Toward Accessibility Enhancement of Dependability Modeling Techniques and Tools

Ann T. Tai [†*] Herbert Hecht [†] Kishor S. Trivedi [‡] Bing Zhang [†]

[†] SoHaR Incorporated
Beverly Hills, CA 90211, USA

[‡] Duke University
Durham, NC 27708, USA

Abstract

Although various dependability evaluation techniques and tools have been developed in the last two decades, no adequate attention has been paid to allow system designers not well versed in analytic modeling to easily employ these techniques and tools. In this paper, we report our experiences on accessibility enhancement for off-the-shelf modeling techniques and tools. In particular, we discuss our approaches to the development of a user-friendly dependability-evaluation workbench which is intended to lead the user to exploit the features and capabilities of the modeling tool SHARPE.

1 Introduction

Dependability evaluation is an important activity for fault-tolerant system specification, design and maintenance. Moreover, fast turnaround time of dependability modeling process is one of the key factors for the efficiency of those activities and for the dependability of the resulting products since timely feedbacks will allow more iterations for design modification under the constraints of project schedule. In other words, a system designer should be permitted to quickly assess whether the current products (specification, design, etc.) satisfy the requirements and where modifications need to be incorporated. Indeed dependability evaluation can become more efficient and effective if modeling techniques and tools are made easier for system designers to employ such that they can handle and control the evaluation process themselves, instead of turning over the problems to the reliability/quality-assurance personnel. Although various dependability evaluation techniques and tools have been developed in the last two decades, significantly fewer efforts have been devoted to enabling system designers not specialized in analytic modeling to access those techniques and tools. Among a few system evaluation software packages that emphasized a high-level user interface, SAVE [1] provides an input language which permits system-oriented model specification. However, SAVE lacks a convenient graphical user interface (GUI). On the other hand, UltraSAN [2] has an excellent GUI, in which model and measure specifications are fully menu-driven and automated documentation facility is provided. Nevertheless, like most other Petri-net based tools, to use UltraSAN requires an adequate analytic background. Clearly, there is a gap in general between the terminologies associated with a particular model type (e.g., Markov chains) and those used in system engineering practice. In our view, a dependability evaluation software package should play a role to diminish this gap, permitting system designers to benefit from off-the-shelf modeling techniques and tools. With these motivation, we have developed a dependability-evaluation workbench called SDDS (System Dependability Evaluation for Design Solutions), aimed at fully utilizing the features and capabilities of SHARPE [3, 4] modeling engine. SDDS features a GUI with which system designers are permitted, through an *interface language*, to specify models interactively in terminologies they are familiar with. The *translator* in SDDS, which plays a prominent role in leading users toward effective utilization of SHARPE's features and capabilities, converts the high-level specifications into the most appropriate (internal) representations that can be automatically solved by the underlying modeling engine SHARPE. The translation and solution processes are transparent to the user.

Section 2 explains the relationships between SHARPE and SDDS. In Sections 3 and 4, we describe the interface language of SDDS and how the translator leads the user to make best use of SHARPE's features and capabilities, respectively. The concluding section summarizes what we have accomplished and discusses our future work.

2 The Modeling Engine versus the Workbench

SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) developed by Sahner and Trivedi is a modeling engine for analyzing hybrid, hierarchical models for a class of performance, dependability and performability models [4]. It provides a textual specification language and solution methods for a variety of model types: series-parallel reliability block diagrams, fault trees, reliability graphs, Markov chains, semi-Markov chains, series-parallel directed acyclic graphs, product-form closed queueing networks, and generalized stochastic Petri nets.

*Ann T. Tai is now affiliated with IA Tech, Inc., Los Angeles, California.

0731-3071/97 $10.00 © 1997 IEEE

Furthermore, any hierarchical combination of above model types can be specified and solved.

However, from the perspective of users with little analytic background, a number of SHARPE's useful features may not look user-friendly. First, SHARPE accommodates a variety of model types which facilitate dependability evaluations of fault-tolerant systems in a complimentary manner. For example, while the fault-tree solver handles fault-tolerant systems with both shared and dedicated components among redundant subsystems via allowing explicit specification for a "repeated node", the block-diagram solver emphasizes simplicity and does not deal with such systems. Thus, a non-expert user who lacks the knowledge about proper model decomposition will have difficulties in selecting the most appropriate model type to represent a particular aspect of a system. Clearly, an improper model decomposition may introduce significant errors in solutions. Second, SHARPE expresses numerical results of dependability measures based on a "mixture distribution" which allows various fault-tolerant systems to be characterized by *exponential polynomials* that easily lend themselves to computer manipulation. However, this may cause difficulties for non-expert users in interpreting evaluation results or intermediate results. For example, a user may have difficulties to understand why the mass at zero (usually corresponding to a system's initial unavailability) is positive when he solves a model hierarchically by passing numerical results from a lower layer to the top layer. Third, although the GSPN solver in SHARPE has the full power of stochastic Petri net, it provides only a small set of library functions, which prevents a non-expert user from conveniently specifying a model and dependability measures at a high level.

