[32] Functional and Performance Requirements Specification for the Earth Observing System Data and Information System (EOSDIS) Core System. Revision A and CH-01.

[33] Flight Operations Segment (FOS) Requirements Specification for the ECS Project, Volume 1: General Requirements, November 1994. By Hughes Applied Information Systems.

[34] K. Lateef, H.H. Ammar, V. Mogulothu, T. Nikzadeh, “A Methodology for Verification and Analysis of Parallel and Distributed Systems Requirement Specifications”, in Proceedings of the 2nd IFIP International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE-97), IEEE Computer Society, May 1997.

[35] Jensen K., “Coloured Petri Nets: basic concepts, analysis methods and practical use”, Springer-Verlag, Berlin; New York April 1992.

[36] Hatley Derek J, Pirbhai Imtiaz A, “Strategies for real-time system specification”, Dorset House Pub, New York, NY87

9 References

[1] Maier M.W., “Integrated Modeling: A unified Approach to system Engineering”, The Journal of systems and software: Vol 32, pp101-119, 1996.

[2] Amoroso E. G., “Creating formal specifications from informal requirement documents”, ACM SIGSOFT, Software Engineering Notes, Vol 20 no 1, pp 67-70, Jan 1995.

[3] Cooke D., et al., “Languages for the specification of the software”, The Journal of systems and software: Vol 32, pp269-308, 1996.

[4] Murata T., Notomi M., “Hierarchical Reachability Graph of Bounded Petri Nets for Concurrent-Software Analysis”, IEEE transactions on Software Engineering, pp 325-336, Vol. 20, No. 5, May 1994.

[5] Fraser, M.D. & Kumar, K. “Informal to formal requirement specification languages: Bridging the gap”, IEEE transactions on Software Engineering, pp 454-, Vol. 17, No. 5, May 1991.

[6] Pezze M., Elmstrom R., Lintulampi R., “Giving Semantics to SA/RT by means of High-Level timed Petri Nets”, The international journal of time critical computing systems, Vol. 5, no 2/3, May 1993

[7] N. Dershowitz, “Program abstraction and instantiation”, ACM Trans. Program. Languages and Syst., pp 446-477, Vol. 7, No. 3, October 1985.

[8] “Automatic translation of SA/RT to high level time Petri Nets” Espirit report, IPTES-PDM-17-V2.3. 1994

[9] Nissen H., et al “Managing Multiple Requirements Perspectives with Meta-models,” IEEE Software, Vol. 13, No. 2, March, 1996

[10] J. C. Munsen and T. M. Khoshgoftaar: The Detection of Fault-Prone Programs. IEEE Trans. on Software Engineering, Vol. 18, No. 5, pp.423-433, 1992

[11] R. W. Selby and V. R. Basili: Analyzing Error Prone System Structure. IEEE Trans. on Software Engineering, Vol. 17, No. 2, pp. 141-152, 1991

[12] M. Z. Wayne and D. M. Zage: Evaluating Design Metrics on Large Scale Software. IEEE Software, Vol. 10, No. 7, pp. 75-81, Jul. 1993

[13] B. A. Kitchenham and L. Pickard: Towards a constructive quality model. Software Engineering Journal, Vol. 2, No. 7, S. 114-126, Jul. 1987

[14] J. C. Munson and T. M. Khoshgoftaar: Software Metrics For Reliability Assessment. IEEE Computer Society Press, Handbook of Software Reliability Engineering, McGraw-Hill, 1995

[15] J. C. Munsen and T. M. Khoshgoftaar: The Dimensionality of Program Complexity. Proceedings of the 11th Annual Conference on Software Engineering, Pittsburgh, May 1989, pp. 245-253

[16] T. J. McCabe: A Complexity Metrics. IEEE Trans. On Software Engineering, Vol. 2, no. 4, Dec. pp. 308-320

[17] G. A. F. Seber: Multivariate Observations, John Wiley & Sons, NY, 1984

[18] J. C. Munson and T. M. Khoshgoftaar: Applications of A Relative Complexity Metric For Software Project Management. Journal of Systems and Software, Vol. 12, no. 3, pp. 283-291

[19] T. M. Khoshgoftaar and J. C. Munson and D. L. Lanning: Dynamic System Complexity. Proceedings of IEEE-CS International Software Metrics Symposium, Baltimore, MD., May, 1993, pp. 129-140

[20] Department of Defense (DoD): Procedures of performing a failure mode, effects and criticality analysis. DoD, MIL_-STD_1629A.

[21] H. Kumamoto and E. J. Henley: Probabilistic Risk Assessment for Engineers and Scientists, second edition. IEEE Press,1996.

[22] F. Belli and J. Dreyer: Systems Specification, Analysis, and Validation by mean of Timed Predicated/Transition Nets and Logic Programming. IEEE, 1995. pp. 68-77

[23] M. H. Halstead: Elements of Software Science. Elsevier North-Holland, NY., 1977

[24] S. H. Kan: Metrics and Models in Software Quality Engineering. Addison Wesley, MA., 1995.

[25] C. Ebert: Evaluation and Application of Complexity-Based Criticality Models. Proceedings of the third international software metrics symposium, Berlin, March 1996, pp. 174-184

[26] T. M. Khoshgoftaar and J. C. Munson: Predicting Software Development Errors Using Software Complexity Metrics. Software Reliability and Testing, IEEE Computer Society Press, 1995. Pp. 20-28

[27] D. I. Heimann: Using Complexity-Tracking In Software Development. Proceedings of Annual Reliability and Maintainability Symposium, IEEE 1995. Pp. 433-437

[28] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan and N. Goel: Early Quality Prediction A Case Study in Telecommunications. IEEE Software, 1996, pp. 65-71

[29] N. Leveson and J. L. Stolzy: Safety Analysis Using Petri Nets. IEEE Trans. On Software Engineering, Vol. SE-13, No. 3, March 1987, pp. 386-397

[30] T. Nikzadeh: Risk Assessment and Complexity Analysis of Software Systems Using Coloured Petri Nets. Master thesis, Electrical and Computer Engineering Dept., West Virginia University, August 1997.

[31] B. Mikolajczak and J. Rumbut: A Systematic Method of Object-Oriented Software Design using Colored Petri Nets, Naval Underwater Warfare center; 14th International Conference on Applications and Theory of Petri Nets, Chicago, June 1993, pp. 21-25.

