A User-Friendly Dependability Evaluation Environment
for System Designers

Ann T. Tai  Herbert Hecht
SoHaR Incorporated
Beverly Hills, CA 90211
E-mail: {tai,herb}@sohar.com

Abstract

In order to permit users with little analytic background to evaluate dependability, modeling tools require a user-friendly front end. With this motivation, we have developed a software tool referred to as SDDS for "Software Dependability for Distributed Systems." In particular, we have designed and implemented a graphical user interface (GUI) based on a "user-language" — basic building blocks representing system components and/or subsystems. Therefore, the user can specify his/her dependability model at a high level. SDDS then translates the high-level specification into a representation that can be automatically solved by the underlying modeling engine SHARPE (from Professor Trivedi, Duke University). The translation and solution processes are transparent to the user. We describe the design of SDDS in terms of its "user language" and the translator that converts the high-level user input (model specification) to the low-level representation that SHARPE will accept.

1 Introduction

Dependability evaluation are crucial for system specification, design and maintenance. Moreover, fast turnaround time of dependability modeling process is very important for those activities. In particular, a system/software engineer should be able to quickly get ideas about how his current products (specification, design, etc.) satisfy the requirements and where modifications need to be incorporated. Dependability evaluation can be made efficient if user-friendly modeling tools are employed such that engineers can handle and control the evaluation process themselves, instead of turning over the problems to the reliability/quality-assurance personnel. With the motivation of developing a tool that can be used by system
architects and designers without specialized mathematical background or experience in model construction and solution, we have been developing a dependability evaluation environment called SDDS (Software Dependability for Distributed Systems) which features a graphical user interface (GUI) that permits the user to specify and arrange the system elements at block-diagram level [1]. SDDS then translates the high-level specification into a representation that can be automatically solved by the underlying modeling engine SHARPE [2]. The translation and solution processes are transparent to the user.

Section 2 provides a brief review of the modeling engine SHARPE. Section 3 then presents the design of SDDS. In Section 4, we illustrate the graphical user interface that enables people with limited analytic background to use the tool, followed by Section 5 which describes how the building-block level user input is translated to the code that the underlying modeling engine can accept. The concluding section summarizes what we have accomplished and points to some area for our future work.

2 Modeling Engine

SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) developed by Sahner and Trivedi is a tool for analyzing hybrid, hierarchical models for a class of performance, dependability and combined performance/dependability (performability) models [3]. It provides a specification language and solution methods for the following model types: series-parallel reliability block diagrams, fault trees, reliability graphs, Markov chains (acyclic, irreducible cyclic and cyclic with absorbing states), semi-Markov chains (acyclic and irreducible cyclic), series-parallel directed (acyclic) graphs, product-form (closed) queueing networks, and generalized stochastic Petri nets. Furthermore, any hierarchical combination of above model types can be specified and solved.

3 Tool Design

We are aimed at making the powerful features of SHARPE accessible to non-modeling specialists. The prospective users are the system architects and engineers who design, develop and maintain software/hardware systems. The requirements for the interface assume that users have only minimal dependability modeling skills. The minimum background presumed is that they understand the concept of component reliability and system state (operational versus non-operational) and are familiar with block diagrams.

Accordingly, we define two types of block that facilitate the user to evaluate a design or a system as follows.
User-defined There are two kinds of user-defined block, namely, simple block and composite block. A simple block represents a single system entity such as a processing or memory module, or a typical \( k\text{-}out\text{-}of\text{-}n \) redundant unit. A composite block represents a subsystem with its details elaborated by the user in a lower layer.

Built-in By selecting/incorporating built-in blocks from the SDDS library which has a collection of template representations corresponding to typical dependable or fault-tolerant systems, the user will be able to evaluate systems that are relatively more complex. The user can assign values to the parameters of a template system to specify system's behavioral (e.g., failure rate and repair rate) and structural (e.g., number of redundant processing elements) attributes. Built-in blocks are useful in the sense that they facilitate the prediction about whether incorporating a typical fault-tolerant architecture into the target system is feasible in terms of dependability improvement.