Aimed at making the powerful features of SHARPE accessible to system designers with little analytic background, we have developed a user-friendly dependability-evaluation workbench. In particular, accessibility enhancement is accomplished based upon two major components of SDDS, namely,

1. An interface language that permits the user to interactively specify models and access capabilities of SHARPE using terminologies they are familiar with.

2. A translator which converts a high-level model specification into the most appropriate *internal representation* that SHARPE can automatically solve and ensures the user to utilize the features and capabilities of the modeling engine in an effective way.

The interface language and translator are described in Sections 3 and 4, respectively.

3 The Interface Language

The interface language features the following ingredients which are implemented in *Tcl/Tk* and can be accessed by the user through a GUI:

A set of graphical block-diagram primitives that facilitates hierarchical model specification.

A taxonomy that categorizes those typical fault-tolerant system architectures and guides the user to map a system component to an appropriate type.

A collection of specification templates each of which is customized to a particular architecture type in the taxonomy and navigates the user to specify dependability attributes of a model component.

The set of graphical block-diagram primitives includes pointer, block, edge, join node, block-attribute specification and up/down arrows, as shown in Figure 1. Note that there are three types of blocks, namely, *simple block*, *composite block* and *library block*. The choice for block type will become available when the user selects the icon "block." A simple block represents a simplex component or a subsystem with N-modular redundancy (NMR or k-out-of-n, meaning that k out of the n redundant components must be operational for the system to remain in an "up" state, which is further discussed in the last paragraph of this section). Whereas a composite block enables the user to conveniently specify a block diagram in a recursive manner, facilitating hierarchical model construction. The user is allowed to move up and down in a model hierarchy to inspect and/or modify the specification using icons. (Note that the block diagram shown in Figure 1 is at its top level so the "up" icon is not available.) A library block refers to a "built-in" representation for a fault-tolerant system architecture that is rather complex such as a duplex system accommodating recovery blocks (a system incorporating both hardware and software fault tolerance mechanisms). A library block can be integrated with other (user-specified) blocks to compose a complete block-diagram representation.

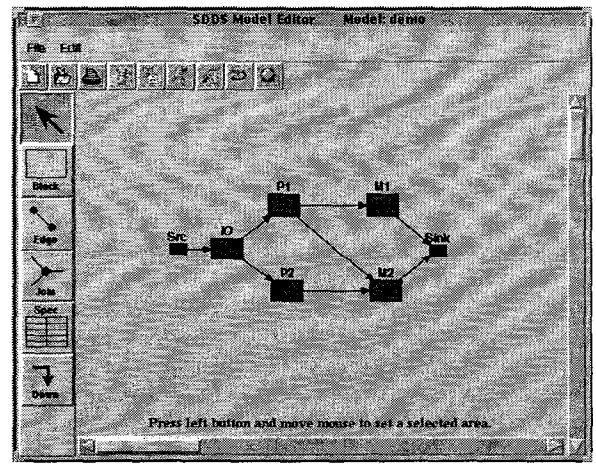


Figure 1: Block-Diagram Level User Input

To help the user map a system component to an appropriate representation, a taxonomy that categorizes typical fault-tolerant architectures (see Figures 2 and 3) becomes available on the screen when the user starts to specify a block (through using the "spec" icon). Upon the user's selection, a dependability-attribute specification template that asks for parameter values and success criterion *specific to the chosen type* will pop-up, guiding the user to correctly and completely characterize a model component. For example, if the user selects the type for the block "IO" as a repairable simplex subsystem (input/output device) with perfect coverage as shown in Figure 2, the corresponding template for parameter-value assignment will pop-up as shown in Figure 4 (a) (where the parameter-value entry for repair coverage is disabled). Whereas if the user selects the type for the block "P1" as a subsystem with non-identical redundant components (usually corresponding to different processing elements for functional/semantic redundancy) which are repairable with shared repair facility and imperfect repair coverages as shown in Figure 3, the corresponding template will ask for failure rate, repair rate and coverage for each component as shown in Figure 4 (b). Thus the combination of the taxonomy and the set of attribute templates facilitates *navigated model specification*, avoiding potential errors by an unexperienced user.