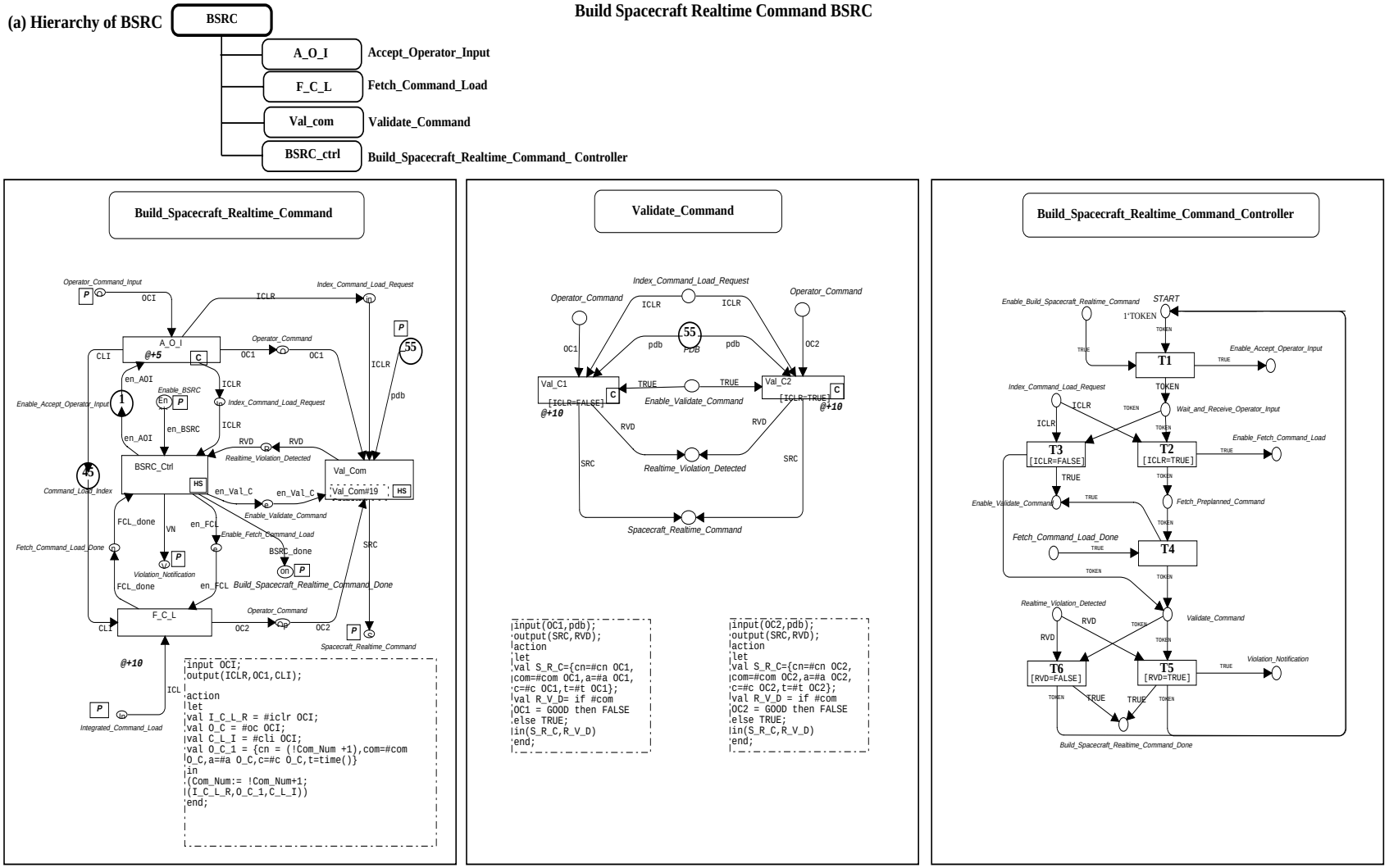


FIGURE 17: Build_Spcecraft_Realtime_Command component of EOC Commanding

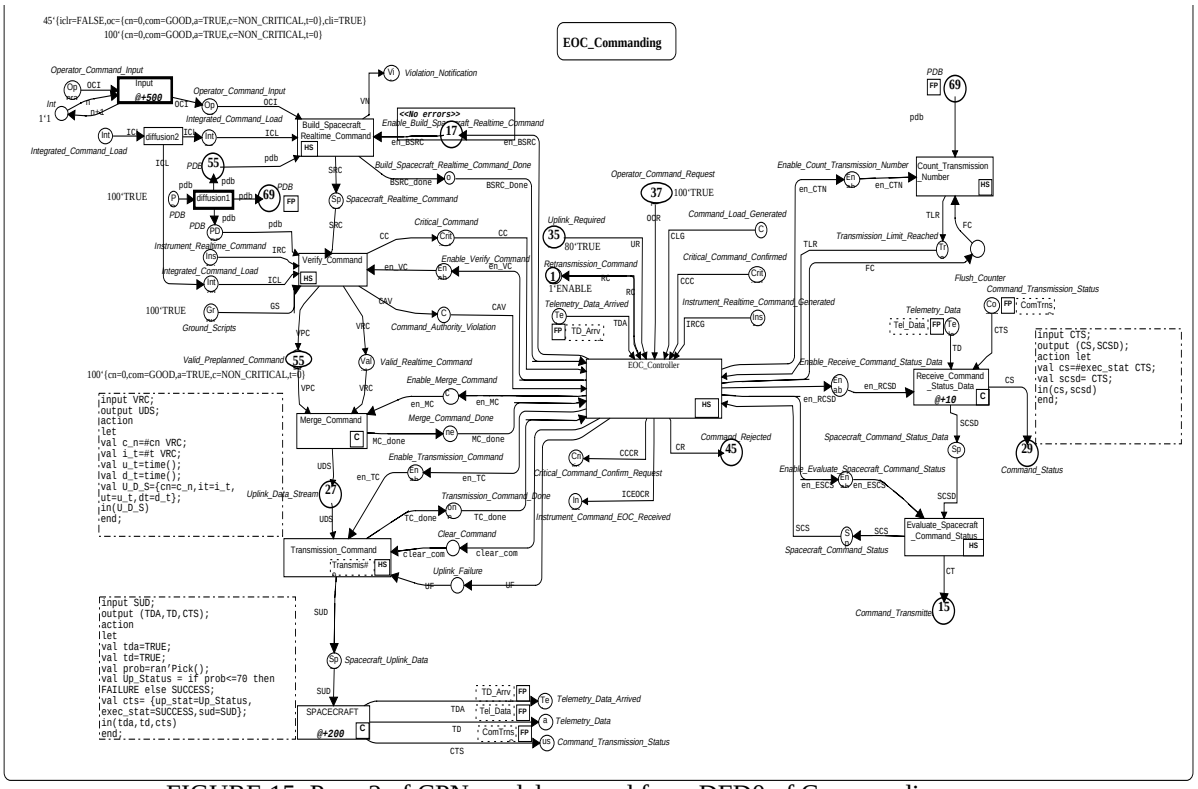


FIGURE 15: Page 2 of CPN model mapped from DFD0 of Commanding

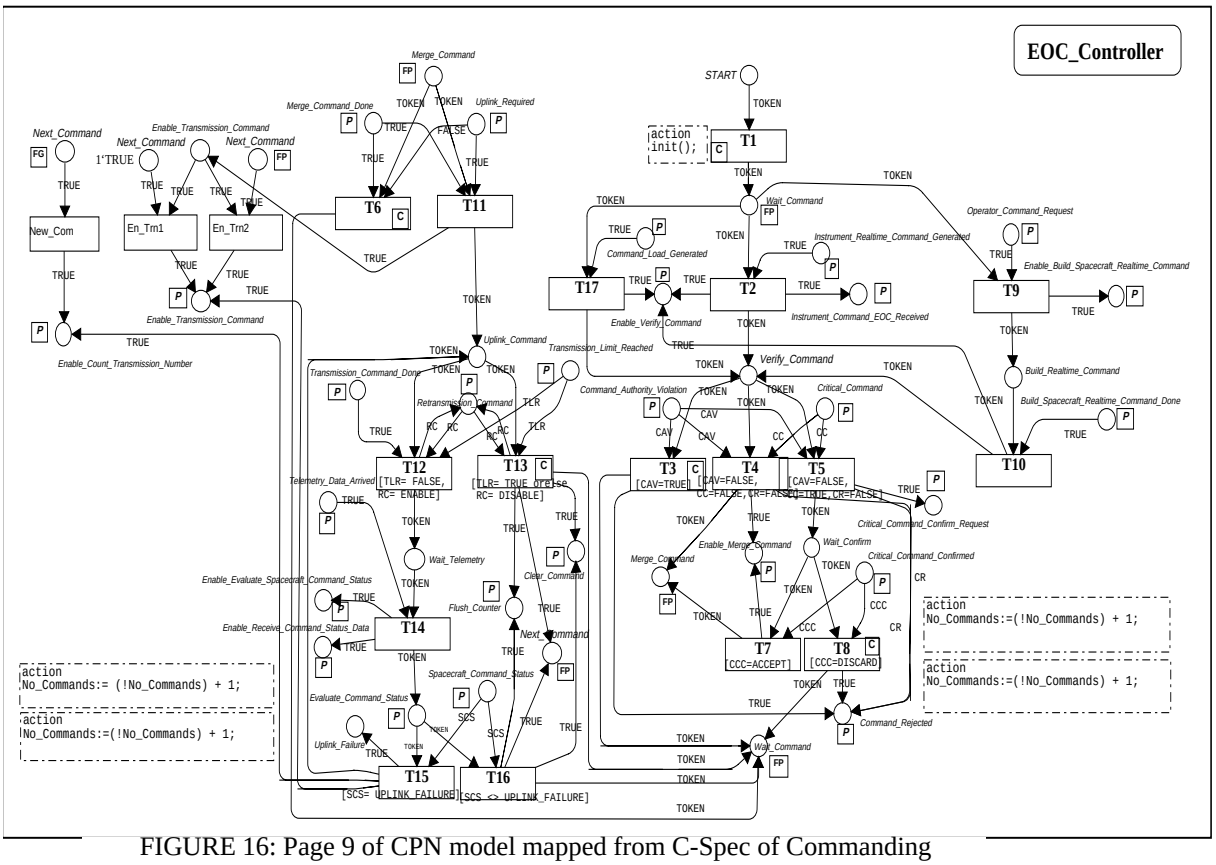


FIGURE 16: Page 9 of CPN model mapped from C-Spec of Commanding

CPN model of BSRC. Then the heuristic risk factor is calculated as shown:

Table 3: Heuristic risk factor calculations

Module	cpx*	cpx^**	svrty	hrf
AOI	10	0.33	0.75	0.33*0.75=.2475
FCL	30	1.00	0.25	1.00*0.25=.2500
VC1	20	0.67	0.50	0.67*0.50=.3350
VC2	20	0.67	0.50	0.67*0.50=.3350

Therefor VC1 - (1-0.3350)(1-0.3350) = 0.5578

and

HRF_{BSRC} = 1- (1- hrf_{AOI})(1- hrf_{FCL})(1- hrf_{VC}) = 0. 75

In Table 3, cpx* and cpx^** refer to the complexity level and the normalized complexity level of the primitive components of BSRC. The overall risk factor of BSRC is obtained by ORing the risk factors of individual subcomponents of BSRC. The moderate high value of HRF of 0.75 for BSRC can be interpreted as a reflection of the high complexity level of FCL, the high severity of AOI, and the risk factor of VC.

8 Conclusions

This paper presented a methodology for generating formal specification models based on CPN. The models are generated from specifications developed in SART. One of the important characteristic of this methodology is scalability. It is adaptable to large scale systems. This can be achieved by mapping the specifications of the model components using a bottom-up approach.

One of the lessons learned in this work is the amount of effort needed to design and implement the semantics mapping utility. The process of mapping a large model also requires the support of a specialized tool to extract the components of a large model from one environment and assemble them in the target environment. This tool is currently being developed as part of our on-going research efforts at West Virginia University.

