INTERNATIONAL JOURNAL OF INTELLIGENT CONTROL AND SYSTEMS
VOL. 10, NO. 2, JUNE 2005, 175-187

Model-Driven Design of Stable Software Systems: A Petri Net Based Approach

Srini RAMASWAMY*, Gopal YAMIJALA**, Rajaram NEELAKANTAN+ and P. Kari RAJAN++

***Abstract*— In this paper, stability of event-driven software systems is studied in terms of its operational failures due to poor design and/or implementation flaws. The primary objective of our work is to develop a usable model driven design methodology for stable software system design, build a Petri net (PN) based application tool and illustrate its use through a simple case study. PN based techniques are employed to analyze factors that affect software stability, by utilizing PN properties such as boundedness, reachability and reversibility. An analysis tool, written using MATLAB software, is developed and utilized to identify sources of these problems. The analysis helps in delineating those points (transitions) in the software system that need to be monitored to prevent unstable operation. These transitions can then be supplemented by guard functions so as to alleviate problems arising from irreversibility, unboundedness, and unreachability. The tool can also be used to: (i) Detect partially disconnected and completely disconnected subnets to verify reachability and irreversibility, (ii) Detect unbounded places and states in the PN model to verify unboundedness, (iii) Detect minimal empty siphons (potential deadlocks) to check for deadlocks, another source of irreversibility.**

***Index Terms*—DEDS, Petri Nets, T-Invariants, Siphons, Traps, Software Stability, Stability Analysis**

1. INTRODUCTION

The essential qualities of a well-designed software system are faster execution, consumption of minimum resources, modularity, and minimum reengineering overhead costs in terms of time and money [1][2]. However, a critical requirement is that the system should never crash or lead to unrecoverable states. A good example is the need for detecting "unbounded buffers" in networks when the rate of reception of packets is more than the rate of transmission of packets at a network hub. Badly designed software solutions exhibit problems such as unboundedness (when any state of the system becomes unbounded), irreversibility (when the system execution cannot proceed meaningfully) and unreachability (when there are unreachable states in the system). The effects of bad design have been observed in several systems, including, real-time firmware, hybrid systems, operating systems, telecommunication and computer networks, and complex web based applications. Such design not only leads the system to an unpredictable state and an eventual system crash, but also places a heavy demand on resources. Thus it becomes very important to analyze and design software systems so that they function properly.

Designing a software system from a reengineering perspective (involving minimum reengineering costs in terms of time and money) solely depends upon the domain intuitiveness and skill/ability of the designer. Designing a system from a performance perspective (efficient use of resources and improved execution time) depends upon the clarity of the system's functional specifications. However, designing a system from an operational perspective (boundedness, reversibility and reachability) requires extensive analysis of the system model. In this work we have used "function" to indicate functionality within a small, well-defined design module, while we have termed "operational" the correct execution of several associated funtions across multiple levels of design abstraction.

By definition, system stability is an operational concept. Hence, for stability analysis of a system, critical sections of the system that lead to system failures due to unboundedness, irreversibility and unreachability need to be identified. Thus, in essence, the study of software stability is inherently paradigm independent, i.e., the software stability theory should be independent of any programming language paradigm (OO, functional, imperative, etc.) and based upon a sound theoretical framework. Viewing stability from a reengineering or performance perspective is obviously paradigm dependent and hence limited by different problem domains or the paradigm itself, whereas viewing it from an operational perspective is paradigm independent and can have a sound theoretical underpinning.

In essence, three primary problems in a software system design may cause software instability and hence a system failure. They are: *(i)* irreversibility: Irreversibility is when the software system reaches a system state from where it cannot continue its execution meaningfully, *(ii)* unreachability: This happens when the software system consists of "methods" or "procedures" that are unreachable by any valid sequence of system operations,

Manuscript received November 1 2004, Revised April 10, 2005. This work was supported, in part, by the Center for Manufacturing Research, Tennessee Technological University.

* Srinivasan Ramaswamy is currently professor and chairperson of the Computer Science Department at University of Arkansas at Little Rock. (e-mail: srini@ieee.org / srini@acm.org).

** Gopal Yamijala (e-mail: gopal_sat@hotmail.com) currently works as a Senior Software Consultant in B2B ecommerce applications.

+ Rajaram Neelakantan (e-mail: n_rajaram@hotmail.com) currently works as embedded software engineer at Stanley Assembly Technologies in Warren, MI.

++ Kari Rajan is professor and chair of the Department of Electrical and Computer Engineering, Tennessee Tech University.(email: pkrajan@tntech.edu)

and *(iii)* unboundedness: This happens when the software system enters a state wherein system resources are inefficiently used and state becomes unbounded (infinite loops, use of system resources, etc.). It is very important to check for these conditions on a system in its initial modeling phase and after that whenever the software is subjected to a change.

In our approach we use Petri Net (PN) techniques [3, 4] for studying stability. PNs, being model-based tools, can be used for building stable models by means of their analysis capabilities that can support both graphical and mathematical analysis. They also have the capability of capturing the dynamic nature of the systems' behavior [5-8]. Hence, in the quest for more expressive (both graphically and analytically) modeling tools, our work uses PNs for software system stability analysis. Such an analysis, i.e. "with selectively exploded subnets", is very important in resource constrained applications, including most software applications. Most publicly available tools often do not support hierarchical structures, and even if they do their modeling of hierarchical structures is not very intuitive – and often do not deal well with "flow" tokens – tokens that are generated as part of the net evolution(for example, SPNP), and are not part of the initial system marking. The tool developed here can analyze hierarchical nets in which subnets can be selectively exploded for inclusion in the invariant analysis process – Currently most freely available tools either do not support such a process, or they do not perform this adequately. Such an analysis consisting of selectively exploded subnets is very important factor in the design of resource constrained applications, including most software applications. Since most available tools do not specifically support such an activity, we have integrated these functions into a usable and "expandable" framework.

It is perhaps helpful to differentiate software stability with software reliability. Reliability means much more than stability – it can include trustworthiness, dependability, etc. By stability we address only the issue of system design such that it does not fail/crash during normally planned operations. However, it can be the baseline over which a very reliable system can be built (where other elements of reliability are enhanced).

The objective of this paper is to develop a "well-integrated" tool by leveraging existing works to address the issues that lead to software instability. The problem of integrating the detection of unboundedness, unreachability and deadlocks is more complicated in software systems because of the intricacies involved in following a multi-level hierarchal abstraction. Such deep abstraction layers often leads to insufficient testing and hence is at the root of several instability problems. By allowing for the inclusion of subnet explosion in a hierarchical PN model of a software system, we have effectively addressed this concern. Our experience with software systems, recently in large scale enterprise java applications, indicates that in Java based systems, chances of encountering any instability problems are scarce since Java inherently does not support unreachability and Java compilers are intelligent enough to detect it, unlike C or C++ compilers. However, traditional problems like unboundedness and deadlocks still seem to baffle programmers due to different JAVA Virtual Machine (JVM) implementations on different operating systems (for example: garbage collection). Hence this paper addresses a very critical issue with respect to software design and development for event-driven (stimulus-response kind of) systems. The use of a more analytically driven approach or PNs is also very different from well-established commercially available heuristic tools such as UML, and OMG's model driven architectures.

The paper is organized as follows: Section 2 provides a brief introduction to software stability, discrete event dynamic systems (DEDS) and its relationship with event-driven software systems. Section 3 introduces PNs briefly and discusses PN analysis techniques that are of consequence to this paper for those readers unfamiliar with PN terminology. The tool architecture is presented in Section 4. A manufacturing control software system case study is presented in Section 5. Conclusions are presented in Section 6.

