardware/software components.

Many approaches have been proposed to make software systems robust against such errors: by introducing reset properties as to rollback, recovery lines, etc., and providing spare software mechanisms as to recovery block, n-version programs, etc. Recovery can then be accomplished in case an error is detected during run-time operation of the system. However, inherent to these strategies is the complexity of the utilized verification procedures as to acceptance tests, decision algorithms that usually deploy more or less heuristic, and incomplete tests for errors [6], [16].

In this paper an approach is presented that systematically takes care of the detection and correction of errors already in the design phase of the system to be realized. This is achieved by a combination of formal description methods—predicate/transition nets as a special class of Petri nets and regular expressions, well-known from the theory of formal languages. Our objective is primarily the practical application of these formal sound methods and not to contribute to the theories, which have been and are still the subject of enormous research efforts.

We will introduce a system model based on the above-mentioned theories in order to:

- Systematically investigate the fault detection/correction properties of concurrent and sequential software systems
- Add redundancy (if necessary) to the software system in a well-defined way to enhance the detection/correction capabilities.

Therefore we proceed as follows:

- Hierarchical specification of the system by a net representation which describes the concurrency issues. This net model is a high-order Petri net which provides a flexible modeling
- Analysis of the net model at a high system level by means of algorithms to detect faults in concurrent systems. These algorithms are efficient and well-known from the Petri net theory
- The fine-grain modeling of system parts whose structure is purely sequential deploys regular expressions
- At the low system level—i.e., regular expression layer—error detection and error correction capabilities of the model will be investigated. Moreover, the additional

Manuscript received October 10, 1988; revised January 2, 1991. Recommended by H. Weber. This work was supported in part by the Senate of Bremen, Department for Research, by the Deutsche Forschungsgemeinschaft, and by the Hochschule Bremerhaven.

The authors are with the Department of Electrical and Electronics Engineering, University of Paderborn, St. Augustin, Germany, and with the Gesellschaft für Mathematik und Datenverarbeitung (GMD), Bonn.

IEEE Log Number 9144258.
redundancy to achieve these capabilities, if not already provided, will be determined and introduced into the system.

With regard to the structure of the sequential processes which will be modeled by regular expressions, restrictions arise due to the well-known rules of structured programming. These restrictions have been proposed to obtain less-complex program structures.

For a software system, during the design phase our method leads to a two-level detection mechanism for run-time errors. Moreover, we can enhance the fault-tolerance properties by introducing additional structural redundancy. These properties are derived through a systematic analysis of certain structural relations between the components of the software system—not by considering the semantics of the application the system performs.

The method we derive applies to any software system which is to be protected against the considered errors; i.e., it is not restricted to specific classes of software applications. We will exemplify the benefits of our strategy by a case of a multistorey shelving system. This system has been designed and realized by applying the rationale summarized in this paper.

In Section II we first present some general properties of predicate/transition nets and regular expressions that will be utilized for modeling systems and processes. In Section III we outline how to combine these methods to specify fault-detecting and fault-correcting mechanisms. Section III also discusses how the described strategy might influence the internal structure of transitions in the net model. Our analysis is completed by considering a concrete application example: The specification, analysis, and extension of an automated shelving system.

II. ELEMENTS OF THE APPROACH

A. Predicate/Transition Nets

The best-known Petri nets are place/transition nets [3], [14]. For various software applications, place/transition net models have been used as a base for the specification of system behavior and for an exact description and analysis of concurrency structures. This is supported by a growing number of automated programming tools which help the user to: develop the graphical structure of such nets, study the net behavior by means of simulation, and systematically verify certain net properties, e.g., liveness, avoidance of deadlocks, etc. An overview of the net tools available at present can be found in [9].

Because of the simple, primitive structure of their constituents, place/transition nets are excellent means to lucidly exemplify general problems of system organization; e.g., conflict solving, deadlock avoidance, etc. On the other hand, because of the uniformity of the basic net elements, descriptions of larger systems by place/transition nets tend to produce very large graphical representations which are hardly comprehensible for the user. Alternatively, in the past years high-level Petri nets—e.g., the predicate/transition (PrT) nets—have been introduced which provide a more structured, compact, and lucid system description without losing any formal precision [3], [10].

PrT net descriptions appear to be a mature and comfortable means of system description [13]-[15]. In the following, we propose to deploy this net class for system modeling during the design and specification phase. Therefore the major objective of the net description is to specify the concurrence within the system. To model all purely sequential details of the system structure, we suggest the use of the finite-state automaton-based method of regular expressions. We shall first proceed by introducing the formal properties of PrT nets. Then the use of PrT net models for stepwise refinement of the system specification is described. Subsequently, in addition to the well-known firing rules of the transitions, we introduce an approach to extend the inner structure of transitions that incorporates checking mechanisms based on regular expressions.

Compared to the well-known place/transition nets (also called ordinary Petri nets), the predicate/transition nets have the following properties [3], [11]:

- Distinguishable items may be used instead of identical, unstructured tokens
- The conditions for the firing of transitions may additionally be specified by means of some logical or algebraic relations between the input and output variables of the transition
- Each transition of a PrT net may represent an entire class of events, independently fulfilling the firing conditions, and concurrently occurring. In the same way, a predicate may represent an entire class of elementary places. Thus PrT nets may be a representation much more compact in comparison to place/transition nets.

A formal description of PrT nets is given in the Appendix. Apart from being only a modeling tool, the transition net models also provide a tool for the evaluation of system properties; e.g., liveness and reachability of system states [4], [12]. This is mainly based on the analysis of reachability trees and of certain net invariants; i.e., properties of a given net which hold independently of the actual marking of the net. Such invariants can be derived from the adjacency matrix, which formalizes the connectivity of places and transitions within a net.

The invariant computation methods were first developed for place/transition nets. These methods can be established in a similar way for predicate/transition nets, although here the corresponding equation systems are much more complex [10], [11]. In Section IV-B we shall address by a concrete example such invariant analyses for PrT nets in more detail (fault detection at a high level).