The SART specifications used in this paper have been used in many industrial projects and have become a standard notations supported by most CASE tools. In contrast, a large number of notations and techniques for object oriented specifications has been proposed in the literature. Further work is needed to generalize the methodology presented in this paper to use meta-modeling concepts and techniques [31] to accommodate specifications based on the various notations of object oriented models.

The PCA technique can be applied to measure the complexity of the modules in a software system based on the functional specifications of the system modeled using CPN.

Concurrency complexity notion; introduced in this paper; is an important aspect of dynamic complexity.

A CPN specification model can be used as a tool for FMEA study, hence minimizing the effort required for analyzing the effects of the failures.

Future research in the early risk assessment and complexity analysis could focus on the following areas:

- **Software Architectures based on Object Technology**: The technique presented in this paper with some modifications on the complexity analysis and severity analysis is applicable to the design methodologies and software architectures based on object technology. Further research is required to establish the risk assessment methodology for Object based systems.
- **Risk assessment and software verification and validation (V&V) methodologies at the early stages of development**: The risk assessment methodology presented in this paper can be extended to provide a metric on the effectiveness of V&V tasks in reducing the risk factors associated with software specification modules.
- **Software Reliability Engineering (SRE)**: One main task in software reliability engineering (SRE) is designing the operational profiles. Operational profiles are built according to the user profile and the understanding of the system analyst/designer. They are used for estimating and measuring reliability of the system. Results obtained from the method presented here can be incorporated in an SRE process for conducting reliability analysis at the early phases of development based on dynamic simulation. More research is needed to establish a methodology for incorporating the risk assessment method within the SRE process.