2. BACKGROUND

2.1 Software Stability

A model driven approach to stability analysis of a software design process is presented in [2]. It presents a definition of software stability from a reengineering perspective based upon object oriented design principles, where software stability is said to be achieved when the software withstands constant changes in requirements and involves minimum reengineering. However this approach does not deal with the behavioral and operational aspects of software stability. Reengineering of a software system is unavoidable and often necessitated by changing client requirements. Requirements usually change *(i)* when there is a necessity to add additional features in the software; *(ii)* when the system needs to be adapted to some other environment (often observed when the software system has been previously designed for a specific operating system, application/web server or database); and *(iii)* when there is a need to enhance the system performance (use of minimum resources, deployment of efficient algorithms, and need for minimum execution time). These changes can be summarized as changes in requirement specifications, need for more software adaptability or portability, and need for better performance. However changing the design of the software does not automatically guarantee stability against system failures. Thus any change made to the system should always be followed by a basic stability analysis procedure, i.e. the analysis of unboundedness, irreversibility, and unreachability. Thus improving the stability performance of the software is a repetitive process during software systems' lifetime. Equilibrium is achieved when the occurrence of unboundedness, irreversibility, and unreachability is minimal and the design specifications are met (adhering to the requirement and functional specifications in the design).

Designing a software system from object-oriented and

reengineering perspectives provides a very narrow focus on the issue of software stability. The issue then becomes the problem of avoidance of reengineering overheads. However system performance and failures are not emphasized. Hence the intent of just minimizing reengineering overheads provides a narrow definition of software stability. Designing the system from a performance perspective clearly depends upon its functional specifications. The designer needs to identify critical resources and temporal relationships between the execution times of different software modules [2, 3]. The issue of identifying critical resources and their efficient usage again depends upon the skill and the intuitive ability of the designer and/or programmer. Designing the system from a performance perspective also does not guarantee stability.

2.2 Stability Issues

In general, stability is defined in many traditional domains as the quality or attribute of a system to maintain equilibrium and constancy of behavior. The reengineering or performance-based definitions of stability do not adequately address this generalized definition of stability. In this paper we identify three primary reasons for software system instability, derived from a study of stability in other domains.

2.3 Irreversibility

Irreversibility, a primary cause of instability, refers to an execution sequence in the system that may lead it to an unrecoverable state, i.e. once this state is reached, the system cannot recover its normal execution or functionality. Thus the initial state of the system is said to lead to irreversibility. In a software system, an example of an unrecoverable state would be a function call with no valid returns. Once this function is called, the system does not know where to resume its execution and hence waits/suspends all its other processes.

Another source of irreversibility is the deadlock problem [1, 4], i.e. a situation when the system continues to remain in the same state (or states) irrespective of any instruction execution. A deadlock is thus, a perpetually unrecoverable and irreversible system state which can only be overcome by a system reset. Simple deadlocks occur when two or more processes are trying to use a single resource concurrently and are dependent on each other for their successful execution. Deadlocks can also occur when there is lack of proper handshaking between two parallel dependent processes. Hence solutions to overcome deadlocks are to have extra resources for resolving mutual exclusion, or have appropriate algorithms to resolve handshaking. All such problems can be traced to partially disconnected code that causes irreversibility in the corresponding model.

2.5 Unreachability

This is another source of instability in software systems, caused by the existence of certain system states, which are unreachable from the initial system state. This often occurs due to the presence of execution (code) sequences in the system, which are not connected to the main control module. Such a problem is often brought about by incremental changes that leave behind areas of "dead" code. If the system reinitializes to begin execution from such modules, then it never reverts back to the main execution sequence. Functions or modules without any calls and returns model this condition.

2.6 Unboundedness

Unboundedness is the third cause of software system instability and arises when there are resource restrictions or when a certain process or module can handle only a limited quantity, or value, of data. The effect of unboundedness is gradual and once all resources are consumed the system malfunctions. Segmentation faults, array out of bounds exceptions, etc., are classical software problems caused by unboundedness.

2.7 Classes of Software Systems

In general, our view of stability is with regard to software system failure at any stage in the software development process. Our work focuses on the identification of those classes of software systems that are prone to instability due to bad design. The class of software systems considered in this work are event-driven, i.e., the execution of the code depends upon the occurrence of an event, such as the invoking of a function from the graphical user interface or an internal function or an external process. Such event-driven software systems can be classified as a special class of discrete event dynamic systems (DEDS).

A DEDS is a dynamic system that evolves in accordance with the abrupt occurrence, at possible unknown irregular intervals, of physical (discrete) events. Since DEDS evolve by event occurrences much like event driven software systems, much of the theory and research from DEDS stability can be effectively applied to even-driven software systems. These applications require supervision and control to ensure orderly event flow. Due to the complexity of such systems with advancements in man-made technology, ad hoc solutions have proved inadequate in the analysis and design of such systems. Hence researchers have used more detailed formal methods (finite-automata, PNs and formal language models) for system description and formalized analysis and design. There are large numbers of research papers dedicated to PN (and several related extensions) applications to DEDS [10-14], scattered in hundreds of journals and conference proceedings. Examples of conversions to PNs from other popular software modeling notations (such as UML) have also been reported [15-17].

While the application of powerful mathematical tools such as linear state space analysis is one approach to investigating software system stability, the knowledge that software executes in different phases, and hence, each process can be described as a discrete event provides a different perspective. A software process transitions as a state from a previous process, and after the completion of a task, it proceeds via a transition to a different process. So a software system can be then defined in terms of a state space with each transition representing a localized dimension of the whole execution process. Thus, the following are some of the advantages of using PNs as a modeling and analysis tool: *(i)* They provide a good visual model of the interconnections between various system elements

and can be analyzed mathematically. *(ii)* PNs can be used to model a variety of software systems that exhibit critical characteristics such as concurrency and dynamism. *(iii)* They can effectively represent quantity through token multiplicity. *(iv)* A variety of system properties like boundedness, conservativeness, liveness, reachability and fairness can be studied using corresponding PN models.

2.8 Defining Software Stability

As discussed above, unboundedness, irreversibility, and unreachability are three primary problems that can occur in software systems due to flaws that creep in during the modeling and design phases. These three factors are responsible for several kinds of operational failures. By failure, we refer to the situation when the software system is not functioning in its normal (expected) operating sequence. Thus we introduce a new definition of software system stability based upon operational failures – *A software system is said to be stable from an operational perspective if it does not fail for any appropriate input and initial state*.

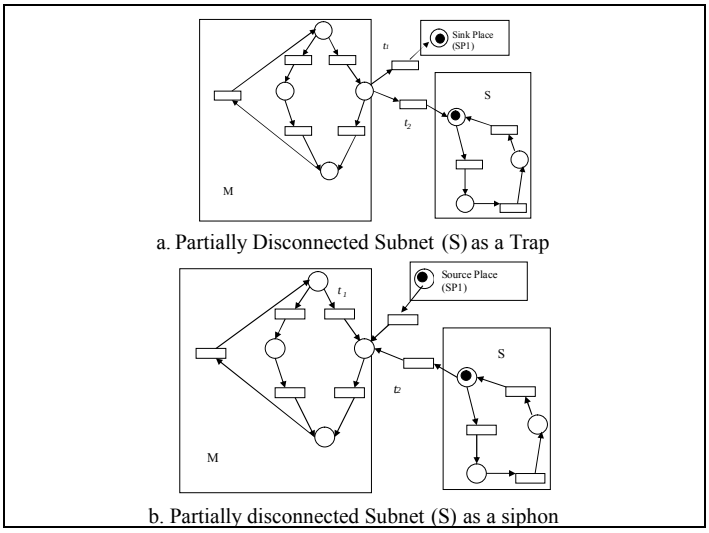
3. PETRI NETS

PNs [1, 4, 8] are bipartite graphs[1], composed of two basic components, a set of places, P, and a set of transitions, T, interconnected by directed arcs, as shown in Figure 1. The relationship between these components can be specified by two functions connecting transitions and places, I (the input function) and O (the output function). The symbol I defines, for each transition t_j, the set of input places for transition, $I(t_j)$. Similarly the symbol O defines, for each transition t_j, the set of output places for the transition, $O(t_j)$. All these four items define the structure of PN. Thus formally a PN is defined by a four-tuple PN = {P,T,I,O}. A PN is a bipartite directed graph since the arcs are directed and its nodes can be partitioned into two different sets such that each arc is directed from an element in one set (place or transition) to an element of the other set (transition or place). Readers are referred to the various references for detailed introduction to routine Petri Net topics. Since the issue of disconnected subnets is critical to our paper, we introduce this briefly.

3.1 Partially Disconnected Subnets

A set of places (and transitions) form a partially disconnected subnet if *(i)* one or more transitions connect this subnet to its complement, and *(ii)* if these transitions are exclusively output or input transitions with respect to the complement, but not both. 1 illustrates two partially disconnected subnets, one with a source place (1b) and another with a sink place (1a).