B. Regular Expressions: Basic Notions

The basic assumption for the use of regular expressions to describe interactive processes is that the system is controlled by sequences of communication events (e.g., user inputs, inputs from other programs or devices, etc.). These sequences are modeled by regular expressions as a representing language; i.e., symbol strings which can be generated by a
Chomsky type-3 grammar [17], [18], [7]. Apart from their relevance for automata theory and formal languages, regular expressions provide an efficient tool for the modeling and analysis of a wide range of problems in practice. Specifically, all nonrecursive sequences of events in a system can be described by regular expressions [18]. The notation of regular expressions is short and precise and can be easily transformed into directed graphs or finite-state automata accepting all the symbol strings generated by the corresponding expressions. Efficient methods have been developed for the analysis and verification of system properties (completeness, consistency, complexity) or the synthesis of more complex expressions. They can be extended to describe concurrent processes also.

A detailed description of the use of regular expressions, especially for the software specification, is given in [18]. Therefore after introducing basic definitions, some special notations and techniques needed only for the following sections shall be considered here.

A regular expression consists of symbols combined by the following basic operations [18]:
- Sequence (concatenation) of symbols—this is not represented here by an explicit operator notation
- Selection between symbols—this operator is represented by "*"
- Iteration represented by "< ... >"—meaning an arbitrary i-fold (0 ≤ i ≤ ∞) concatenation of the contents of the brackets.

Therefore to model system behavior, a symbol of the expression may represent an arbitrary object; e.g.:
- A character of an input alphabet
- A sequence of characters
- A command
- A complex system activity, etc.

Therefore the system behavior may be described in a hierarchical order; i.e., a symbol itself may represent a regular expression.

Example 1: Let the symbols a, b represent arbitrary events. Then

ab: b is successor of a
a + b: either a or b occurs
⟨b⟩: an arbitrary number of instances of event b occur.

Example 2: Let us consider the slightly more complex regular expressions:

T1 = [(a(b + c))]; T2 = [(a + b)]; T3 = [(a(b + c))(a + b)].

These expressions may model simple file-access activities where "[", "a, b, c" represent certain user operations; e.g., ["open file," ""] close file, "a" delete, "b" read, "c" print a record.

As a specification, these expressions may have the following interpretation: T1 means that after the user has opened a file, the first record is read and immediately deleted. Then records may be read or printed in an arbitrary order. The entire process may be arbitrarily repeated. Finally, the file must be closed.

T2 represents an open-file operation, followed by an arbitrary order of read or delete operations; finally the file is closed.

T is a sequence of operations which are specified by T1 without its rightmost symbol, and by T2 without its leftmost symbol. Each sequence of operations such as "babababa, babbb, babacc,..." that is given by T represents a correct system behavior.

In the following, we shall demonstrate by this example,

\[ T = [(ba(b + c))(a + b)] \]

how a regular expression is transformed to analyze its fault-tolerance properties.

First we denote each instance of a symbol in the string T by its order of occurrence and obtain an indexed expression,

\[ T = [1(b1a1(b2 + c1))(a2 + b3)]]. \]

Here, "b1" means “first occurrence of b,” etc. In this way, symbols occurring in the expression more than once are uniquely denoted.

Furthermore, we use well-known methods (represented in, e.g., [17]) to construct the corresponding deterministic, finite-state automaton $E_{forward}$ that accepts all strings forwardly generated by the expression T. The states of this automaton are represented by numbers: a transition from state \( i \) to state \( j \) that is caused by the input of a symbol “a” is denoted by

\[(a)i := j.\]

Therefore the operation “:=” abbreviates that “the state \( j \) will be defined as a follow-on state of the state \( i \) transduced by the input \( a \)” The acceptor states may be seen to be equivalent to the symbol that leads to them:
initial state \( = 0 \),

\[(1) = (a^2) \quad 1 \Rightarrow 2 \equiv a^2, \quad (b) = (b^1 + b^3) \quad 1 \Rightarrow 3 \equiv b^1 + b^3 \]

\[ (0)_{(1^1)} 1 \Rightarrow 2 \equiv a^2, \quad (a)_{(1^2)} 2 \Rightarrow 3 \equiv b^3 \]

final state \( = 4 \).

Therefore if \( E_{\text{forw}} \) is in the initial state and reads symbol "\( a \)" (which can correctly only be "\( a^1 \)"), the automaton goes into state 1. The symbol "\( a \)" correctly read by state 1 is always "\( a^2 \)" and causes a transition into state 2, etc. Each symbol string not generated by \( T \) is not accepted by \( E_{\text{forw}} \); i.e., the automaton is transduced into a reject state \( e \). The graph and state table of \( E_{\text{forw}} \) are shown in Fig. 1. For reasons of more compact presentation, the reject state \( e \) is omitted. An additional relationship between the symbols \( s' \) of \( T \) and the states of \( E_{\text{forw}} \) is established if we replace the index \( i \) of the symbol \( s' \) by the set of states which contains \( s' \). Thus we transform \( T \) into a new expression:

\[ T_{\text{forw}} = \left[ \begin{array}{c}
1 \quad (b^2 + \bar{b}) \quad (b^3 + \bar{c}) \quad (a^2 + b^3 + c^3) + b^3 + 3^3 \end{array} \right].\]

This operation is called forward indexing of the regular expression \( T \). Additionally, we form the mirror image \( T_{\text{mirr}} \) of \( T \) and index it in the same way as \( T \):

\[ T_{\text{mirr}} = \left[ \begin{array}{c}
1 \quad (b^2 + \bar{a}) \quad (b^3 + \bar{c}) \quad (a^2 + b^3 + c^3) + b^3 + 3^3 \end{array} \right].\]

The finite-state automaton \( E_{\text{forw}} \) corresponding to \( T_{\text{mirr}} \) is constructed in the same way as \( E_{\text{forw}} \) (see Fig. 2; the reject state \( f \) is again omitted). Now we analogously derive a relationship between the symbols of \( T_{\text{mirr}} \) and the states of \( E_{\text{forw}} \):

\[ T_{\text{forw}} = \left[ \begin{array}{c}
1 \quad (b^2 + \bar{a}) \quad (b^3 + \bar{c}) \quad (a^2 + b^3 + c^3) + b^3 + 3^3 \end{array} \right].\]