It can also be noticed from the figures that, the taxonomy and attribute templates enable the user to access a number of useful capabilities of SHARPE. Among other things, an important feature we let the user to exploit is that SHARPE permits explicit specification for redundancy and success criterion of a k-out-of-n system. Moreover, the taxonomy (see Figures 2 and 3) indicates that, although the non-state-space solvers in SHARPE (e.g., block-diagram and fault-tree solvers) restrict each of the redundant components in a repairable k-out-of-n system to have dedicated repair facility ("independent repair"), SDDS accommodates repairable k-out-of-n systems with shared repair facility for (identical or non-identical) redundant components. This is accomplished by converting the high-level user specification into a GSPN representation (described in the next section).

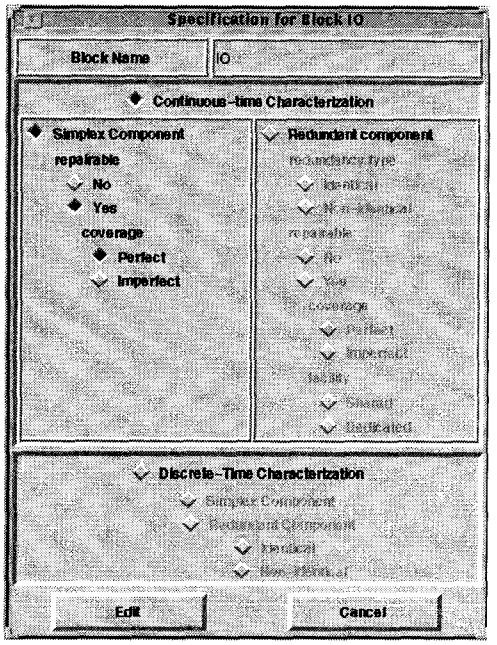


Figure 2: Block-Type Taxonomy (I)

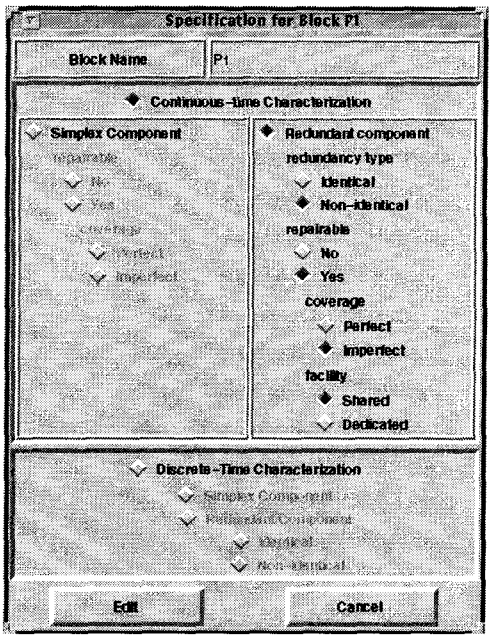
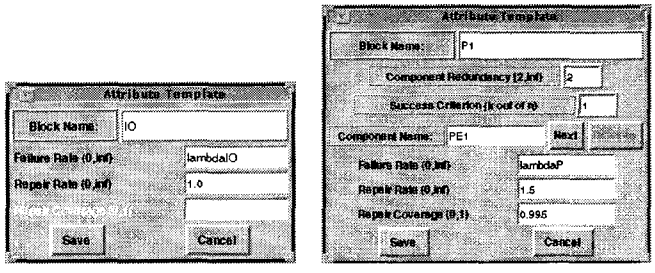


Figure 3: Block-Type Taxonomy (II)

4 The Translator

As mentioned previously, the translator plays a prominent role in ensuring that the user can access and utilize the features and capabilities of SHARPE in an effective way. In particular, the translator is intended to enable users to exploit SHARPE's hybrid/hierarchical modeling capabilities, which is described below.