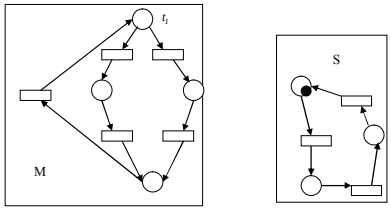
[1] While other alternate definitions exist, for example [27], the classical definition of PNs suffices for the work presented here.



1: A PN with a partially disconnected subnet S

3.2 Completely Disconnected Subnets

A set of places *P* forms a completely disconnected subnet if there are no transitions that connect the complement *P'* to *P*, where the complement *P'* is the main net. 2 illustrates a completely disconnected subnet.



2: A PN with Completely Disconnected Subnet S

3.3 PN Based Analysis Techniques

In this section all the three problems, viz. unboundedness, irreversibility, and unreachability are described, in terms of PN-based terminology. Again, for detailed introductions to these topics readers are referred to the various references[4]. We briefly identify different PN-based techniques here for completeness purposes.

3.4 Unboundedness:

A PN with m places and with an initial marking M_0 is said to be k-bounded or simply bounded if the number of tokens in each place does not exceed a finite number k for any marking reachable from the initial marking M_0, i.e., $M(P) \leq k$ for every place in the set of places P, and every marking M belonging to $R(M_0)$ of the PN, where M(P) is the total number of tokens in all places in P at marking M and $R(M_0)$ is the set of all reachable markings from marking M_0.

The widely used approach to detect unboundedness in PNs is the coverability tree approach [1, 4, 18], which is used to verify boundedness and to detect those places that eventually grow unbounded. In our work, a modified coverability tree algorithm is used to detect unboundedness, i.e., detect unbounded markings and unbounded places. The classical algorithm is extended to detect guard (entry) transitions that lead to

unboundedness.

3.5 Irreversibility:

PN irreversibility is one pointer to instability in a software system model. When the initial state of the system is reached after a sequence of successive firing vectors, then the PN is said to be reversible. In other words the PN is reversible if it is live and if all possible legal states are reachable. Different algorithms are available to detect deadlocks. These include the classical siphon and trap detection algorithm, network unfolding algorithm, and linear programming method. While the network unfolding algorithm (a partial order method) [19] can assess only the liveness of PN models, it fails to localize potential deadlocks (minimal empty siphons). The linear programming method can localize a maximal empty siphon, but cannot identify multiple subsets of empty minimal siphons within this empty maximal siphon. Since our objective is to localize the empty minimal siphons, the classical siphon and trap detection algorithm is chosen [4,5] for detecting deadlocks. We use the classical siphon and trap detection algorithm to not only assess liveness, but augment it with Thelan's prime implicant algorithm for detecting minimal siphons [20-21]. This allows the detection of all possible empty minimal siphons (potential deadlocks) of the PN model. However, after this the coverability tree algorithm is used for detecting transitions leading into these potential deadlocks. The algorithm to find minimal T-invariants and all the paths of the PN model to test is then implemented to detect partially disconnected paths and source and sink places [5].

3.6 Reachability:

The algorithm to find T-invariants and all the paths of a given PN model under test is implemented to detect completely disconnected paths. The best method to detect completely disconnected subnets is to adopt the approach discussed later in this paper (Section 4). This helps find those invariants that are not a part of the main net, i.e., which are not a part of the linearly independent paths.

4. TOOL ARCHITECTURE

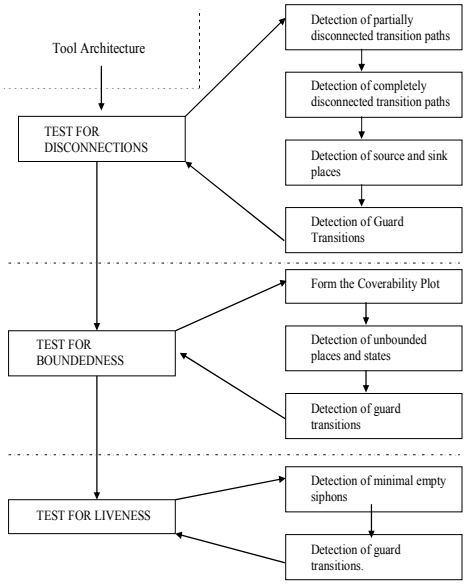
In this section, the sequence of steps taken to detect subnets (places) and transitions leading to instability are described. Before analysis begins, the following housekeeping functions need to be performed: *(i)* The given net is converted to a pure ordinary PN (without self loops - self-loops are converted into regular loops by adding additional places and transitions and all arc weights are '1', *(ii)* If the given net is a MIMO (Multi Input, Multi Output) net, then it is converted to an equivalent SISO (Single Input Single Output) net by having a common source place for all source places (or transitions), and a common sink place for all sink places (or transitions); and *(iii)* A loop back transition always connects the common sink place back to the common source place. The PN model under test, therefore, always has a loop back transition (identified by a specific attribute in the corresponding PN editor). The detection-of-disconnections module detects this "color" attribute of the loop-back transition to enumerate all the paths. If the PN model does not have a loop-back transition, then a loop-back transition is added.

Subsequently, the following steps as described by the block diagram in 3 are executed. *(i)* The given net is first tested to find all partially and completely disconnected subnets. Then those transitions that connect the main net with the partially disconnected subnets are identified. These transitions are the guard[2] transitions connecting partially disconnected subnets. *(ii)* After verifying and correcting disconnections, the PN is tested first for boundedness. *(iii)* After verifying and correcting unbounded places, the PN is then tested for liveness, note that liveness cannot be verified for unbounded PNs in general, although some recent work makes that of certain unbounded PNs possible [18].

The architecture shows the sequence of operations of the main modules with their associated sub-modules. Since the algorithm to detect partially and completely disconnected subnets is the same, a single module – Detection-of-disconnections.

4.1 Detection of Disconnections

This section describes in detail the first module of the tool developed. This module verifies if the net under test has *(i)* partially disconnected subnets, *(ii)* completely disconnected subnets, and *(iii)* source and sink places. By detecting partially disconnected subnets this tool verifies one aspect of the reversibility property of the net and by detecting completely disconnected subnets it verifies the reachability property of the net. The detection of disconnections test is the first test for a given PN system model.



3 : Stability Analysis Tool Architecture

3 shows the top-level design (block diagram) of the module. Each block has a specific set of functions associated with it. The

2 The term guard transition is used to warn the user of the tool to place monitors on these transitions that lead into disconnected parts of the PN model. This term will also be used in subsequent sections.

following is a description of each block: *(i) Detection of partially disconnected paths:* This sub-module finds those minimal T-invariants, which are not part of the paths. These invariants form the set of disconnected subnets. Those sets that have a link to the main net through a transition or a set of transitions, form the set of partially disconnected subnets. *(ii) Detection of paths and invariants of the PN model:* This sub-module first detects the minimal T-invariants of the PN model and then combines them to obtain all paths. *(iii) Detection of completely disconnected paths in the PN model:* From the set of those invariants that are not part of paths, this sub-module finds invariants that do not have any link to the main net through a transition. This set forms the set of completely disconnected subnets. *(iv) Detection of source and sink places:* This sub-module identifies those places which do not have any input arcs and those places which do not have any output arcs, thus identifying the source and sink, respectively. *(v) Detection of guard transitions:* This sub-module detects those transitions that lead into or originate from the partially disconnected subnets or source/sink places.

The following steps describe the algorithm used:

Step 1. Find the set of minimal T-invariants of the PN model. If no T-invariants are generated for the model, then it means that the net is dead. Hence, the net needs to be further tested for deadlocks, which is done in the block 'Test for Liveness'.

Step 2. Generate the firing sequence using the initial marking and the connection matrix.

Step 3. With the firing sequence, minimal T-invariants and the loop-back transition find all the paths of the PN model from the first transition to fire until the loop-back transition using the following algorithm [5]:

- ***Step 3.1.*** Find the size of the invariant matrix.
- ***Step 3.2.*** For the invariant matrix find all the permutations of the invariant rows.
- ***Step 3.3.*** For each invariant row generated from Step 3.2, check if it forms a proper path using the firing sequence matrix. The proper path is defined as the path leading from the first transition to fire to the paths leading to the loop-back transition.
- ***Step 3.4.*** The set of all the proper paths gives the paths of the PN model.