This operation is called backward indexing. Here, to distinguish the indices in \( T_{\text{mirr}} \) from those in \( T_{\text{forw}} \), they are written in a subscript position. After a second mirror operation we obtain the expression:

\[ T_{\text{back}} := T_{\text{forw}} \text{mirr} \]

\[ = \left[ \begin{array}{c}
\{ s_1 \} \quad \{ s_2 \} \quad \{ s_3 \} \quad \{ s_4 \} \quad \{ s_5 \} \quad \{ s_6 \} \quad \{ s_7 \} \quad \{ s_8 \} \quad \{ s_9 \} \quad \{ s_{10} \} \quad \{ s_{11} \} \end{array} \right].\]

Performing forward and backward indexing simultaneously we obtain:

\[ T_{\text{forw}} = \left[ \begin{array}{c}
1 \quad (b^2 + \bar{a}) \quad (b^3 + \bar{c}) \quad (a^2 + b^3 + c^3) + b^3 + 3^3 \end{array} \right].\]

This double indexing is called a coding of the expression \( T \). The coding is the basis of all the tools we need for our error treatment. One important tool is the compatibility relation \( C \). \( C \) consists of all pairs \( (i, j) \) of a forward index \( i \) and a backward index \( j \) existing in a coded symbol \( s \) of \( T \). This is described by the notations \( iCj \) of \( isj \); the latter means that states \( i \) and \( j \) are compatible via the symbol \( s \). The relation \( C \) for \( T \) is given in Fig. 3.

As a second, more complex tool, the relations for the right context and left context \( r_{\text{forw}} \) and \( f_{\text{forw}} \) respectively, are used. They determine for every \( s_1 \) or \( s_2 \) the symbols which may appear as its immediate successor or predecessor, respectively.

By means of the compatibility and context relations we can now construct an automaton \( E_{\text{back}} \) (as a combination of \( E_{\text{forw}} \) and \( E_{\text{back}} \)) which accepts all coded-symbol strings \( w \) given by the regular expression. Therefore the states of the automaton \( E_{\text{back}} \) are again identified with the corresponding symbols \( s \) leading to them. Possible successor states of the automaton are given by:

\[ r_{\text{back}}(s_1) = \{ \text{from} : \text{forw}(s') \land e_{\text{back}}(s_j) \}. \]

The automaton \( E_{\text{back}} \) (compare context tables) is also shown in Fig. 3. In the following we show how the introduced notations are used to handle erroneous input data.

### III. Combining Predicate/Transition Nets with Regular Expressions

Practical applications of Pt nets showed that this modeling technique is accepted also by unskilled users if its utilization is restricted to those application areas and modeling levels where it is evidently superior to conventional methods. For the representation of purely sequential process structures, however, well-known conventional description methods as flow charts, finite automata, etc., should be preferred, since the use of nets here would always imply a higher complexity of the description.

Due to these experiences a general strategy for the stepwise specification of complex systems by means of Pt nets in connection with sequential methods is proposed here, especially to evaluate fault tolerance properties. Therefore in a first step the raw structure of the entire concurrent system is modeled by one simple net that is to give only an informal survey of the system. Correspondingly, this net description
contains only a small number of variables; the logical formulas in the transition have a relatively simple structure, since no detailed specification is intended here. Every internal detail of the variables, items, and tuples is either omitted or explained by informal texts.

This basic net is then stepwise refined by means of two mechanisms:

a. Transitions are substituted by subnets
b. Predicates, edge inscriptions, and logical formulas are refined by a reinterpretation process: the range of predicates and variables may be refined in its structure; variables and items may be substituted by tuples of variables or items.

The refinement process is iterated until the entire concurrency structure is graphically represented by a net whose transitions stand for a purely sequential subprocess. (In the case of a transitional form representing an entire class of concurrent events, at least every instance of the transition at this specification event has a purely sequential detail structure.) Such a subprocess receives its entire information from the input predicates of the transition and transfers results only to its output predicates.

The PrT net layer gives an overview of the system and all synchronization properties. Moreover, the net model can be used to quantitatively analyze the system flow; e.g., for potential deadlocks, reachability of desired final states, etc. Here especially the analysis of net invariants [4], [11] provides a simple tool which can be used for error detection at a high system level. By a scalar product of the actual marking of the net (i.e., a state of interest of the modeled system or process with an invariant), one can check whether this state is reachable from an initial marking or not. Such checks can be implemented at the level of the transition structure of the process to enable run-time error detection at a high system level. This will be illustrated by a case study in Section IV-B.

Methods to systematically solve the equations to derive invariants for PrT nets have been proposed; e.g., in [3]. To facilitate solving of the invariants, a method has also been recently proposed to canonically transform the PrT net presentation to a similar net structure—the so-called High Level Petri Nets, where the corresponding equation systems are algebraically more easily tractable [11].

For the specification of the sequential system behavior we propose the use of regular expressions as introduced in the preceding section. This method enables one to systematically integrate redundancy into the sequential processes of the single transitions, as will be presented in the next section.

After net modeling of the system, we specify the behavior of each transition by means of regular expressions. Also, this process has to be carried out by stepwise refinement if the complexity cannot be coped with within a single design step. Thus, on one hand, the internal operations sequentially performed inside a transition are specified by a regular expression which is stepwise refined up to the level of details which the system designer requires. On the other hand, each input item (which usually also represents a certain message, i.e., a string of data symbols) should be specified by an own regular expression.

Therefore the complexity of this expression depends on how many restrictions the system designer can pose to the inner structure of the particular message.

The next section outlines our approach to the utilization of regular expressions to implement fault detection and fault tolerance.