Among other representation types, system designers are in general most familiar with block diagrams which are usually heavily used throughout a system's life cycle and thus are well suited at the user-input level in SDDS. However, as mentioned in Section 2, the block-diagram solver of SHARPE does not accommodate a system with both shared and dedicated components among redundant subsystems such as the one shown in Figure 1 in which the upper processing element "P1" has a dedicated memory module "M1" and another memory module "M2" shared with "P2." On the other hand, the powerful fault-tree solver in SHARPE is able to handle "repeated nodes" (a node that appears in more than one branch in a fault tree), which can be utilized to represent a fault-tolerant system of such type. Therefore, we choose fault tree as the *top-*



(a) (b)

Figure 4: Attribute-Specification Template

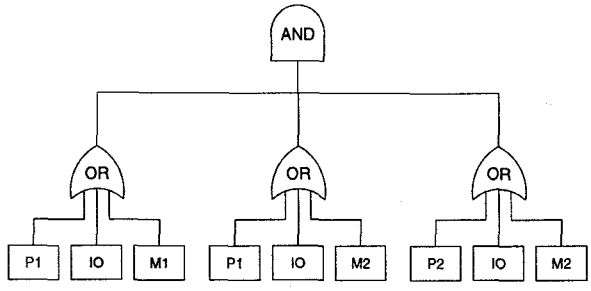


Figure 5: Equivalent Fault-Tree Representation

layer internal representation type. Accordingly, by taking the advantage that the fault-tree solver permits explicit specification for repeated nodes, we developed a simple yet efficient recursive translation algorithm. The algorithm exhaustively enumerates all the paths from source to sink in a block diagram such that any block diagram can be translated into a 2-level fault tree [5]. Figure 5 illustrates the fault tree translated from the block-diagram input shown in Figure 1. Further, the fault tree at the top-layer of the model hierarchy may need to be elaborated depending upon the characteristics of each component or subsystem. The translator takes the responsibility to make the decision on which model type (supported by the underlying modeling engine) a node in the fault tree should be converted to. Figure 6 depicts the decision tree for the translator in which rules are driven by the goal of enabling the user to exploit SHARPE's features and capabilities. Specifically,

1) The decision rules are intended to utilize as much as possible the simple and powerful fault-tree solver for better performance.

2) When Markov chain is considered as the candidate for the lower-layer representation, we choose to translate a block into a GSPN representation first instead of directly generating the Markov chain. This is because i) a GSPN representation is simpler and more concise than Markov chain (especially for systems with redundancy), which allows the translator to be less complex and, ii) the GSPN solver in SHARPE can be exploited to carry out the further conversion.

While the non-state-space solvers in SHARPE such as fault tree and block diagram allow the user to specify redundant systems with dedicated repair facility for each component, evaluation of such systems with shared repair (which is more typical than dedicated repair in real-life applications) in general requires the use of state-space solvers, which could be difficult for a non-expert user. On the other hand, with the taxonomy and translator's decision tree (see Figures 2, 3 and 6), the user can easily evaluate such systems and needs not to worry about the underlying GSPN/Markov-chain specification and solution. By utilizing the small set of library functions of GSPN, SDDS exploits the full power of the solver to allow the evaluation of redundant systems with non-identical components that have shared, perfect or imperfect repair. Figure 7 depicts the internal GSPN representation generated by the translator for a redundant system consisting of two non-identical-components with shared repair and imperfect coverage. In this figure, a special case in which $k = n = 2$ (i.e., a duplex system) is used for the simplicity of illustration. It can be observed that inhabit arcs are utilized to maintain the order of those non-identical components in a repair queue. For the particular marking shown in the figure, the token in the place "q21" is forbidden to enter the place "q22" (corresponding to the front of the repair queue) until the place "q12" (also corresponding to the front of the queue) becomes empty (the first component finishes repair). By utilizing auxiliary places and immediate transitions, any types of k-out-of-n system can be represented by GSPN in a similar manner. Based on these internal representations, the library functions available in the GSPN solver `prempty(kofnsys,totalUp)` and `premptyt(t;kofnsys,totalUp)` (probabilities of the place "totalUp" being empty at steady state and at time t, respectively) can then be exploited for dependability measures.

From the solution speed point of view, the translator is intended to improve performance by taking advantage of SHARPE's flexible interface between layers in a model hierarchy. For example, although we can take a straight forward fault-tree approach for the transient measures of a system where each of the repairable components has its own repair facility and perfect repair coverage, the translator chooses to convert the user-specified block-diagram for such a system into the SHARPE code with which the final solution is obtained by integrating the numerical results from each component system at the top-layer fault tree. Thus by avoiding a non-hierarchical approach that combines the exponential-polynomials (which characterize the components) before numerical evaluation, the solution speed is improved by several orders of magnitude.