Step 4. Find the set of minimal T-invariants that are not subsets of any of the paths of the PN model. This set of minimal T-invariants gives the set of disconnections of the PN model.

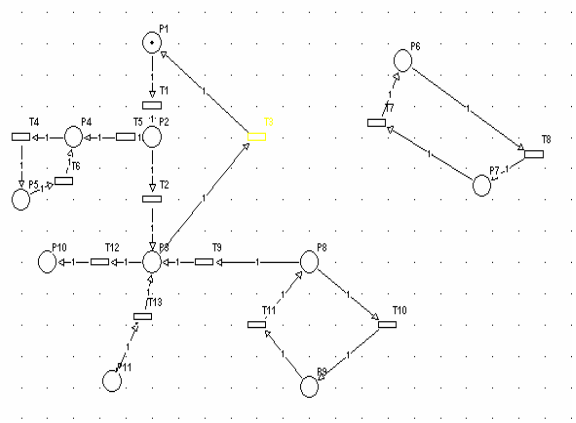
Step 5. From this disconnected set find those transitions that link these disconnections from their complement net. Those disconnections, which do not have any link, form the set of completely disconnected subnets, while those that have at least one link form the set of partially disconnected subnets.

Step 6. From the incidence matrix find those columns that do not have an element greater than or equal to 1. These rows are the set of places that do not have any input arcs. Hence this set of places form the set of source places. From the incidence matrix find those columns that do not have an element less than or equal to –1. These rows are the set of places that do not have any output arcs. Hence this set of places form the set of sink places. The set of source and sink places with the set of partially disconnected subnets form the set of partial disconnections.

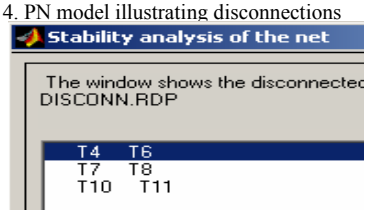
Step 7. For every place present in the set of paths of the PN model, find the input and output transitions using the incidence matrix. From this set, find that set of transitions that are not part of the minimal T-invariants and the paths. This set gives the set of transitions leading into or originating from partially disconnected subnets and/or source and sink places. This is because those transitions, which are not included in the set of minimal invariants, are not included in the set of paths either. Hence, this set of transitions does not contribute to any closed path of the PN model. Therefore the transitions of this set should be connecting to either source or sink places or partially disconnected subnets.

In the following sequel, we illustrate the detection of disconnections and the guard transition set of a PN model containing partially and completely disconnected paths and source and sink places. The guard transition set is the set of transitions connecting the partially disconnected paths and source and sink places of the PN model. 4 shows a PN model with disconnections. This model shows partially disconnected paths, completely disconnected paths, source and sink places, and the main net. From this PN model it is evident that the path p_1-t_1-p_2-t_2-p_3-t_3 forms the main net. The partially disconnected paths are p_4-t_4-p_5-t_6 and p_8-t_{10}-p_9-t_{11}. The completely disconnected path is p_6-t_8-p_7-t_7. The source place is p_{11} and the sink place is p_{10}. t_5 is the guard transition that leads into the disconnected path p_4-t_4-p_5-t_6. t_9 is the guard transition that originates from p_8-t_{10}-p_9-t_{11}. t_{12} and t_{13} connect the source and sink places respectively. The objective of this module, as discussed in the previous sections, is to detect these disconnections and the guard transitions.

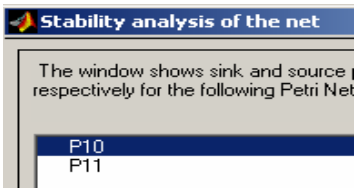
The first result the tool provides pertains to partially and completely disconnected transition paths, as shown in 5. It clearly shows the paths t_4-t_6, t_7-t_8, and t_{10}-t_{11} as disconnected. The next result that is obtained is shown in 6 which finds the source and sink places of the PN model. The result shown is places p_{10} and p_{11}. Finally, Figure **7** shows the guard transition set of the corresponding PN model. It is the transition set $\{t_5, t_9, t_{12}, t_{13}\}$.



4. PN model illustrating disconnections



5. Partially and Disconnected Paths of 4



6. Source and Sink places of 4

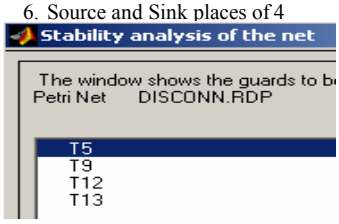


Figure 7. Guard transitions of 4

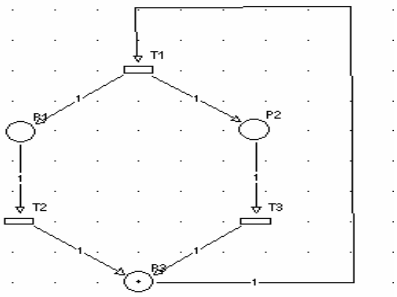
4.2 Detection of Unboundedness

This section provides a detailed description of the second module of the tool developed. It checks the PN model under test for unboundedness and, if unbounded, it identifies the places that become unbounded under successive markings. Furthermore it also identifies guard transitions that, when fired, cause unboundedness. Finally, it draws the Coverability graph of the PN model. This graph helps a user get a clear view of the unbounded states in the PN model. By detecting unbounded places this tool verifies the boundedness property of the net. But most importantly it delineates guard transitions in the PN model that need to be monitored. This is the second test that the given PN model should be subjected to before proceeding to the other tests.

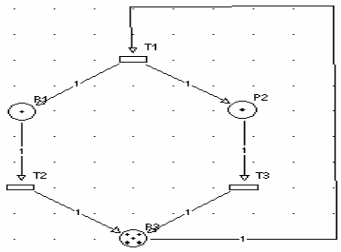
3 shows the top-level design (block diagram) of the module. Each block has a specific set of functions associated with it. The following is a description of each block:

1. Form and plot the coverability tree of the PN model: The coverability tree is formed for the PN model under test according to the algorithm in [4]. Unbounded states, dead states and repetitive states are clearly delineated in this sub-module. The output of this module is a coverability tree in matrix form. The coverability tree is also plotted in the sub-module.
2. Find the unbounded places of the PN model: The unbounded places are identified but not explicitly shown as an output in this sub-module. The coverability tree in matrix form is converted into an equivalent MATLAB plot in the form of a graph.
3. Detection of guard transitions leading to unbounded places: This sub-module finds those transitions that are most recently fired and cause the first occurrence of an unbounded state in every branch of the Coverability tree.

In the following sequel, the detection of unbounded places and the guard transition set of a PN model containing places that eventually become unbounded with successive firing of enabled transitions are illustrated. The guard transition set is the set of transitions that, when fired, causes a particular state of the PN model to become unbounded. 8 shows a unbounded PN model, which needs to be tested for unboundedness. This model shows a transition with one input and two output arcs modeling concurrency.



8: PN model to be tested for unboundedness



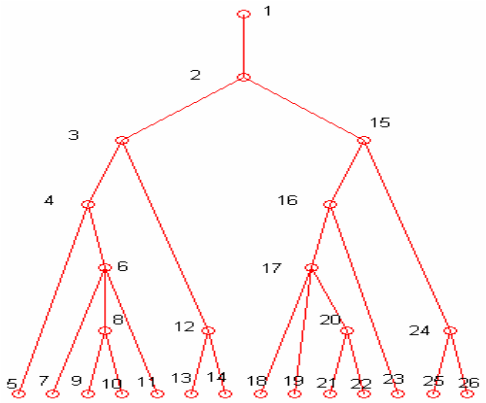
9. PN model showing unbounded place p3

This model shows the initial marking in place p_3. It also has a loop back transition t_1. This model is a pure PN model and thus is ready to be tested. It can be observed that this PN model will eventually become unbounded by accumulating tokens in place p_3. This is illustrated in 9. 10 shows the coverability tree obtained while testing this PN model for unboundedness and 11 illustrates the transition set $\{t_1\}$ identified, which leads to p_3 becoming unbounded. This alerts a designer to monitor it carefully and understand its implication on the PN design model.