A. Error Treatment and Fault Tolerance Enhancement by Regular Expressions

For the systematic correction of a detected error, the system has to apply a certain hypothesis about the structure of the error. Most errors in the input string of a system component can be corrected if one of the following hypotheses is used (the letters "G", "H", and "J" used to identify these hypotheses have been selected arbitrarily; i.e., without making an acronymic sense):

1) Between two adjacent symbols of the string an additional symbol has to be inserted (G-correction)
2) At one position of the string a symbol has to be replaced by another one (H-correction)
3) At one position of the string a symbol has to be deleted from the string (J-correction)

These three hypotheses can correct errors caused by a missing, false, or superfluous symbol in the string. The hypotheses may be generalized to the cases, where, instead of a single symbol, a substring of n symbols is considered (n ≥ 1) (see Fig. 4):

1) Between two adjacent symbols of the string an additional substring of n symbols has to be inserted (G^n-correction)
2) A substring of n symbols of the string has to be replaced by another one (H^n-correction)
3) A substring of n symbols of the string has to be deleted from the string (J^n-correction)

The location in the string where a hypothesis might be applied is called a correction position, and the symbol used by the correction is called a correcting symbol.

If for a given erroneous string one of the hypotheses applies exactly to one position, then the place of the error can be localized unambiguously. Such errors are called Q^n-diagnosable, where Q^n ∈ {G^n, H^n, J^n}. The system is Q^n-diagnosing if all errors are Q^n-diagnosable. If two hypotheses P^n and Q^n out of {G^n, H^n, J^n} may be used for the same string, these are P^nQ^n-dependent; otherwise they are P^nQ^n-independent. Erroneous strings that can be corrected by use of one hypothesis Q^n, and by exactly one symbol are said to be Q^n-correctable. The system is Q^n-correcting if all errors are Q^n-correctable.

If an error is Q^n-diagnosable, Q^n-correctable, and all hypotheses P^n, Q^n are pairwise independent, it can be automatically corrected by the system itself; i.e., this error is P^nQ^n.
self-correctable. If for a system all errors are self-correctable, the system is said to be $P^n Q^n$ self-correcting.

The treatment of erroneous symbol sequences at run-time consists of the following steps:

a) Detecting that the sequence is erroneous
b) Localization of a substring, where a Q-correction may apply
c) Correction.

1) Detection of Erroneous Inputs: For this purpose the input string is successively indexed forward and backward (read from left to right by $E_{\text{forward}}$ and vice versa by the corresponding automaton $E_{\text{backward}}$). Thus if the string is correct, it can be transformed into a coding, as demonstrated by the example of $T_{\text{forward}}$. Otherwise, at certain positions the coding is no longer possible without violating the compatibility relation. Consider the example of an erroneous sequence $w$:

$$w = [bbbacabba]; \text{ the corresponding coding is } w_{\text{forward}} = \{b^3\sigma^15\sigma^1_{12}a^2c_2\} \sigma^1_2 = \{a^3\sigma_1\sigma_2\} \text{ (see Fig. 3)}$$

where $e$, $f$ represent the reject states of $E_{\text{forward}}$ or $E_{\text{backward}}$, respectively.

During the forward indexing the string cannot be read properly after the symbol $a^2$ by $E_{\text{forward}}$. For the backward indexing, the same happens at the symbol $b_7$ by $E_{\text{backward}}$. In these cases the automata are transduced into a reject state $e$ or $f$, respectively.

2) Localization: It has been proven [8] that the error which causes the automaton to stop during the forward and/or backward indexing is always situated in the substring between the leftmost symbol that can be accepted at backward indexing and the rightmost symbol that can be accepted at forward indexing. This is the only part of the string where, if possible, correction hypotheses may apply. Therefore this part is called a correction area.

3) Correction: Regarding an H-hypothesis, a correcting symbol placed between two other ones of the correction area must belong to the left context relation of its successor, and to the right context relation of its predecessor. This is demonstrated by the example in Fig. 5 of the correction area $k_{\text{forward}}$ of $w_{\text{backward}}$ (compare the context and compatibility tables in Fig. 3). Fig. 5 already shows that the given system is not H-diagnosing and that the hypotheses are pairwise-dependent.

For a complete analysis of the system, all correction areas of different length are systematically constructed. This requires that all combinations of symbols are constructed which “do not fit together,” i.e., which cannot be neighbors in a correct string. Correction areas of length $l = 1$ are obtained if all symbols are placed between neighbors which cannot be their left or right context. For the regular expression $T$ from example 2—a.g., in the strings $a^2ca_6$, $b^6ca_6$, and $b^6ca_6$—the symbol $c$ is not in correct context within its neighbors. Correction areas of length $l = 2$ are formed by the combination of all symbol pairs $(s_i, t_m)$ which are mutually not in context; i.e., $s_i \notin l_{\text{context}}(t_m)$, $t_m \notin l_{\text{context}}(s_i)$, for all $i, m$.

In this way, the Q-diagnosis and correction ($Q \in \{G, H, J\}$) capabilities of the system can be analyzed by means of a corresponding regular expression. The procedure can be generalized to the case of the arbitrary length of correction areas and arbitrary $n$ for $Q^n$ [8]. Thus we are able to analyze canonically in the design phase the fault-tolerance properties of the system. This provides the basis for improving its fault tolerance by use of additional redundancy.

4) Basic Idea to Achieve the Self-Correcting: If the analysis of a regular expression reveals that the system is not Q-diagnosing or Q-correcting or that hypotheses are mutually dependent, errors cannot be unambiguously diagnosed and corrected. By introduction of redundancy into the system, this missing property can be given to the system. This is illustrated by the example in Fig. 6. Since there are two positions where the hypothesis H might apply, a unique H-diagnosis and an H-correction are not possible.

The general idea in avoiding this problem is to embed in the regular expression at least one of the four symbols (s, u, t, or z; or v, x, y, or k) into a different, appropriate left or right context; e.g.:

$$s_i^j \rightarrow a \sigma_j^k b$$

so that the H-correction can only apply to one position. This process is called the extension of the regular expression. The extension yields fault-tolerant system behavior if the introduced redundancy is sufficient.

In [1] it has been proven that symbol embedding can be systematically performed to remove the considered ambiguities from the correction areas. It does not imply the necessity of additional corrective activities.