Factor (CCF)) in the system during the execution. The concurrency complexity (ccpcx) is defined as:

$ccpx_i = [(CCF - 1) / F_{max}] * fcpx_i$ (Eq. 13)

where: Fmax is the maximum number of fired transition throughout the whole simulation time

Note that if there is only one process active (or running) in the system, there is no concurrency. Hence we subtract one from the CCF to get the number of modules executing at the same time. The system concurrency complexity (CCPX) is the sum of the subcomponents concurrency complexities (ccpxi):

$CCPX = \sum ccpx_i$ (Eq. 14)

7.9 Severity

Severity is a procedure by which each potential failure mode is ranked according to the consequences of that failure mode. According to the MIL_STD_1629A, severity considers the worst potential consequence of a failure, determined by degree of injuries or system damages that ultimately occurs.

Failure mode and effect analysis (FMEA) technique is a systematical approach which, on a component-by-component basis, details all the possible failure modes and identifies their resulting effects on the system [20]. This technique is used to perform single-random-failure analysis as required by IEEE standard 279-1971, 10 CFR 50. FMEA is done after the initial system designs are completed.

System safety requirements could be determined applying FMEA technique. CPN inherently have all the features required to be used as a FMEA tool since they can be used to determine the effects of a failure or fault in a component. In other words by injecting faults in to the model, one can precisely study the failure propagation throughout the system. By simulating the model, we can observe the effects of a failure in a component in the system.

Ranking severity is rated in more than one way and more than one purpose [21]. In this study severity classifications recommended by MIL_STD_1629A are being used:

1- Catastrophic: A failure that may cause death or system loss.

2- Critical: A failure that may cause sever injury, or major system damage that results in mission loss.

3- Marginal: A failure that may cause minor injury, or minor system damage that results in delay or loss of availability or mission degradation.

4- Minor: A failure not serious enough to cause injury, or system damage, but that results in unscheduled maintenance or repair.

Based on the effects observed after injecting faults to mimic the failure of components in the system through dynamic simulation of the model, a severity index of 0.25, 0.50, 0.75, 0.95 is assigned to minor, marginal, critical, and catastrophic classes respectively. Severity index is used for heuristic risk factor calculations.

7.10 Heuristic Risk Factor (hrf, HRF)

The objective of the risk assessment is to classify the system functional requirements according to their relative importance in terms of such factors as severity and complexity. The heuristic risk factor (hrf) is defined as the measure of risk. There is a strong relation between software quality and the complexity of the component comprising it.

On the other hand, one must assess the effect of failures in the critical components of the system. Severity of a component affects the quality of the software too. These two aspects therefore; are to be considered for overall risk assessment in a (software) system. Hence, we define the heuristic risk factor (hrf) which takes into account the complexity and the severity of components in a system as:

$hrf_i = cpx_i * svrty_i$ (Eq. 15)

where: 0<= cpxi <=1, and 0<= svrtyi <=1

are the normalized complexity level (static, dynamic, or concurrency) and severity level of component i, respectively. The normalized complexity metric of component i is obtained by dividing the component complexity by the complexity value of the most complex component in the system specification (or across several system specification if more than one system specification model is being analyzed).

The system heuristic risk factor is given by:

$HRF= 1 - \prod (1- hrf_i)$ (Eq. 16)

Suppose following results have been obtained for complexity and severity of the modules in the Build_Spacecraft_Realtime_Command (BSRC) component of Commanding. Figure 17 shows the detailed

7.5 Dynamic complexity (dcpx, DCPX)

The relative complexity measure of a program, scpx, is a measure of the program at rest. However, when a program is running, the level of exposure of its modules is a function of the execution environment (operational profile and the platform). Consequently, both the static complexity and the systems operational environment influence its reliability [19]. The dynamic complexity is a measure of complexity of the subset of the code that is actually executed as a system is performing a given function.

While a program is executing any one of its many functionalities, it will apportion its time across one to many program modules depending on the execution profile. The execution profile for a given functionality will be the proportion of time spent in each program module during the time that function was being expressed [14]. Let pi be the probability that the ith module in a set of n modules is in execution at any arbitrary time. The pi could be calculated from simulation results obtained by running the CPN model of the system. All the dynamic measures of system complexity are based on the results obtained from the dynamic simulation of the CPN model (i.e. simulation report) of the system. In such report one can extract the information regarding the dynamic behavior of the system such as the number of times each function has been fired (invoked). This information is used to calculate pi.

7.6 Functional complexity (fcpx, FCPX)

Once the pi is calculated, then the functional complexity (fcpx) of the ith module in the system running an application with an execution profile is defined as:

$$fcpx_i = scpx_i * p_i \quad \text{(Eq. 8)}$$

Similar to the static complexity, the system functional complexity (FCPX) is the sum of the components functional complexities (fcpxi).

$$FCPX = \sum fcpx_i \quad \text{(Eq. 9)}$$

7.7 Operational complexity (ocpx, OPCX)

In the course of execution, changes in the functionalities during program execution will cause the program to select only a subset of possible paths from the set of all possible paths through the control flow graph. As each distinct functionality fi of a program is exercised, a subset or subgraph of the program will execute. This subgraph has its own complexity ci, where ci <= scpxi, representing the complexity of just the code that was executed. This metric clearly cannot be greater than the static complexity of the program. For a specific operational profile, the dynamic cyclomatic complexity (VG) is obtained from the simulation report. From the simulation reports, it is possible to observe what components have been executed and hence the dynamic cyclomatic complexity can be calculated. This process can be automated by a program which reads and analyzes the report.

Once the dynamic cyclomatic complexity is computed, one may calculate the ci as:

$$c_i = (VG' + 1) / (VG + 1) \quad \text{(Eq. 10)}$$

Thus, the operational complexity (ocpx) of the ith module in the system running a program is:

$$ocpx_i = p_i * c_i \quad \text{(Eq. 11)}$$

The system operational complexity (OCPX) is the sum of the operational complexities of the components (ocpxi) it contains:

$$OCPX = \sum ocpx_i \quad \text{(Eq. 12)}$$

7.8 Concurrency complexity (ccpx, CCPX)

One important aspect of software systems built nowadays, is having concurrent processes in the real time systems. Functional complexity and operational complexity, described in the previous sections; do not count for concurrency in a system which is different aspect of the systems dynamic behavior. In other words, FCPX and OCPX do not capture the effects of concurrency in the system. However, it is obvious that for dealing with concurrency, the state space of the system could be very large. A methodology is described below, to measure the complexity added to the system due to concurrency in the software system. A new measure for concurrency related complexity or a concurrency complexity metric for software systems is needed to be introduced here.