4.3 Detection of Deadlocks

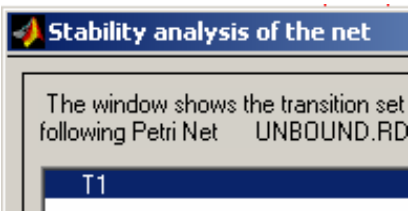
This section provides a detailed description of the third module of the tool developed. It determines if the PN model under test is prone to deadlocks and then finds the sets of places that form potential deadlocks (minimal empty siphons).

Furthermore it would also identify guard transitions that, when fired, eventually lead the system into a siphon. By detecting minimal empty siphons this tool verifies the liveness property of the net, and thus verifies the other aspect of reversibility. But most importantly it delineates guard transitions in the PN model that need to be monitored. This is the third and final test for the given PN model. 3 shows the top-level design (block diagram) of the module. Each block has a specific set of functions associated with it.



Node	Marking	Node	Marking	Node	Marking
1	001	10	0@@	19	@1@
2	110	11	1@@	20	@0@
3	0@1	12	0@@	21	@1@
4	1@0	13	@@@	22	@0@
5	0@1	14	0@@	23	@01
6	1@@	15	@01	24	@0@
7	@@@	16	@10	25	@@@
8	0@@	17	@1@	26	@0@
9	1@@	18	@@@		

10. Coverability Tree of PN model in 8



11. Guard transitions of 8

The following is a description of each block.

1. *Identify the minimal empty siphons of the PN model:* This sub-module finds the empty minimal siphons of the PN model, which are potential deadlocks. A minimal siphon is said to be empty when it does not contain any marked trap. A marked trap is a trap containing at least one token in one of its places. The presence of empty minimal siphons in the PN model indicates that the PN is not live and is prone to deadlocks.
2. *Find the coverability tree of the PN model and identify dead states:* This sub-module makes use of the coverability tree to find the set of dead nodes or dead states. The objective behind finding dead states is to detect firing vectors that lead into dead states when fired.
3. *Detection of guard transitions leading to dead states:* After identifying the dead states all the firing vectors are obtained from the root node to the terminal dead node for every branch. Furthermore, firing vectors are identified for each terminal dead node in the coverability tree.

4.3.1 *Algorithm to Detect Empty Minimal Siphons and Guard Transitions*

The algorithm that has been implemented to detect the empty minimal siphons and guard transitions leading into deadlock states of the PN model is as follows:

Step 1. Obtain the incidence matrix of the PN model.

Step 2. For every place p_i in the matrix find the set of places that output to the input transitions of p_i.

Step 3. For every place of the PN model form the set of Boolean clauses.

Step 4. From this set of Boolean clauses, form the Boolean equation of the entire PN model.

Step 5. Solve this Boolean equation by using Thelen's Prime Implicant algorithm to obtain prime implicants of the PN model. Thelen's Prime Implicant algorithm is discussed in the next section. This set of Prime Implicants gives the set of siphons of the PN model.

Step 6. From this set of siphons, find those siphons that do not form the subset of any other siphon. This set of siphons gives the set of minimal siphons.

Step 7. Repeat steps 1 through 5 by reversing the sign[3] of the incidence matrix in step 1. This gives the set of all traps of the PN model.

Step 8. From this set of traps find all those traps, which are marked, i.e., they have at least one token present in one of their places. This gives the set of marked traps.

Step 9. From the set of marked traps obtain the set of minimal marked traps.

Step 10. Find the set of minimal siphons, which do not form the superset of any of the minimal marked traps. This gives us the set of minimal empty siphons, which are potential deadlocks.

Step 11. Obtain the coverability tree of the PN model. From the coverability tree obtain the set of all terminals which are dead, i.e., no transitions are enabled after this state is node is generated.

Step 12. For every dead terminal node find the most recent split[4] along its path to the root. Find the most recent firing vector after this split along its path to the dead terminal node. The set of all these firing vectors for every dead terminal node gives the set of transitions,

[3] Reversing the sign of the incident matrix reverses the PN model, i.e., for every place the input transitions become output transitions and vice-versa. The set of siphons of this new PN model is apparently the set of traps for the original PN model

[4] A split in the Coverability tree is that node which is a parent node to at least two child nodes.

which need to be monitored in the PN model. This set of transitions is the set of guard transitions.

4.3.2 Applying Thelen's Prime Implicant Algorithm

In this section the application of Thelen's Prime Implicant algorithm [20-21] is discussed. A prime implicant is a product of variables or their components. For instance, if a, b, c, and d are variables, then one of the possible prime implicant could be ab'cd '. If a prime implicant has value 1, then the boolean function has a value 1. Thelen's prime implicant algorithm (simpler to implement and efficient in time and memory) can be used for the following purpose for a product-of-sums Boolean equation: (i) complementation, (ii) detection of essential primes, (iii) computational of a minimal cover, and (iv) reduction of prime implicants. The problem of finding prime implicants can be defined as follows. Given a conjunctive normal form of a boolean equation defined by

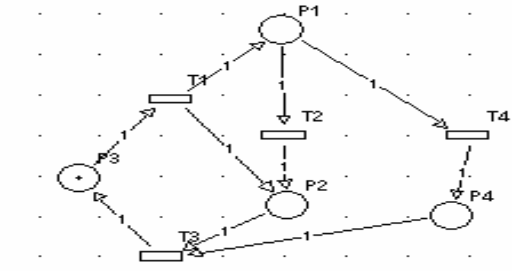
$$F = (a_1 + a_2 + a_3 + \ldots..)(b_1 + b_2 + b_3 + \ldots..) \quad (1)$$

The objective is to use Thelen's algorithm to convert F into the sum of its prime implicants. The Boolean function F is expanded into a disjunctive sum-of-products form by multiplying out the disjunctions of F and deleting products that subsumes others. The reader is referred to available literature for details of this algorithm [20-21].

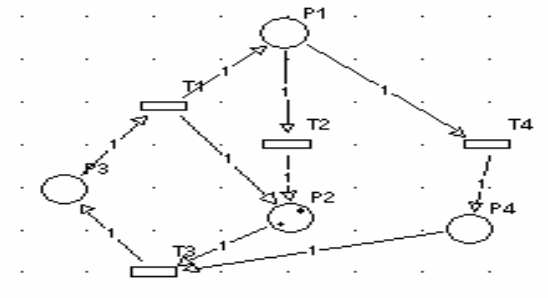
A PN model to be tested for liveness is shown in 12. This model shows the initial marking in place p_3. After successive firings, the PN model would eventually become deadlocked as illustrated in 13. The objective is to detect empty minimal siphons and the guard transition set of a PN model that when fired will eventually lead to such a deadlock state. Here it is observed that no transitions are enabled once all the tokens are present in p_2. Applying equation (1) to this net we get the following:

$$F = (p_1' + p_3)\,(p_2' + p_3)\,(p_2' + p_1)\,(p_2 + p_3' + p_4)\,(p_1 + p_4') \quad (2)$$

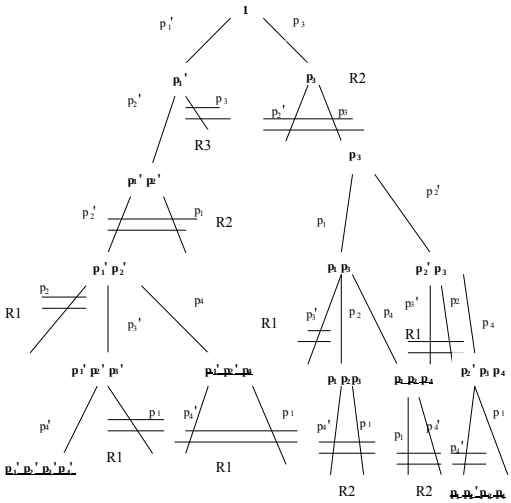
This equation can then be applied to obtain Thelen's prime implicant tree as shown in 14. This tree gives us the minimal siphons and these are the underlined nodes. We then eliminate marked traps from this set of siphons to obtain the empty minimal siphons.