5) Generalization to Arbitrary Error Conditions: According to the derived principles, for every hypothesis we are now able to treat all correction areas where a Q-correction is not possible or where we have pairwise independent hypotheses. To do so, by complete analysis of all possible correction areas the symbols $s_i^j, u_i^j, t_i^m,$ and $z_j$, which are involved in a correction, are combined to a set $S$. Then by extension of some symbols of $S$ it is attempted to systematically reduce the correction positions to exactly one, and the correcting symbols also to exactly one. In the same way, appropriate elements of $S$ are extended to achieve independence of the hypotheses. Therefore the added symbols may differ with regard to the number of
eliminated correction positions, correcting symbols, etc., in correction areas.

To avoid unecessary redundancy, the number of the extended symbols in the sequence should be as small as possible. Therefore the set $S$ is minimized to achieve a specified fault-tolerance property.

For our example, it can be shown that $H$-diagnosability is achieved only by extending the symbols $a_2^0$, $a_0^0$, and $b_3$. To materialize this, on one hand we can use embedding symbols already existing in the string. In our case the extension then could be:

$$a_2^0 \rightarrow a_2^2a, a_0^0 \rightarrow ba_0^0b, b_3 \rightarrow ab_3.$$

On the other hand, to mark the positions where redundancy has been included into the string, it is often useful to introduce new embedding symbols. Choosing "$\$", "$-$", and "&" as additional symbols, then the extension could be performed as follows:

$$a_2^0 \rightarrow a_2^2\$, a_0^0 \rightarrow a_0^0 b_3^0 \rightarrow &b_3.$$

Using these, we obtain the extended expressions:

$$T1 = \langle bhab(b+c)\rangle(aa+ab) \text{ or } T2 = \langle b\#a\#(b+c)\rangle(a\$+&b)$$

which are $H$-diagnosing.

The principles shown here for the use of $Q$-corrections can be generalized to the cases of $Q^n$-hypotheses (i.e., $n$ errors have to be simultaneously corrected). Thereby the basic idea is to transform the simultaneous $n$-fold corrections into $n$ sequential steps.

The addition of the redundancy to the system may be performed stepwise, so that the degree of redundancy can be adapted to the required error detection/correction properties. As a further advantage of the method, the detection and correction of errors at run time can be performed by a straightforward algorithm without any backtracking procedures.

It is beyond the scope of this paper to present the derived method in all its details; a comprehensive description is given in [1]. A description of the tools which have been developed to apply the method in a comfortable environment can be found in [19].

B. Extended Transition Structure

As a consequence of the introduced specification strategy, a system description produced according to our method consists of two layers:

a) A PrT net; i.e., a set of predicates and transitions together with a connection structure of edges
b) For every transition and for every edge a regular expression.

As was shown, it is possible to systematically introduce redundancy into the expression so that the $Q^n$ error detection or $Q^n$ error self-correction ($Q = G, H, J; n = 1, 2, \ldots$) can be implemented as we have described in previous sections. Thus for each single transition the following formal fault model results where we distinguish three kinds of transitional faults:

1) An input fault is caused by a $Q^n$ error in one of the input data sequences of the transition
2) An operational fault is caused by a $Q^n$ error in the sequence of operations within the transition itself
3) An output fault is caused by a $Q^n$ error in one of the output data sequences.

The execution of the operations specified by one transition then implies the following steps (see Fig. 7):

a) The number of input items necessary to enable the associated transition must be available
b) The semantic properties of the input items as specified by the logical formulas added to the transition are checked; if they do not hold, the transition remains idle
c) Additionally, the structure of all input sequences is checked for whether a noncorrecatable $Q^n$ error is detected; if so, the transition only generates an error message
d) If no input errors have been detected, the transition is activated and its internal operations are carried out. Therefore either concurrently or after execution, it is checked whether the sequence of operations fulfills the specification given by the corresponding regular expression
e) Correspondingly, for all output strings it is also checked whether the generated output sequences fulfill the associated specification. If in phases d) or e) $Q^n$ errors are detected, then again only an error message shall
be generated by the transition. This especially implies that all changes in data made by the operations of the transition shall be committed—i.e., made valid—only after the acceptance test of the transition which is implemented by the checks for phases d) and e). Here, our concept of the transition structure coincides with the all-or-nothing properties of transactions known from the field of distributed systems. For such transactions, however, usually internal concurrency is admitted.

IV. CASE STUDY: MODELING A MULTISTOREY SHELVING SYSTEM AS A PRACTICAL EXAMPLE

In this section the feasibility of the introduced technique shall be demonstrated by means of a practical example. We shall consider an automated commodity storage management system as utilized in many trading companies. In such a system, incoming commodities are stacked on pallets. The pallets, being automatically conveyed by the system, are then usually stored in a multistorey shelving. The retrieval and processing of the pallets is performed by the control system (usually including a computer) which memorizes:

- The contents of the pallet
- The location of the pallet in the multistorey shelving.

Whenever an order for supplying commodities occurs, the control system decides which pallet(s) shall be used to fulfill the request. If all the commodities on the pallet are required, the whole pallet is conveyed to the system’s output point. Otherwise, the pallet is conveyed to a commissioning robot for processing. The robot may remove certain commodities from the pallet and replace them with others ("commissioning" process).

To perform the above-mentioned tasks, the system is comprised of:

- Conveyor-belts to convey the pallets within the system
- Turntables to change the direction of the pallets
- A commissioning robot
- A waiting circuit for pallets to be processed by the robot
- Automatic forklift vehicles (flv) to access the multistorey shelving
- Transfer points where the pallets can be transferred to or from the forklift vehicles
- Multistorey shelving organized in parallel lanes to enable concurrent access to pallet locations.

The topological structure of such a system is shown in Fig. 8; the photograph in Fig. 9 displays the overall system and its components.

From the input point, a pallet may be forwarded to the shelving, to the waiting circuit, or directly to the output point, depending on the invoked operation to store, commission, or directly output this pallet. The pallets are distributed due to one of these alternatives by means of turntables in the central conveyor belt, controlled by a real-time computer system. The conveyor belt is in fact a series of belt tables, each with a capacity of one pallet. Each table contains sensors which detect the presence of a pallet, read its identification code, and control its conveyance either to another table to the shelving or to the waiting circuit.