Simulation reports are used to measure the complexity due to the concurrency. Concurrency does not exist for single components or modules (primitive transitions), therefore concurrency complexity is to be calculated at the function level. In the example considered here, AOI; as a primitive transition; doesn't have concurrency within itself. But concurrency is present at the BSRC (i.e. function level) by having two or more processes running at the same time.

From the simulation reports, one can measure the maximum degree of concurrency (called as ConCurrency

Table 2 shows all the possible values of cyclomatic complexity of a transition representing a primitive component.

Table 2: Cyclomatic complexity of primitive components

[g]	[G]	[C]	VG_t
Yes	Yes	Yes	$2+VG_c$
Yes	No	Yes	$1+ VG_c$
No	Yes	Yes	$1+ VG_c$
No	No	Yes	VGc
Yes	Yes	No	2
Yes	No	No	1
No	Yes	No	1
No	No	No	0

The system complexity in the model is simply the sum of the complexities of all the subsystems comprising the system. To estimate the static complexity of the components in the system, the PCA technique is used. This technique is applied on the above mentioned complexity metrics. The following section summarizes the PCA technique, for more details the reader is referred to[15] and [30].

7.3 Principal Component Analysis

A software quality model is a statistical relationship between some independent variable (e.g. complexity metrics) and a dependent variable (e.g. software complexity measure). After calculating the model parameters, one can calculate the value of a dependent variable given a set of independent variables.

For PCA, a standardized original data set matrix is developed such that there is a mean of zero and a variance of one. By doing so, the unit of measure will be one standard deviation. $Z_{n\times m}$ is the standardized matrix of the original data set, where each row corresponds to a module, and each column is a standardized variable.

PCA is a technique that transforms raw data variables into new variables, called principal components, that are uncorrelated [17]. If the underlying data set is a software metric, then each principal component is a domain metric, D_j. The principal components are linear combinations of m standardized random variables, Z_1,..., Z_m. The principal components represents the same data in a new coordinate system, in which transformation is chosen both to maximize the variability in each direction and to make the principal components uncorrelated [17].

Let Q be the covariance matrix of Z (i.e. Q = covar [Z]). If Q is a real symmetric matrix with distinct roots, then it could be decomposed as Q = T Y T where: Y = diagonal matrix with eigenvalues y_j along the diagonal. T is the orthogonal matrix where column j is the eigenvector associated with y_j, and T is the transpose of T. The y_j / trace(Q) gives the proportion of complexity metric variance that is explained by the jth principal component. Select principal components with associated eigenvalues greater than one. Let p be the number of principal components selected.

ST be the standardized transformation of matrix T. An element t_{ij} of ST gives the coefficient, or weight of the ith complexity metrics ($1 <= i <= m$) for the jth domain ($1 <= j <= p$). If D = Z * ST, then D_{nxp} is a new vector of orthogonal domain metrics. Each D_j has a mean of zero and a variance of one.

7.4 Relative static complexity

Once the domains are identified, it is possible to compute the relative complexity metric for each program module k by forming the weighted sum of the domain metrics as follows:

$$scpx_k = d_k W' \quad \text{(Eq. 5)}$$

where: W is the vector of eigenvalues associated with the selected domains, W is the transpose of W, and d_k is the vector of domain metrics for specification module k.

A scaled version [18] of this metric, called relative static complexity; is more easily interpreted as follows:

$$sscpx_i = [(STDEV.) * scpxi / SQRT(V(P))] + MEAN \quad \text{(Eq. 6)}$$

where: V(P) = sum of the square of the eigenvalues, and the scaled metric is distributed with a mean of MEAN and a standard deviation of STDEV (typical values for STDEV and MEAN are selected as STDEV = 10, and MEAN = 50.)

The system static complexity (SCPX) is then the sum of the static complexities of all the components (modules) (sscpxi) comprising the system:

$$SCPX = \sum_i sscpx_i \quad \text{(Eq. 7)}$$

7.2 Static complexity (scpx, SCPX)

The model of complexity presented in this paper is based on the CPN model of the functional specification of the system. This complexity model includes the number of inputs to and the number of outputs from a component and its McCabe's cyclomatic complexity obtained from the control flow graph (Eq. 1). Based on the CPN model of the system; we use the following metrics for static complexity analysis:

M, the number of input places,

N, the number of output places, and

VG, the cyclomatic complexity of a primitive transition

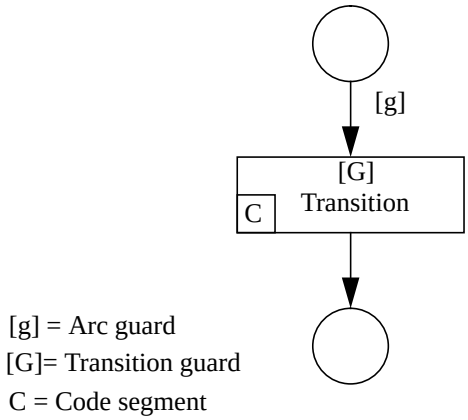


FIGURE 12: A primitive transition with g, G and code segment.

The cyclomatic complexity (VG) is the number of independent paths in a strongly connected graph. McCabe developed this nonprimitive metric which measures some aspects of control flow complexity [16]. This number can be calculated by constructing the control flow graph in a program. In such directed graph, nodes represent entry points, exit points, code segments, or decisions in the program and edges represent control flow in the program. A transition; depending on having guards and/or code segment; will have the following control flow graph (Fire_Transition() procedure in Section 2.2).

Cyclomatic complexity of a primitive transition having guard at input arc, transition guard, and code segment is VG = 2 (Eq. 1). Note that the control flow graph of the code segment should also be considered in calculating the VG. For example if the code segment has the following control flow graph, then the overall VG, VG_t; for that component will be $VG_t = VG + VG_c$.

Different combinations of CPN models and their relevant cyclomatic complexity is shown in Figure 13 and Figure 14. Note that if the code segment by itself has a cyclomatic complexity, say VGc, then the overall cyclomatic complexity, VG_t will be: $VG_t = VG + VG_c$ where VG is as shown in the diagrams below (Figure 14).