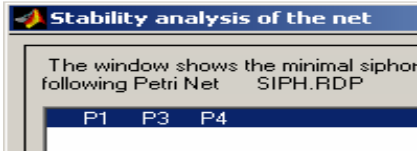
12. PN model to be tested for liveness



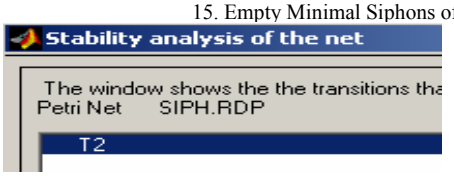
13. PN model (from 12) in a deadlocked state



14. Thelen's tree for Equation 8



15. Empty Minimal Siphons of 12



16. Guard transitions of 12

When the user decides to find potential deadlocks (minimal siphons) in the net and the corresponding transitions causing them, the result is shown in 15. It shows one empty minimal siphon, which is a potential deadlock. The next result that is obtained is shown in 16, listing transitions leading to dead states for the model in 12. The result shown is the transition t_2. These are the guard transitions – those when fired will lead to a deadlock state.

5. CASE STUDY

This section provides a case study of a hypothetical electronic manufacture and assembly plant and the software system used for the manufacturing control process, modeled as a DEDS. The plant comprises of two separate primary processes *(i)* board

etching and drilling, and *(ii)* IC layout and soldering. Let us assume that there are two machines, A and B operating in tandem. Machine A has specially equipped tools to do the board etching and IC layout operations in parallel, and machine B has specially equipped tools to do drilling and soldering operations in parallel. 17 shows the first cut PN model of automated operations that happens in this DEDS. The presence of tokens in places p3 and p4 implies that machines A and B are not in operation respectively, and are ready to work in tandem.

Place p6 models the initialization operation. In other words when transition t7 fires the machines are available to be operated. Thus after t7 fires, tokens are present in places p3 and p4 and the two machines are ready. The system starts with the firing of transition t3 that activates machine A to perform the board etching operation. Place p2 models the board etching operation. The firing of transition t2 depends upon the availability of machine B for the drilling operation. Transition t2 fires and places token in place p1 that models the drilling operation. The path t11-p9-t12-p10-t13 models the condition of rejection and reprocessing of badly drilled and etched boards. The firing of t1 places tokens in places p3 and p4 indicating the availability of machine A and B for further operations. The firing of t6 makes machine B perform the IC layout operations on already drilled and etched boards. Place p5 models this operation. Enabling of t4 depends upon the availability of machine A for the soldering operation. Transition t4 fires indicating the end of IC layout and soldering operations. There are two outgoing arcs from transition t4 to places p13 and p14. This is to model the parallel processing of two boards at one time. In other words the plant is capable of soldering and laying out ICs on two separate boards at one time. Place p11 models the buffer zone where the finished boards are placed. The firing of transition t5 indicates the removal of one finished board at a time and resetting machines A and B for further operation. The loop p7-t10-p8-t9 models a special automatically activated interrupt which automatically resets all the machines when a major problem is encountered.

5.1 Design Flaws

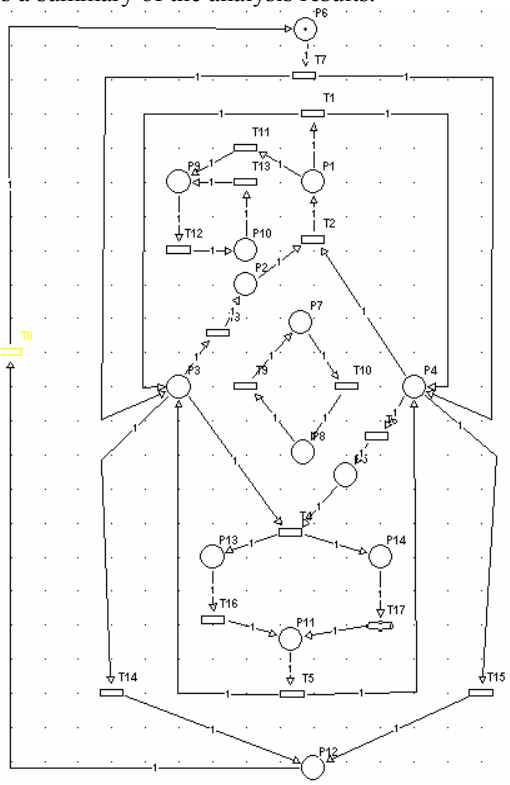
This plant is prone to all the three major design flaws of unreachability, unboundedness and irreversibility. If we take a closer look at the equivalent PN model we would see that the firing of transition t2 is contingent on the availability of both machines for the drilling and etching operations, while the firing of transition t4 is contingent on the availability of both machines for the IC layout and soldering operations. However this dependency leads to a circular waiting condition, wherein the two parallel processes wait for each other to release the machines and eventually end up in a circular deadlock. This models one aspect of irreversibility viz. deadlocks. Furthermore the path t11-p9-t12-p10-t13 modeling the rejection and reprocessing of badly drilled and etched boards has a way in but no way out. In other words once the boards are rejected or reprocessed, the control does not lead back to the main process.

Once tokens enter this loop they stay in because of the absence of any valid return path. This models the other aspect of irreversibility, viz. partially disconnected subnets. Path p7-t10-p8-t9, although vital, does not have any appropriate call or return. In other words this interrupt is never activated when the machines undergo a major operational failure. This models unreachability. The parallel operations of IC layout and soldering on two boards at a time (places p13 and p14) and the removal of one board at a time (one token at a time from place p11) indicates the place p11 eventually becomes unbounded when its capacity to hold tokens increases out of bounds. This models unboundedness.

5.2 Usage of the Analysis Tool

This section illustrates the detection of all unstable conditions of the plant in terms of places and transition in the equivalent PN model. 19 models the detection of the partially and completely disconnected subnets in terms of paths p9-p10 and p7-p8 respectively. This is the first output obtained from the detection of disconnections module.

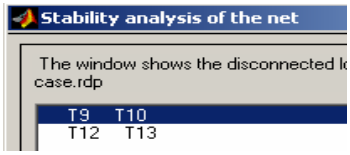
20 indicates that transition t11 needs to be monitored. In other words guards need to be placed on this transition to prevent it from firing, else it would lead into an unstable condition. Figure 21 shows that place p11 eventually becomes unbounded and 22 shows that guards need to be placed on transition t4 which is responsible for this unstable operation. 23 shows the set of places p1-p3-p4-p6-p11-p12 which forms an empty minimal siphon and is responsible for causing the potential deadlock. 24 shows that when transitions t3 and t6 are fired, the net enters into a dead state because the empty minimal siphon would not allow any token to enter it. Hence guards need to be placed on these transitions to prevent them from firing. 25 provides a summary of the analysis results.



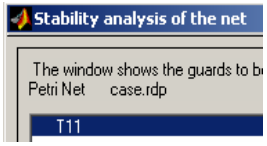
17. First Cut PN model of a hypothetical assembly plant

PLACES	TRANSITIONS
p1: Status of board drilling process p2: Status of board etching process p3: Machine A available to work p4: Machine B available to work p5: Status of IC layout operation p6: Initialize the assembly plant p7: Status of machine condition p8: Status of interrupt routine p9: Status of drill and etch correction p10: Status of drill and etch check on boards p11: Buffer for removal of finished boards p12: Status of process completion p13: Status of soldering process p14: Status of soldering process	t1: Stop machines A and B t2: If etching process is complete and machine B is ready, initiate drilling process t3: Initiate board etching process t4: If IC layout operation is complete and machine A is ready, initiate soldering process t5: Stop machines A and B t6: Initiate IC layout process t7: Prepare machines A and B for operation t8: Initiate board etching process t9: Activate interrupt routine t10: Check machine condition t11: Activate drill and etch correction t12: Activate drill and etch check t13: Activate drill and etch correction t14: Indicate machine A has completed operation t15: Indicate machine B has completed operation t16: Deposit finished boards in buffer t17: Deposit finished boards in buffer t18: Loop-back transition