Figure 1 displays the overview of the design, where we can see that SDDS consists of the GUI, the modeling engine SHARPE, and a library. Further, the GUI has two major components:

1. User language: A set of the graphical elements which enable the user to specify models at a block-diagram level.

2. Translator: Two \( Tcl/Tk \) functions. The first function translates graphical user input into a textual representation; the second function translates the textual representation into SHARPE code ready for the analytic solvers of the modeling engine and converts the SHARPE's output into a format that is easily readable by the user (i.e., table and plot).

Note that the translator and the back-end tool will communicate with a library such that modules representing typical components, subsystems, or built-in template systems (see above) can be invoked and/or generated during an evaluation.

4 User Language

As mentioned previously, the key goal of the user interface design is to enable a user who has little knowledge about modeling to 1) specify his problems at a high level, 2) focus

\(^1\text{Tcl/Tk is a software package for developing and using graphical user interface applications in X-Window System.}\)
on the dependability attributes of the object system and, 3) to subsequently use SDDS for quantitative evaluation. Therefore, the low-level model construction and interface between submodels/layers should be made transplant to the user. This goal has been accomplished by automatically converting a high-level concise block diagram to a lower-level specification that SHARPE can accept as input. We call the set of ingredients for block diagrams (the graphical primitives for the user to specify a diagram) a "user language." To illustrate the idea of the concise block-diagram level input, consider the decentralized system shown Figure 2, which consists of two identical dispatch stations and three identical service stations [4]. Although this system can be translated into a conventional block diagram as shown in the upper part of Figure 3, the simplified block diagram version shown in the lower part of the figure provides us a more concise representation. The simplified representation not only reduces the clutter that the conventional forms produce on large systems, but also eases the processes of model specification, inspection (for correctness check), and modification because redundancies are explicitly specified as model components' attributes, instead of being represented by multiple components.

Figure 1: SDDS Overview

Figure 2: A Decentralized System
This user language takes advantage of a number of useful capabilities of SHARPE. Specifically, an important feature we exploited is that SHARPE allows the user to explicitly specify redundancy and failure criterion in terms of $kofn$, meaning that $k$ out of $n$ of the redundant elements (not necessary to be identical) must fail for the system being in a "down" state.

To enable the user to map a system component to an appropriate representation, SDDS provides a taxonomy when the user starts to specify a block as shown in Figure 4. Upon the user's selection, a tabular form that asks for values of the parameters and success/failure criterion specific to the chosen type will pop-up, guiding the user to completely characterize a model component. For example, if the user selects the type for the block "memory1" as a redundant subsystem with identical components (memory modules) which are repairable with perfect detection coverage and imperfect repair coverage, the corresponding table for parameter value assignment will pop-up as shown in Figure 5.

5 Translator

Depending upon the characteristics of the system and dependability measures specified by the user, the translator makes decision on which model type (supported by the underlying modeling engine) the block-diagram level input shall be converted to. The decision rules for the translator are outlined in Figure 6. It is worth noting that, when Markov chain is considered as the candidate for the translation, we can either directly generate the Markov chain or translate a block to a generalized stochastic Petri net (GSPN, [3]) representation first and then convert it to a Markov chain. We choose the latter method because 1) a GSPN
Figure 4: Taxonomy for Block-Type Specification

Figure 5: Element Specification
representation is simpler and more concise than Markov chain (especially for systems with redundancy), which allows the translator to be less complex and, 2) the GSPN solver in SHARPE can be exploited to carry out the further conversion.

Figure 6: Decision Tree for the Translator

To implement the translator which is responsible for converting a block diagram to a fault tree, we have developed a rather simple and efficient algorithm, which is described below.

5.1 Algorithm

Malhotra and Trivedi developed an algorithm to translate a reliability graph to a fault tree [5]. Based on those ideas, we derived an algorithm to translate a block diagram to a fault tree. The translation can be divided into two stages. In the first stage, all the simple paths from the source node to the sink node are exhaustively searched and recorded. For each path, an OR gate is constructed to connect all the nodes (excluding the source and sink nodes) in the path and represent the (dependability) relationships among them. In the second stage, a top-level AND gate is constructed connecting all the OR gates previously constructed.

To exhaustively enumerate all the paths, a depth-first search is employed. Figures 7 and 8 display the search function DFS and the recursive function that DFS utilizes, respectively.