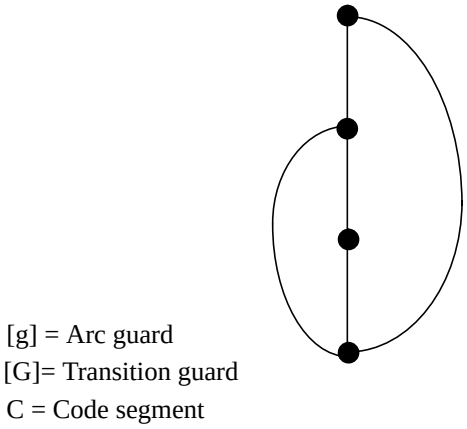
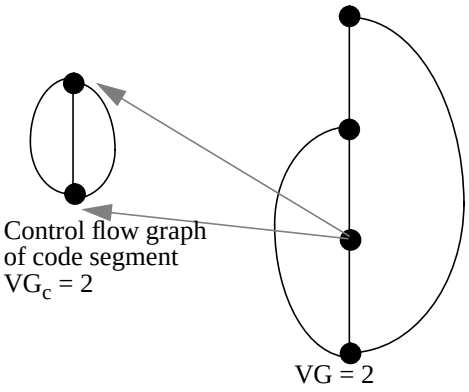


FIGURE 13: Control flow graph of Fire_Transition()



$VG_t = VG + VG_c = 2 + 2 = 4$

FIGURE 14: VG for transition and the code segment

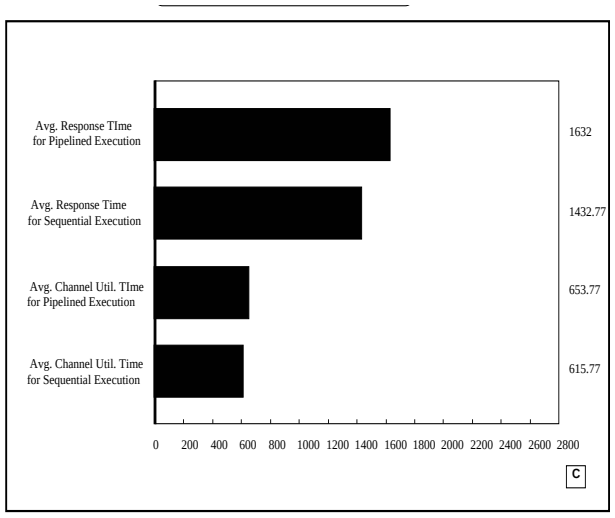


FIGURE 11: Response time & Channel utilization time per command for Sequential & Pipelined execution under failures

total commands processed and total commands uplinked (Figure 9, Figure 9). The average response time (the response time per command) for pipelined execution is greater than that of sequential case because the commands which are being processed by the initial modules still have to wait to be uplinked when a command being transmitted experiences an uplink_failure and is being retransmitted. Although this difference is not much (Figure 9), for a particular command the pipelined case might take a lot longer to uplink the command due to failures in system modules and uplink failures of the previous commands as well as itself. Also when realtime commands are read from a preplanned script then the rate of input commands would be greater and this would make the commands wait longer than usual in the pipeline case. This also adds to the risk involved for the pipelined execution. Since under real time conditions a command might miss the deadline for uplinking which may result in hazardous conditions. This needs to be taken care of in the system design.

7 Risk Assessment using HRF

Traditionally we measure system reliability in terms of time between failures or in terms of number of faults within a certain period of time. These two measures are, however, interchangeable. There is a correlation between the number of faults and the complexity of the system [13], [14]. The more complex a system is, the more the number of faults are expected to be in the system. However, to improve software quality, one must be able to predict early on, in the development process, those components of the software system that are likely to have a high fault rate. Hence, complexity metrics are used in this study.

On the other hand, one must also identify those components in the system which require special development resources due to their severity and/or criticality. There could be a relatively less-complex component (or subsystem) but, having a safety role in the system at the same time. The effect(s) of failures in such component could be catastrophic (e.g. braking system in a car may not be very complex but, its failure could lead to the passengers death.) Therefore, for risk assessment it is necessary to considers the severity associated with each component based on how its failure(s) affect the system operation and performance.

These two aspects therefore; are to be combined for overall risk assessment in the (software) system. The following subsection presents the methodology and applies it to BSRC component of the Commanding subsystem (Figure 17).

7.1 Complexity

Characterizing the software quality in terms of a set of measurable attributes is the main objective in any software measurement process. Based on the observation that software complexity has a direct impact on its quality [14], we focus on complexity metrics. Higher likelihood of failures is expected if a complex module is executed. The important fact here is that measures of software complexity can be obtained very early in the software life cycle.

There are several software complexity metrics such as: Halstead metrics [23], Lines Of Code (LOC), control flow graph, data flow graph, number of fault counts, number of change counts, etc. Linear combinations of these elements are measures for most of the existing software metrics. However, it should be noted that these measures are interrelated together. In other words we are facing multicollinearity in these measures. To resolve this problem, we apply a statistical technique known as Principal Component Analysis (PCA). This technique detects and analyzes the relationships among software metrics and provides us with a new set of orthogonal variables (known as principal components) that convey all, or almost of, the information in the original set. The principal components are constructed so that they represent transformed scores on dimensions that are orthogonal [15].

processes are inactive and the next Operator_Command_Input will be processed only after the current command is transmitted. In the pipeline design the firing of transition T-10 deposits a token in the Wait_Command state as well as Verify_Command state. This enables the Build_Spacecraft_Realtime_Command to process the next Operator_Command_Input even while the previous command is still being processed. This is propagated to all other processes down the line. Thus a pipelined execution of the system is effected.

The performance improvement under this pipelined design is observed. The throughput of commands for the pipelined execution is double the throughput for sequential execution under the same conditions. The average response time is only slightly greater than the sequential case. The response time per command was constant throughout the execution of the simulation. This is because the input rate of commands is low considering the time taken for the operator to input the command. Thus the average response time for the pipelined case is almost equal to that of the sequential case, but if a sequence of preplanned realtime commands is read from a script of commands then the average response time of a command will be much larger than the sequential case. This is because the input rate would be higher and there will be a build up at the communication channel. The commands have to wait for longer times at the channel for being uplinked. This is an added risk in the pipelined design since some of the commands will have a large waiting time. These commands may violate the deadlines of uplinking and execution because of this delay.

Performability analysis of the pipelined design

In this case, the commanding model was simulated with faulty behavior in the system. This was accomplished by simulating the effects of failure and recovery in the system functions such as BSRC and Verify_Command. The failure and recovery activities of these modules were simulated by adding CPN ML code in the code segments associated with the respective transitions to simulate the activity of causing a failure with which an estimated recovery time is attached. The degraded performance of the system under failures and repairs is observed in the simulation. This scenario is simulated for a sequential design of the system and also for a pipelined design and the relative performance is evaluated.