A. Modeling the Overall Features by Predicate/Transition Nets

As an example, we will consider a precise formal model of the processing of commodities in the multistorey shelving system. Initially, a commodity is ordered by means of an order form, \( o \). The order form is sent to an external supplier; the copy remains with the ordering company. After delivery, the commodity is entered into the system on a pallet. For the pallet there is then issued an identification tag bearing the contents of the pallet in a code suitable for automatic character recognition; e.g., bar-code reading. The pallet is then wrapped into a stretch or shrink foil film to avoid loss of its contents. Following that, a series of checks is performed as to whether these actions have been properly carried out. If one of those checks fails, the above-mentioned operations are retried—otherwise the pallet is automatically conveyed into the system (see Fig. 10(a)).

In a hierarchical manner we refine the component multistorey shelving of the system represented in Fig. 10(a), yielding the nets depicted in Fig. 10(b) and (c). We arbitrarily quit the refinement procedure of this transition at this step, since the achieved subnet is no more trivial but is still lucid; i.e., it contains evident transitions.

As a next step, we further refine the component pallet processing and obtain a more detailed level, as depicted in Fig. 10(c).

As a last step, we consider the further refinement of the transition pallet processing as represented in Fig. 10(d). Again, we arbitrarily quit the refinement procedure of this transition at this step, since the achieved subnet has again an appropriate complexity. Moreover, the procedure described by Fig. 10(d) contains only transitions which will be operated sequentially; i.e., the entire process is a sequential one.

Generally, following alternative actions are possible once a pallet containing an ordered commodity has to be transported:

- The pallet is conveyed to and stored in one of the shelves
- The pallet is conveyed to the commissioning robot
- The pallet is conveyed directly to the output point.

The distribution of the pallets according to one of these alternatives is performed by turntables in the central conveyor-belt system (predicate PALLET BUS).

Considering the case that the pallet, \( p \), is to be stored in a shelf (transition INITSTORE), the following operations are performed:
First, the pallet is conveyed to a transfer point. If the forklift vehicle (fv) serving the requested shelving lane, \( l \), is in position, it picks up the pallet. Simultaneously, it receives the location of the shelf from the control system, in which the pallet is to be stored. In our model, this is formally expressed by an address, \(<adr>\), consisting of the lane address, \( l \), the rack address, \( r \), and the number of the storey, \( s \) (it can be assumed that each shelf has a capacity of one, and just one, pallet). The forklift vehicle then moves to the designated location (predicate STORE ACCESS POINT). This is denoted by a formal sum:

\[ \langle p, adr \rangle + \langle p \rangle + \langle l \rangle. \]

This formula represents that the pallet, \( p \), is being transported by the forklift vehicle, \( l \), with an order, \( \langle p, adr \rangle \), and that \( p \) is to be placed at address, \( adr \). At the STORE ACCESS POINT with the location \( l, r, s \), the pallet is shelved. The fulfilled order is completed by a done message to the control system; this is represented by the tuple \( \langle p, adr \rangle \) flowing back to the predicate CONTROL. Subsequently, the forklift vehicle moves back to the initial transfer point (transition MOVE BACK).

Correspondingly, if a pallet is to be retrieved from a shelf, the control system delivers the location, \( \langle l, r, s \rangle \), of the pallet to the forklift vehicle which serves lane \( l \) (transition MOVE). After reaching the access point the forklift lifts the designated pallet from its shelf and transports it back to the transfer point, from where it may be directed to another part of the system.

As the second main alternative for a pallet after entering the system, the pallet can be conveyed to the commissioning circuit. After reaching the commissioning robot, commodities may be removed from or placed upon the pallet. For and during this process, empty pallets may be required and generated, respectively. It is therefore necessary to be able to convey empty pallets between the commissioning robot and the pallet rack (see Fig. 10(d)).

It cannot be stressed too strongly that each predicate and transition represents an entire multilayer scheme of pallet states and state changes. Therefore an arbitrary number of independent pallets may concurrently flow between the predicates. The necessary synchronization between pallets can very elegantly be expressed just by means of the capacities of the predicates. For a predicate name, \( i (i = 1, \ldots, n) \), representing a multilayer of \( n \) simple predicates, for each of these components the capacity is just 1 (summing up over the index \( n \) then gives a total capacity \( n \)). The net specification automatically models the necessary constraint that in one location of the system (cell of the central belt, cell of the commissioning loop, shelf cell, etc.) at most one pallet can reside.

The operations of functional units in the system are controlled by means of operation codes distributed by the control computer into the units. This is modeled by an additional item, \( \text{control} \), flowing into every transition. As an abbreviation, in most cases this item is represented by an uninterpreted edge (see Fig. 10(b) and (d)).

A transition is permitted only when all input predicates carry the necessary items. For instance, the transition LOAD of Fig. 10(b) needs the items \( \langle p, l \rangle \), \( \langle l \rangle \), and \( \langle \text{control} \rangle \). Additionally, before the transition starts according to our approach, a check of the input items must occur. Such a check can be performed by interpreting the items as an ordered string, e.g., \( \langle p, l \rangle \langle l \rangle \langle \text{control} \rangle \); this string can be modeled by regular expressions, therefore systematically introducing check redundancy.
B. Analysis of the Modeling Net to Optimize the System Structure

In our model, some combination of items has been shown by a formal sum, e.g.:

\[(p, adr) + (p) + (l)\]

as the output of transition LOAD (also named T2). Therefore the range of the corresponding output predicates is given by the union of the disjoint ranges for the item variables, \[(p, adr), (p)\] and \[(l)\].

Distinguishing between these items can be performed without the restriction that these items have to occur properly in an order; e.g., as a “super-tuple”:

\[\langle(p, adr), (p), (l)\rangle\].

It should be noted that the items of the formal sum in every case are not completely independent of one another. As an example, for transition T1 there is the restriction that in order to permit the transition, the value of \(p\) in \((p, adr)\) coincides with that one in \((p)\); in the same way for permitting T2, the...
value of (l) must coincide with that one of the component I of adr in (p, adr) (see above). In most cases, the items of the formal sums are tied together in a well-defined way; it is not possible that the item (p, adr) is moved by a transition together with an item, (p'), which represents a pallet p' different from p.