In the pseudo code, $u$ is the source node, $x$ is the sink node; $\text{adj}(u)$ corresponds to the set of the neighboring nodes of $u$. Upon the completion of the search, the set $\text{CompletePath}$ shall contain all the simple paths from the source node to the sink node.
DFS
for each v in adj(u)
    Append(PartialPath, v)
    for each w in adj(v)
        DFS-Visit(w, x, PartialPath)
    end
end

Figure 7: DFS Function

DFS-Visit(W, X, PP)
    if W = X
        Add(CompletePath, PP)
        return
    else
        if W in PP
            return
        else
            Append(PP, W)
            for each y in adj(W)
                DFS-Visit(y, X, PP)
            end
        end
end

Figure 8: DFS-Visit Function
5.2 Example

We use the architecture of the Pluto Express Data System (see Figure 9) [6] as an example to illustrate the conversion algorithm. Pluto Express is a NASA mission to explore Pluto, the only unsurveyed planet in our solar system. Currently, Jet Propulsion Laboratory is performing studies to achieve the mission objectives. In order to meet the stringent mission reliability requirements, the Pluto Express Data System employs a dual-string fault-tolerant architecture and applies the I/O cross-strapping techniques. Figure 10 shows the corresponding top-level block diagram specified using the SDDS model editor.

![Block Diagram](image)

Figure 9: Pluto Express Data System Architecture

Note that for this particular block diagram, six paths from the source node to the sink node will be enumerated by the searching function DFS as follows.

**Path 1** I/O1 stringA memory1
**Path 2** I/O2 stringB memory2
**Path 3** I/O1 stringB memory1
**Path 4** I/O2 stringA memory2
**Path 5** I/O1 stringB memory2
**Path 6** I/O2 stringA memory1

Accordingly, the translator will construct the following six OR gates:

**ORgate1** I/O1 stringA memory1
**ORgate2** I/O2 stringB memory2
Figure 10: Block-Diagram Level User Input

ORgate3 IO1 stringB memory1
ORgate4 IO2 stringA memory2
ORgate5 IO1 stringB memory2
ORgate6 IO2 stringA memory1

Subsequently, the top-level AND gate will be constructed:

FTree-Ex ORgate1 ORgate2 ORgate3 ORgate4 ORgate5 ORgate6

Figure 11 depicted the resulting fault tree representation generated by the translator.

In general, a system represented by a block diagram will not fail as long as there is at least one operational path from the source to the sink. On the other hand, a failed node will block all the paths where it locates. It is clear that our translation algorithm directly implements these rules. It is worth noting that the paths in the block diagram shown in Figure 10 “share” the nodes (e.g., the node stringA is shared by Paths 1, 4 and 6). Correspondingly, the resulting fault tree shown in Figure 11 contains “repeated” nodes. The fault tree model type in SHARPE is uniquely implemented to handle those situations [3]. Accordingly, our translator exploits this feature by using the operators repeat and transfer available in SHARPE when repeated nodes are detected in any paths, such that 1) any block diagram can be translated into a 2-level fault tree, as a result, the algorithm for the
Figure 11: Equivalent Fault Tree Representation
translation becomes much simpler and, 2) the user needs not to beware of those modeling details.

6 Summary and Future Work

We have been developing a user-friendly dependability evaluation environment that accepts model specifications in terms familiar to system/software engineers. In other words, SDDS permits model construction with block diagrams and building blocks such that users without dependability evaluation background can easily and quickly assess design alternatives by adding, deleting or moving blocks (or subsystems represented by "composite blocks") around and increase/decrease component redundancies.

Currently, we are enhancing the GUI-level error-checking facility for model specification, aimed at making dependability evaluation process more dependable. In order to realize the objective of enforcing software engineering practice, we plan to integrate SDDS into one or more computer-aided software development environments (CASE tools) in the future.

Acknowledgments

This work was performed under the Small Business Innovative Research (SBIR) Contract F30602-94-C-0232 from the U.S. Department of Defense. The authors want to acknowledge the support and interest of Mr. Andrew J. Chruscicki, the technical monitor for this project. We also wish to acknowledge Prof. K. S. Trivedi of Duke University (Durham, NC) for his participation in the conduct of this research. The programming efforts were contributed by Bing Zhang of SoHaR Incorporated.
References