The throughput of commands for the pipelined execution under failures of the system modules is again larger when compared to the throughput for sequential execution under the same conditions (Figure 9, Figure 9). This indicates that the pipelined design has improved the system performance under faulty conditions also. However, the

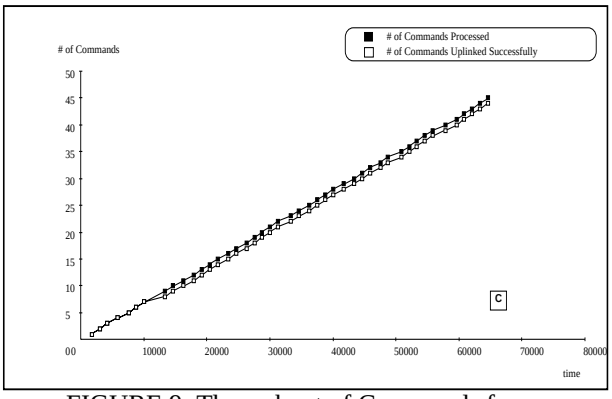


FIGURE 9: Throughput of Commands for Scenario 5 i.e. Sequential Execution under Failures in Communication and Fault injection

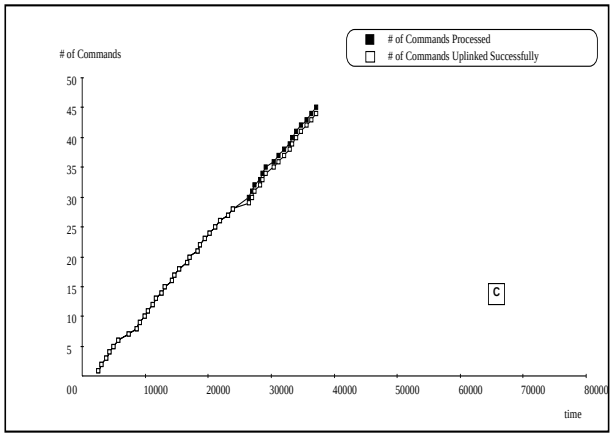


FIGURE 10: Throughput of Commands for Pipelined Execution under Failure in communication and Fault injection

throughput of the pipelined model should reduce when commands are flushed from the pipeline due to an uplink or execution failure. The response time per command in both pipeline and sequential execution is greater than (almost 2.5 times) the channel utilization time per command. When there are failures in the system modules then the commands experience a delay in processing and there will be a decrease in the build up at the communication channel. When the faulty modules like BSRC or Verify_command fail and the system takes some time to recover to normal execution, the communication channel can uplink the commands which were already processed and verified.

It is also observed that the processing time for each command is not the same throughout and that some of the commands are lost without being uplinked. Whenever there is a loss of command we can observe this by the widening of the gap between the lines representing the

to the state Wait_Command and waits for the input of a command. The function init() is invoked which initializes the statistical variables needed for the dynamic analysis.When an Operator_Command_Request token is received by the Controller it goes to the state Build_Spacecraft_Realtime_Command and enables the BSRC module (transition T9 is fired). When the BSRC module is done processing the request the Controller enables the Verify_Command function (transition T10 is fired). If the Command_Authority_Violation (CAV) is false and the Command is Non_Critical then the Controller enables Merge_Command (transition T4 fires). If the Command is Critical then the operator is informed (transition T5 fires) and upon receipt of a positive response from the operator goes to the Merge_Command state (transition T7 fires).

When Merge_Command function is done and the Uplink_Required token is set to true then the Controller goes to the Uplink_Command state (transition T11 fires). The Controller enables the Transmission_Command function to uplink the next command. When the uplinking of a command is done then the Next_Command token is set to true so that the next command can be uplinked (Transition T13 puts a token in the Next_Command place). When a command is being uplinked the Controller enables the Count_Transmission_Number function (transition New_Com fires).The transmission command function can be fired for retransmitting a command when an uplink failure occurs. The retransmission is done only when the Retransmission_Command token is set to Enable and when the Transmission_Limit_Reached is not false.

When the Spacecraft_Command_Status token received from Evaluate_Spacecraft_Command_Status function is set to Uplink Failure then a retransmission of the command is attempted. The operator is also notified of the Uplink Failure. If there was no failure then the Controller goes back to the Wait_Command state to process the next command (transition T16 fires).

The code segments of the transitions update the number of Commands processed and the number of Commands uplinked and collect the statistical data for the execution.The CPN model for Commanding was validated using the formalism of petrinets. Through the simulation of this model several inconsistencies were found in the SART model of Commanding.

6 Performance Analysis of the Commanding Component

This section discusses the performance analysis carried out on the Commanding component of the EOS system and presents the results and conclusions of the analysis.

The following scenarios were simulated to assess the performance and/or performability of different execution profiles of the Commanding model.

Performance under normal sequential execution

The Commanding model was simulated to analyze the performance under favorable conditions. The timing behavior of each module was specified and it was assumed that all the modules function normally. The simulation of this scenario produced measures on the throughput and total execution time of the operator commands. The average response time for a command and the average channel utilization time were also calculated. The average channel utilization time was almost equal to the average response time. This indicates that the bottleneck in the system is the communication channel. This is because the time taken to build, validate & verify a command is comparably less than the time taken for uplinking of the command and downlinking of the command_status. The total execution time for a sequence of 45 operator input commands was more than the sum total of the response times for these commands because of the time taken for the input of the commands. The response time per command was constant, since each command encounters the same conditions.

Performance under pipelined normal execution

In this case instead of processing one command at a time, a sequence of operator commands are pipelined through the system. Several functions are concurrently active to process the command sequence.

The flexibility of CPN notation to express the control flow of a system greatly reduces the amount of effort needed to design alternate specifications and explore the system behavior under new specifications. The State Transition Diagram specification of Teamwork is limited in the sense that it does not allow the specification of a parallel design without the introduction of many more states and transitions; making the system too complex to visualize and analyze. The specification of the pipeline design in the Design/CPN model is almost identical to the sequential design. The minor modification that was done in the EOC controller is an addition of an arc from the Transition T-10 to the State Wait_Command (**figure** 6). In the sequential scenario the when Verify_Command processes the output of Build_Spacecraft_Realtime_Command, all other

different. The enabling places get one token in the state where a process is to be enabled. Once the token is consumed by the enabled process, there is no more enabling token until the system goes back to the same state. The Hatley Pirbhai approach can be incorporated through slight modifications to our mapping rules.