18. Places and Transition Descriptions



19. Partially and completely disconnected subnet of Figure **26**



20. Guard transition into subnet p9-p10.

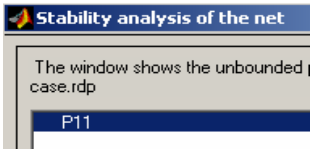
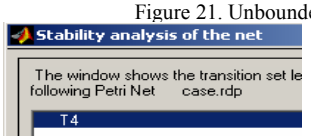
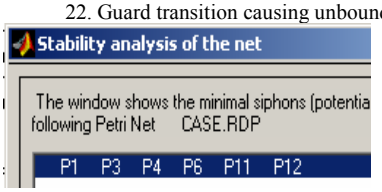


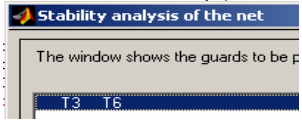
Figure 21. Unbounded place in Figure **26**



22. Guard transition causing unboundedness in Figure **26**



23. Empty minimal siphon of Figure **26**



24. Guard transition causing potential deadlock in Figure **26**

PN model	*Partially disconnected subnets*	*Completely disconnected Subnets*	*Guard Transitions*
Case.rdp	t9-t10	t12-t13	t11

Results obtained from the detection of disconnections module

PN model	*Unbounded places*	*Guard Transitions*
Case.rdp	p11	t4

Results obtained from the detection of unboundedness module

PN Model	*Empty minimal siphons (potential deadlocks)*	*Guard Transitions*
Case.rdp	p1-p3-p4-p6-p11-p12	t3-t6

Results obtained from the detection of deadlocks module

25. Summary of Analysis for Figure **26**

5.3 Stable Design Model

Figure 26 shows the stable PN model of the assembly plant. Transition t17 forms the return path of the rejection and reprocessing loop. Transitions t18 and t19 connect to the automatically activated interrupt routine when either of the machines goes out of order. Transition t20 forms the return path of this routine to the exit place p11. The deadlock is eliminated by forcing either transition t13 or t20 to fire first, thus eliminating a circular wait condition, however at the expense of sequential processing instead of concurrency. The weight of the arc connecting p10 and t5 is '2' indicating that two boards are removed at a time. The stable model is then tested using the stability analysis tool and the results do not list any disconnections, unbounded places and deadlocks.

The above illustrates that the stability analysis tool can be used to study the stability of a PN design model, helping identify unstable areas in the plant using the PN properties of reversibility, boundedness and reachability. These problems that cause major software issues if not properly implemented, are easily identified and rectified using a PN design model.

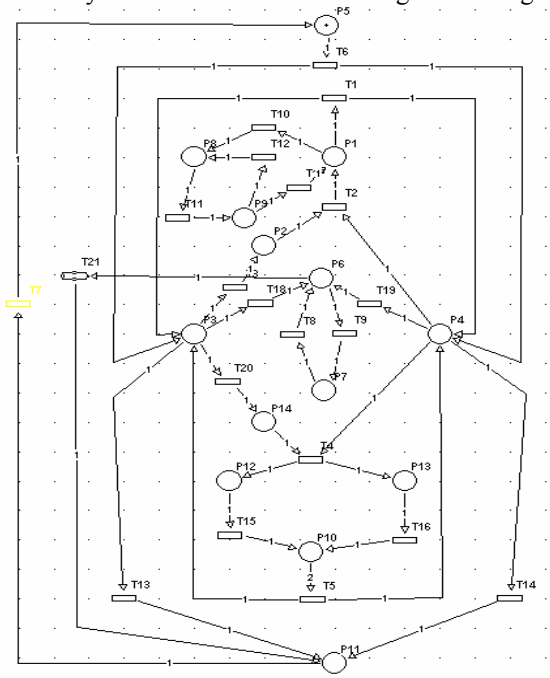


Figure **26**. Stable Assembly Plant Model

PLACES	TRANSITIONS
p1: Status of board drilling process	t1: Stop machines A and B
p2: Status of board etching process	t2: If etching process is complete and machine B is ready, initiate drilling process
p3: Machine A available to work	t3: Initiate board etching process
p4: Machine B available to work	t4: If IC layout operation is complete and machine A is ready, initiate soldering process
p5: Initialize the assembly plant	t5: Stop machines A and B
p6: Status of machine condition	t6: Prepare machines A and B
p7: Status of interrupt routine	t7: Loop-back transition
p8: Status of drill and etch correction	t8: Activate interrupt routine
p9: Status of drill and etch check on boards	t9: Check machine condition
p10: Buffer for removal of boards	t10: Activate drill and etch correction
p11: Status of process completion	t11: Activate drill and etch check
p12: Status of soldering process	t12: Activate drill and etch correction
p13: Status of soldering process	t13: Indicate machine A has completed operation
p14: Status of IC layout operation	t14: Indicate machine B has completed operation
	t15: Deposit finished boards in buffer
	t16: Deposit finished boards in buffer
	t17: Drill the board again
	t18: Check condition of machine A
	t19:Check condition of machine B
	t20: Initiate IC layout process
	t21: Stop/Reset Machines A and B

Figure **27**. Description of Places and Transitions

6. Conclusions and Future Work

The primary objective of our work is to develop a usable model driven design methodology for stable software system design, build an PN based application tool and illustrate its use through a simple case study. A PN-based analysis tool, written within MATLAB, is shown to identify those places/transitions of an ordinary PN model which lead to software stability problems. Subsequent analysis helps to delineate those points (transitions) in the software system that needs to be monitored to prevent unstable operation. These points are represented by guard transitions and may be responsible for problems that cause irreversibility, unboundedness, and unreachability.

As of now the developed tool requires a PN editor to develop the PN system model and write a corresponding text description of the model. However, additional generic features such as a code parsing can be added to the tool, which could read a software code written in C, C++, Java, etc., and translate it into an equivalent PN model directly without the user actually drawing the model by hand. In addition to the issues presented in this paper, controllability and observability issues have to be further analyzed for the development of a completely stable software system.

A majority of research has only been done in the area of deadlock avoidance. Specific policies have been designed in the controller of flexible manufacturing systems, a special class of discrete event dynamic systems, which can help prevent the system from entering a deadlocked state. Fanti in [24] uses digraph theoretic concepts to derive necessary and sufficient conditions for a deadlock occurrence. The plant is modeled as a DEDS. Specific restriction policies are designed, which use the state knowledge to avoid deadlocks by inhibiting or enabling some transitions. The restriction policies involve small on-line computation costs, and thus are suitable for real-time implementation. A supervisory control mechanism is introduced in [25] for handling the problem when not all of the siphons in a given PN are controllable. The method involves adding a place for every uncontrolled siphon to make them controllable. These controller places act to restrict behavior in the original plant that would lead to deadlocks, thus playing the part of a supervisory controller. Thus research can be done in integrating the work done in this paper with deadlock avoidance mechanisms. Furthermore, it is also observed that research in identifying unboundedness and using it in software systems design has been minimal. Recently, some researchers have reported on measures [26] that have been taken to halt the system execution by handling exceptions due to unboundedness when they arise.

Acknowledgements

The authors would like to thank Dr. Joe Anderson, Dr. Jeff Austen and Kannappan Rajamani from the ECE department of Tennessee Technological University for their help during tool development in MATLAB. A special thanks to several people on the PN mailing list http://www.daimi.au.dk/PetriNets/pnml/ including Andrei Karatkevich and Bernd Baumgarten for their valuable implementation related suggestions. The first author would like to thank Dr. Ligou Yu, Mr. Eric Brown (TTU) and Dr. MengChu Zhou (from NJIT) for several specific suggestions for improvement of this paper.

References

[1] J. L. Peterson, "Petri Nets," *Computing Surveys*, Vol.9, No.3, September 1977, pp. 223-252.

[2] M. E. Fayad and A. Altman, "An Introduction to Software Stability," *Comm. of the ACM*, Vol.44, No.9, September 2001, pp. 95-98.

[3] A. Perkusich and C.A. Jorge, "G-Nets: A Petri Net Based Approach for Logical and Timing Analysis of Complex Software Systems," *Journal of Systems Software*, Vol. 39, No. 4, October, 1997, pp. 39-59.

[4] T. Murata, "Petri Nets: Properties, Analysis and applications," *Proceedings of the IEEE*, Vol. 77, No. 4, March 1989, pp. 541-580.

[5] R. Neelakantan, "*Information sharing and testing using Petri nets for interactive and co-operative behavior*," M.S. Thesis, Tenn. Tech. University, August 2002.