Let us now demonstrate analytical net evaluation by the example of net invariance analysis. For reasons of simplicity, we shall confine this to the subnet shown in Fig. 10(b). As abbreviations, the places and transitions of these subnets are additionally labeled, S1, . . . , S9 and T1, . . . , T7, respectively. Fig. 11(a) depicts the corresponding incidence matrix I which denotes adjacency between predicates and transitions: In this matrix, consider the case that a tuple, (a), flows from transition Tj to predicate St when Tj fires; this is denoted by setting the matrix element Ist to (a); a tuple, (b), flowing from Sj to Ti is represented by

\[ I_{ji} = -\langle b \rangle. \]

The S-invariants of a net are solutions x of the equation system:

\[ \Gamma x = 0, \]

where \( \Gamma \) is the transposed of the incidence matrix I. The inner product of such a solution array x and a marking M of the net remain constant under a sequence of firings of transitions in the net [3], [11]:

\[ xM = xMO = \text{const} \quad (MO: \text{initial marking}). \]

This feature provides a simple check as to whether or not a desired final state can be reached from an initial one by transitions of the net: The state can be reached if the constancy will not be violated, computed as the product of the array x and actual marking M. The incidence matrix of Fig. 11(a) leads to the equation system of Fig. 11(b). A solution of the equation system represented in Fig. 11(b) is given by the array,

\[ \langle x1, \ldots, x9 \rangle = \langle 1, 1, 1, 1, \ldots \rangle. \]

As an example of the invariance of the product of marking and the invariance array, let us consider as an initial marking, MO, for the analyzed subnet the situation that two pallets, p and p', are residing on S1, an item (p, adr) (i.e., (p, I, l, s))—see above—is on S9, and an item (l) is on predicate S3. It is evident that the product,

\[ MOx = \langle p \rangle + \langle p, \text{adr} \rangle + \langle l \rangle \]

remains constant under the firings of the subnet. This inner product constitutes a simple check whether, starting from an initial situation, the subsequent states are correct; a failure of the system which could be seen either as an incorrect behavior of a transition or the disappearance/appearance/changing of items on a predicate without firing a transition, would—at least with a high probability—also lead to a change in the inner product M x and thus could be simply detected. The presented example is a simple one, with a very compact and elegant solution of the equation system. For more complex cases, special “projection techniques” [3], [11], [13] are proposed to simplify the equation systems derived from the incidence matrix.

C. Modeling the Sequential System Components by Means of Regular Expressions

The PrT net we constructed in the previous section (Fig. 10(d)) presents the general features of the system to be modeled, especially the concurrent processes within the system. This net can be used to make the design issues of the system visible. Thus the overall structure of the system can be optimized before any implementation starts.

Based on the PrT net, we can produce instantaneous records or blowups when we trace the net. To perform such a record, we start at a predicate we are interested in, go through the transitions, and stop at an instance which can be final or not. Each time we reach a predicate which has multiple, concurrent transitions, we make an arbitrary decision to determine the next transition to be fired. The tacks we produce in this manner consist of pure sequential processes. They may occur as special constellations while operating the system.

In this section we will arbitrarily select such an instantaneous path through the PrT net depicted in Fig. 10(d). We will then analyze this path of sequential events by means of the methods described in Section III-A. Before the analysis, the path of events must be transferred to a regular expression.

As an example, in Fig. 10(d) we produced a sequence of transitions selected out of the PrT net represented in Fig. 10(c). (Again, we use the subscriptions 14, 15, etc., as abbreviations of the transitions.) The nodes represent events which occur subsequently. We transfer the diagram into the graph depicted in Fig. 12(a). Therefore for the sake of simplicity and transparency, the transitions represented in 10, 11, 12, and 16 have been integrated into adjacent states of Fig. 12(a) as expressed below in (2). In this example, if these transitions
were included separately, the analysis would proceed just analogously, without bringing any further results; causing, however, tedious expressions.

As the events $t_6$, $t_7$, and $t_8$ constitute a sequence without any loops or alternative outcomes, they can be abbreviated to a single event $tx$:

$$tx := t_6 \cdot t_7 \cdot t_8.$$  \hfill (1)

Correspondingly, we simplify some events:

$$t_8 : \text{ includes the adjacent state } t_{12};$$
$$t_9 : \text{ includes the adjacent states } t_{10} \text{ and } t_{11};$$
$$t_{13} : \text{ includes the adjacent state } t_{16}. \hfill (2)$$

The events $tx$, $t_8$, $t_9$, and $t_{13}$ are compound sequences which will be used to simplify the graph in Fig. 12(a). With these abbreviations we obtain Fig. 12(b). After having carried out the analyses for the error treatment according to Sections II-B and III-A, we will interpret the results in terms of the real system elements.

The graph in Fig. 12(b) can be represented as the following regular expression:

$$T = (t_4tx((t_4 + t_9)tx)t_9(t_{13} + t_{14}t_{13} + t_{14}t_{15}t_{13})). \hfill (3)$$

The coding of the term $T$ turns out as follows:

$$T_{foward}^{t_{13}^{foward}} = (t_4^2x(t_4^2 + t_9^2)x_2(t_4^2t_9^2)(t_13^2 + t_{14}^2 + t_{14}^2t_{13}^2)) \hfill (4)$$

Analysis of the Modeling Expression to Implement Fault-Tolerance: Based on the algorithms described in Section III-A, we can now analyze the regular expression we derived above. As an example, we carry out the analysis for $H$-errors; i.e., errors which occur when the operations will be mistaken or the symbols of the string have been corrupted.

As the treatment of the Fig. 13 correcting areas implies, the system which has been modeled via the regular expression is not $H$-correcting, and the hypotheses $H$ and $J$ are not independent. Furthermore, the complete analysis of the system implies the following results:

- The events $tx$ and $t_9$ will be involved in most errors
- $H$-corrections are often not unique.

As a next step, we demonstrate the self-correcting extension of the regular expression with regard to $H$-errors:

$$
\tilde{T} = \{t_4tx \cdot \text{end}((t_4 + t_9)tx \cdot \text{end})t_9 \\
(t_{13} + t_{14}t_{13} + t_{14}t_{15}t_{13})\}.
\hfill (5)
$$

The extended expression $T$ is $H$-correcting.