As mentioned in Section 2, the output format of SHARPE for dependability measures based on a mixture distribution may not be easy for users with little ana-

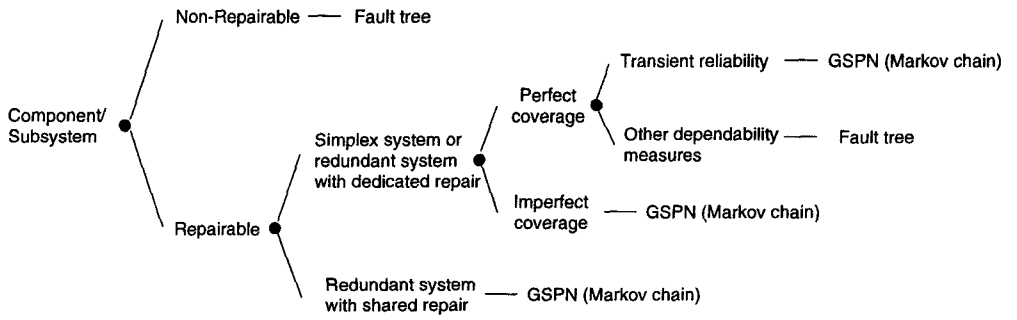


Figure 6: Decision Tree for the Translator

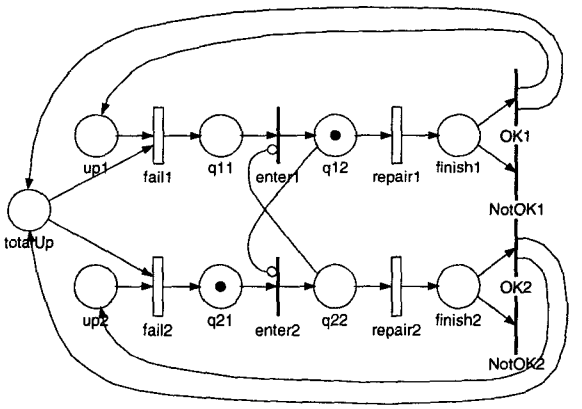


Figure 7: A Duplex System with Non-Identical Components and Shared Repair

lytic background to interpret. To circumvent this problem, the interface language and the input/output filters in the translator incorporate in a way so that SDDS is able to accept measure specification and display numerical results in more conventional terms (e.g., unavailability at time t, steady-state unavailability, etc.).

5 Conclusion and Future Work

Aimed at enhancing accessibility for off-the-shelf modeling techniques and tools for system designers, we have developed a dependability-evaluation workbench SDDS that is intended to lead the user to effectively utilize the features and capabilities of SHARPE. Although SDDS does not increase the analytic power of the underlying modeling engine (i.e., supported model types, solution efficiency, etc.), the interface language and translator can improve the turnaround time for system designers to assess their design alternatives because

1. Navigated model specification requires minimal learning of analytic techniques and reduces the time potentially needed for specification debugging.
2. User-transparent hybrid/hierarchical model construction aimed at optimizing the utilization of SHARPE's features ensures solution efficiency.

While a high-level user interface is a promising means of diminishing the gap between modeling process and system engineering practice, navigated model specification may introduce restrictions on tool usage. The limitations in the current version of SDDS include the lack of 1) capability of representing failure/repair dependencies among subsystems and, 2) flexibility that allows the user to obtain measures of subsystems for identifying the dependability bottleneck of a design. Accordingly, we plan to investigate into i) the feasibility of adapting iterative solution methods for the translator to accommodate repair dependencies across subsystems and, ii) the approaches to enhancing "traceability" in dependability evaluation. More generally, to investigate and resolve the conflicting criteria (e.g., simplicity versus flexibility) for modeling tool development is a challenging task for our future work.

References

[1] A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenberg, and K. S. Trivedi, "The system availability estimator," in *Digest of the 16th Annual International Symposium on Fault-Tolerant Computing*, (Vienna, Austria), pp. 84–89, July 1986.

[2] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko, "The UltraSAN modeling environment," *Performance Evaluation*, vol. 24, no. 1, pp. 89–115, 1995.

[3] R. A. Sahner and K. S. Trivedi, "Reliability modeling using SHARPE," *IEEE Trans. Reliability*, vol. 36, pp. 186–193, Feb. 1987.

[4] R. A. Sahner, K. S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems An Example-Based Approach Using the SHARPE Software Package.* Boston, MA: Kluwer Academic Publishers, 1995.

[5] A. T. Tai, H. Hecht, K. S. Trivedi, and B. Zhang, "Making dependability evaluation more effective and efficient for engineers," in *Proceedings of the International Workshop on Computer-Aided Design, Test, and Evaluation for Dependability*, (Beijing, China), pp. 145–150, July 1996.