[6] F. Chu and X.L. Xie, "Deadlock Analysis of Petri Nets using Siphons and Mathematical Programming," *IEEE Transactions on Robotics and Automation*, Vol.13., No.6, December 1997, pp. 793-804.

[7] S. Tanimoto, M. Yamauchi and T. Watanabe, " Finding Minimal Siphons in General Petri Nets," *IEICE Trans. Fundamentals*, Vol-E79A No.11, November 1996, pp. 1817-1824.

[8] A. E. Kostin and S. A Tchoudaikina, "Yet Another Reachability Algorithm for Petri Nets," *SIGACT News*, Vol. 29, No. 4, Dec. 1998, pp. 98-110.

[9] G. Stemersch, "*Supervision of PetriNets*," Klewer, 2001.

[10] J. O. Moody and P. J. Antsaklis, "*Deadlock Avoidance in Petri Nets with uncontrollable transitions*," Technical report of the ISIS-97-016 group, University of Notre Dame, October 1997, http://citeseer.nj.nec.com/cache/papers/cs/5710/http:zSzzSzwww.nd.eduzSz~isiszSztechreportszSzisis-97-016.pdf/moody97deadlock.pdf

[11] M. Webster, "Webster Dictionary Online," http://www.m-w.com/home.htm

[12] X. D. Koutsoukos, K. X. He, M. D. Lemmon and P. J. Antsaklis, "Timed Petri Nets in Hybrid Systems: Stability and Supervisory Control," in *Discrete Event Dynamic Systems: Theory and Applications, Special Issue on Hybrid Systems* (Eds. Panos Antsaklis and Michael Lemmon), Volume 8, No. 2, June 1998.

[13] G. Juhás, and M. Kocian, "Generalized T-Invariants of Petri Nets and Control of DEDS", *1st IFAC Workshop on New Trends in Design of Control Systems 1994*, Slovakia, pp. 408-413.

[14] L. Portinale, Exploiting T-invariant Analysis in Diagnostic Reasoning on A Petri Net Model, *LNCS 691*, Springer-Verlag, 1993, pp. 339-356.

[15] P. King and R. Pooley, "Using UML to derive stochastic Petri nets models," *Proceedings of the Fifteenth Annual UK Performance*

Engineering Workshop, pages 45-56. Department of Computer Science, University of Bristol, July 1999.

[16] A. Aghasaryan, C. Jard and J. Thomas, "UML Specification of a Generic Model for Fault Diagnosis of Telecommunication Networks", *Proceedings of Telecommunications and Networking - ICT 2004: 11th International Conference on Telecommunications*, Fortaleza, Brazil, August 1-6, 2004.

[17] J. P. López-Grao, J. Merseguer, and J. Campos, "From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering," *Proceedings of the Fourth Internl. Workshop on Software and Performance (WOSP'04)*, pp. 25-36, California, USA, January 2004.

[18] F. Y. Wang., Y. Gao, and M. C. Zhou, "A Modified Reachability Tree Approach to Analysis of Unbounded Petri Nets," *IEEE Trans. on Systems, Man, and Cybernetics: Part B*, Vol 34, No. 1, 303-308, Feb. 2004.

[19] K. X. He and M. D. Lemmon, "Liveness-enforcing of Bounded Ordinary Petri Nets using Partial Order Methods," *IEEE Transactions on Automatic Control*, July 2002, Vol. 47, pp. 1042-1055.

[20] B. Thelen, *"Investigations of Algorithms for Computer-aided Logic Design of Digital Circuits*, (in German)," PhD dissertation in Computer Science, ITIV, Univ. of Karlsruhe, Germany, 1988.

[21] H.J. Mathony, "Universal logic design algorithm and its application for the synthesis of two-level switching circuits," *IEEE Proceedings*, 1989, Vol. 136, Pt.E, No. 3, pp. 171-177.

[22] M. Svadova and Z. Hanzalek, "*MATLAB Toolbox for PetriNets. Graph of Reachable Markings*," Center for Applied Cybernetics, Dept. of Control Engineering, Czech Technological University, dce.felk.cvut.cz/cak/Publikace/2001/Svadova01_MatlabTolbox_Miskolc.pdf

[23] A.A. Khan and H. Singh, "Petri Net approach to Enumerate all Simple Paths in a Graph," *Electronic Letters*, 10th April 1980, Vol. 16. No. 8 pp. 291-293.

[24] M. P. Fanti, "Event-Based Feedback Control for Deadlock Avoidance in Flexible Production Systems," *IEEE Trans on Robotics and Automation*, June 1997, Vol. 13, No. 3, pp. 347-364.

[25] J. O. Moody. *"Petri Net Supervisors for Discrete Event Systems"*, PhD thesis, Dept of Elec. Engg, Univ. of Notre Dame, Notre Dame, IN, 1997.

[26] V. Khomenko and M. Koutny, "*Verification of Bounded Petri Nets Using Integer Programming*," Technical Report CS-TR-711, Department of Computing Science, University of Newcastle (2000).

[27] Zhou, M. C. and K. Venkatesh, "*Modeling, Simulation and Control of Flexible Manufacturing Systems: A Petri Net Approach*" World Scientific, Singapore, 1998.

Dr. Ramaswamy is currently the chairperson of department of Computer Science at University of Arkansas at Little Rock. Earlier he was the Chairperson of the Computer Science Department at Tennessee Tech University. His research interests are on intelligent and flexible control systems, behavior modeling, analysis and simulation, software stability and scalability; particularly in the design and development of better software systems for real-time control issues in manufacturing and other distributed, real-time applications. He has actively consulted on the design, development and implementation of a knowledge capture, classification and mining project with eFutureKnowledge, Inc. and on a NIST sponsored ATP project with InRAD, LLC. During the summers of 2003 and 2004, he was a visiting research professor of Computer Science in the Institute of Software Integrated Systems (ISIS) at Vanderbilt University as part of a NSF ITR project - Foundations of Hybrid and Embedded Software Systems. In 1994-1995, and subsequently during the summer months of 1996 and 1998, he was a post-doctoral research fellow / visiting scientist in the Laboratory for Intelligent Processes and Systems (LIPS) at the University of Texas at Austin where he helped with research efforts on Sensible Agents. Dr. Ramaswamy earned his Ph.D. degree in Computer Science in 1994 from the Center for Advanced Computer Studies (CACS) at the University of Southwestern Louisiana, Lafayette, LA, USA (now University of Louisiana at Lafayette). He is member of the ACM, Society for Computer Simulation International, Computing Professionals for Social Responsibility and a senior member of the IEEE.

Mr. Gopal Yamijala is currently working as a Senior Software Consultant for Ingram Micro in B2B ecommerce applications. He is employed by Tallan Inc. based in Orange County, CA. His expertise includes: Tibco,J2EE, Java frond end, application servers such as WebSphere etc. His work includes building and automating B2B software to exchange RosettaNet/cXML/SOAP XML messages with trading partners such as Wal-Mart/Dell/Intel etc. in the supply chain industry. Gopal has a undergraduate degree (B.E) in Electrical Engineering from Government College of Engineering Pune, India and a M.S degree in Electrical Engineering from Tennessee Technological University at Cookeville, TN, USA.

Mr. Rajaram Neelakantan is currently working as an embedded software engineer (Firmware Engineer) at Stanley Assembly Technologies, Warren, MI. His work involves the design and programming of embedded systems. His interests include RTOS, Embedded DSP algorithms, Digital Logic Design and Programming, OO Design and Programming and Software Engineering. Rajaram has a B.S degree in Electronics and Communication from Barathidasan University, India and a M.S. Degree in Electrical Engineering from Tennessee Technological University at Cookeville, TN, USA.

Dr. P. K. Rajan received his Ph.D. in Electrical Engineering from the Indian Institute of Technology, Madras, 1975. He is currently the chairman of the ECE department at Tennessee Technological University at Cookeville, TN. He has been a summer research fellow at the NASA Johnson Space Center at Houston, TX, and the Wright-Patterson Air Force Base in Ohio. His research interests are in Circuits and Signals, Digital Signal processing, Digital Image Processing, Pattern Recognition and Optical Signal Processing. He is a fellow of the IEEE.