The expression $T$ in (7) has been produced according to the algorithm to achieve self-correcting features, as described in Section III-A. Here we extend $tx$ by means of the additional event "end" to the right:

$$tx := tx \cdot \text{end}. \hfill (6)$$

As a final task, we analyze the extended expression. Remember that $tx$ has been introduced in (1) as an abbreviation of three subsequent events $t_6$, $t_7$, and $t_8$. The extension in (6) concerns the rightmost event; i.e., $t_8$. The insertion of the corresponding transition $t_8$ of PrT net in Fig. 10(d) is: "check for correct operations." Therefore the extension end of $t_8$ due to (6) can be interpreted as a flag which signals the proper abandonment of the checks.

The extension (6) which has been determined according to Section III-B is very interesting, because it affirms the practical experiences made during the operation of the real system. The event $t_8$ of $tx$ has been placed in a critical position within the system: It is the final act before $t_4$ or $t_9$ can take place (see (4)). Moreover, $t_8$ is the final act of the inner iteration in (4). Thus the system needs additional information to recover an $H$ error in this position, because it cannot restore the previous constellation once it invokes such an error. The redundancy needed to treat this error could have been introduced into the system by means of a sensor which acknowledges the successful execution of the checks of $t_8$. Therefore we expressed the extending symbol in (6) as the "end" of the check of operations. Adding more sensors to the system in other positions would not increase the chance of proper operations or the system reliability.

It can be concluded that the analysis according to the theoretical results was confirmed by the practical experiences.
made during the operation of the system. As an example, \( t_8 \) really turned out to be a neuralgic component of the system.

V. CONCLUSION

In this paper a combination of methods is proposed which might be useful for the modeling and specification of the fault-tolerance properties of complex software systems. This comprises the use of advanced Petri net constructs for the description and analysis of systems, especially those with internal concurrency, on a high system level. Modeling of more detailed system structures is performed by means of regular expressions. This technique permits one to investigate the error detection/correction capabilities and to formally specify the redundancy to be added to achieve fault tolerance.

The described methods are exemplified by a practical case study in which a multistorey shelf system is modeled and analyzed.

VI. APPENDIX

A BRIEF REVIEW ON PREDICATE/TRANSITION NETS (P/T NETS)

Formally, a P/T net is given by the following structure [10], [11]:

1) A directed net \((S,T,F)\) with a set of predicates (represented by circles), a set of transitions (represented by rectangles), and a flow relation \(F \subseteq S \times T \cup T \times S\) (represented by directed edges) between them. A predicate is called an input or output predicate of the transition \( t \) if an edge exists from \( s \) to \( t \), or from \( t \) to \( a \), respectively

2) Some sets of items together with operations and relations defined for these sets (these are means to define logical formulae for the firing of transitions)

3) For each predicate, a referencing name together with the sets of items on which the predicate is defined. The number \( n \) of these sets defines the \( n \)-arity of the predicate

4) A marking of the predicates by a formal sum of tuples of items

5) An inscription of each edge by a formal sum of tuples of variables or items (cf. Fig. 14).

6) In some transitions, an inscription by means of a (quantor-free) logical formula composed of the introduced operations, relations, items, and the input and output variables of the transition

7) A function denoting for each predicate how many identical samples of an item may maximally reside at the predicate. This function is also called the upper limit of the multiplicity of the predicate.

8) The firing rule for transitions in P/T nets (see Fig. 14); a transition \( t \) is permitted to fire provided that all of the following conditions are fulfilled:

   a. The logical formula (if existing) in \( t \) is valid
   b. The marking of each input predicate of \( t \) contains a sufficient number of appropriate items
   c. The firing of the transition does not violate the upper limit for the number of identical items at the output predicates.

ACKNOWLEDGMENT

The authors wish to thank Dr. H. J. Genrich (GMD, Birlinghamen) and Dr. W. Bartscher (Hochschule Bremerhaven) for fruitful hints and proposals.

REFERENCES

[12] P. Huber, K. Jensen, and R. Shapiro, "Hierarchies in colored petri nets," in Proc. 10th Int. Conf. on Application and Theory of Petri Nets (Bonn), 1989 (to be published as Volume of Lecture Notes in Computer Science (LNCS)).
[13] K. Jensen, "Colored petri nets are becoming a mature specification language," in Proc. 10th Int. Conf. on Application and Theory of Petri Nets (Bonn), 1989 (to be published as Volume of Lecture Notes in Computer Science (LNCS)).
Fevzi Belli received the M.S., Ph.D., and Habilitation degrees in electricity and computing science from the Berlin Technical University.

He is presently a Professor of Computer Science in the Department of Electrical and Electronics Engineering, University of Paderborn, Paderborn, Germany, and a faculty member of the University of Maryland, European Division. Prior to this, he headed several software projects at a software house in Munich and was a Professor of Computing Science at the Hochschule Bremerhaven. His research interests are in software-fault tolerance, software reliability, programming techniques, and knowledge-based systems.

Dr. Belli is a member of the IEEE Computer Society, ACM, and the German Association for Informatics and its working groups, "software reliability" and "fault-tolerant computing," etc. He served as Program Chairman of several international conferences, including the Third International GI/ITG/GMA Conference on Fault-Tolerant Computing Systems in 1987.

Karl-E. Grosspietsch received the Diploma in computer science in 1974 from the University of Hamburg, and the Ph.D. degree in 1979 from the University of Bonn.

He is presently a Scientist at the German National Research Institute for Mathematics and Data Processing, the Gesellschaft für Mathematik und Datenverarbeitung (GMD), St. Augustin, Germany, where his research interests include computer architecture, fault-tolerant computing, VLSI architecture, VLSI design, and system modeling.

Dr. Grosspietsch is a member of the German Computing Society, the Gesellschaft für Informatik (GI), where he currently serves as the Vice-Speaker of the interest group, "fault-tolerant computing." He was the Program Chairman of the Second GI/ITG/GMR Conference of Fault-Tolerant Computing Systems in 1984.