5 Description of the Commanding Subsystem of EOS

The Earth Observing System (EOS) being developed by NASA is a large scale Parallel and Distributed System. Based on the requirement specifications, and scheduling scenario, it was observed that the commanding module plays an important role. A model of the Commanding Subsystem was built based on the requirement specifications of NASA. The major tasks performed by the Commanding module are listed below according to the NASA requirement specifications [32],[33].

- Generate and verify real-time commands. This is accomplished by the functions "Build_Spacecraft_Realtime_Command" and "Verify_Command" [32].
- Merge and uplink the pre-planned and real-time commands to EDOS. The functions "Merge_Command" and "Transmission_Command" are responsible for this job [32].
- Receive and evaluate the command status. This is done by functions "Evaluate_ Spacecraft_ Command_Status" and "Receive_ Command_ Status_Data" [32].
- The automatic retransmission is also provided when an unsuccessful transmission occurs. This is managed with the help of the function "Count_ Transmission_ number" [33].

The Build_Spacecraft_Realtime_Command function generates the spacecraft realtime command based on the operator command input and the pre-planned command script[32]. Verify_Command checks the authorization level of a command [32] and determines whether a specific command is critical based on its definition [33]. Merge_Command merges a Valid_Realtime_Command and a Valid_Preplanned_Command. Transmission_Command receives Uplink_Data_ Stream from the Merge_Command sub-function and sends it to the space crafts as a Spacecraft_Uplink_Data. Count_Transmission_Number controls the retransmission efforts needed when the data received from the space craft indicate that the command has been rejected. Receive_Command_Status_Data is used to monitor the status of data received by the space craft. The "Evaluate_Spacecraft_Command_Status" function verifies the successful receipt and execution of all commands by the spacecraft [32][32].

5.1 CPN model of the Commanding subsystem

The model of the Commanding subsystem as described in the previous section was built using the CASE tool TeamWork. The Teamwork model of Commanding was translated to the Design/CPN environment for Dynamic Analysis. The translation was preformed by mapping the semantics from Teamwork to Design/CPN as described in [7]. The Hierarchy page of the CPN model is shown in Figure 8. The CPN page corresponding to Commanding DFD is shown in Figure 15 and the CPN page corresponding to the Controller for Commanding is shown in Figure 16. The CPN model mapped from the Teamwork model needs to be completed by adding the missing semantics needed for dynamic analysis. CPN Meta Language code is written to map the outputs from the inputs. The information needed to implement a particular scenario is also added to the model. Thus the model is customized for each simulation and the behavior of the model is analyzed under different scenarios.

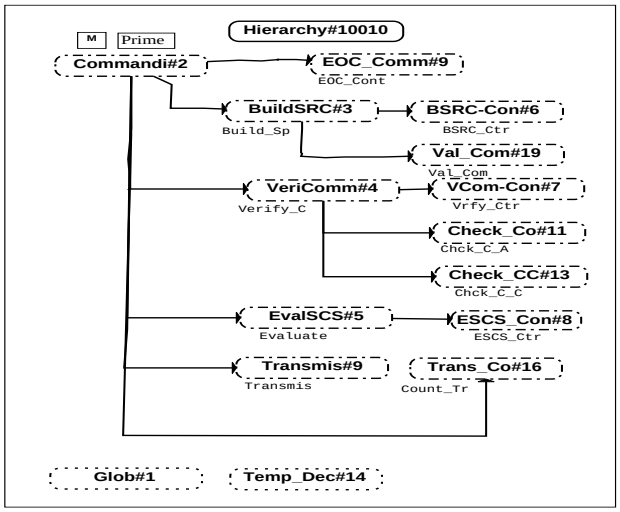


FIGURE 8: The hierarchy page of CPN model.

The CPN page corresponding to the Controller for Commanding is shown in Figure 15. It contains the control specifications for the Commanding module as a whole. It produces the tokens necessary for the invocation of the functions of the Commanding module.

The transition T1 is enabled by the presence of the token START. When the transition T1 fires the Controller goes

substitution transition represents the mapping of its C-Spec. A state transition diagram is the tuple (std_state, std_tb, std_trans):

- std_state: is the object representing state in the STD
- std_tb: is used to give a definition to an STD as mealy or moore.
- std_trans: represents the transition object.

The mapping rules used are:

Rule: $\forall$std_state $\longrightarrow$ P

Rule: $\forall$std_trans $\longrightarrow$ T

Rule: $\forall$std_tb $\longrightarrow$ f(G,C,P) where both guard (G) as well as code segments (C) are derived from the conditions needed for the change in the system state. The places (P) correspond to the input and output signals.

A cruise control example is presented in Figure 6 to illustrate the mapping process for State Transition Diagrams.

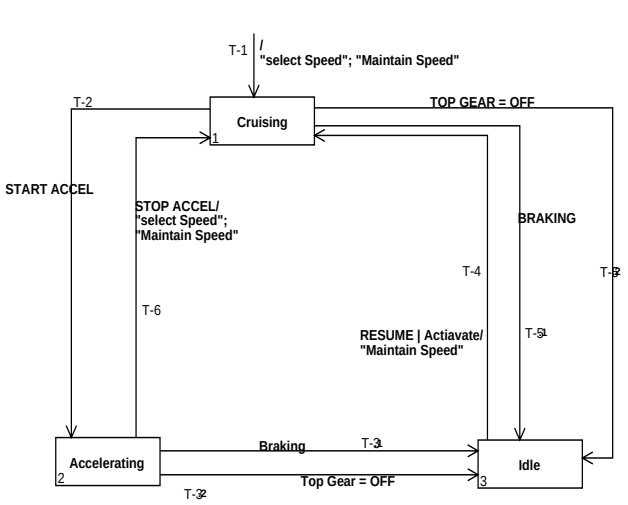


FIGURE 6: Cruise Control STD.

The transitions are marked as T-1 through T-6 for a comparison with transitions in the CPN page shown in Figure 7. The places are named according to the mapping rules given earlier. Two places named as Enable_Select_Speed and Enable_Maintain_Speed are used for enabling/disabling the processes, Select_Speed and Maintain_Speed, respectively. The presence or absence of a token in these places, is used to enable or disable the corresponding process.

Different shades are used to distinguish between places in Figure 7. The places shaded black are the enabling, disabling places. The larger size places shaded grey are the states of the controller. The places with light grey shade are input signals to the controller. There are always tokens present in the input places. For example, the place Braking has a token with a value TRUE if the brakes are on, otherwise it is false. Guards for the transitions are shown in a separate labeled box to reduce diagram clutter.

Note: Arc inscriptions and intermediate place names in Figure 7 are hidden from the view for the sake of clarity.

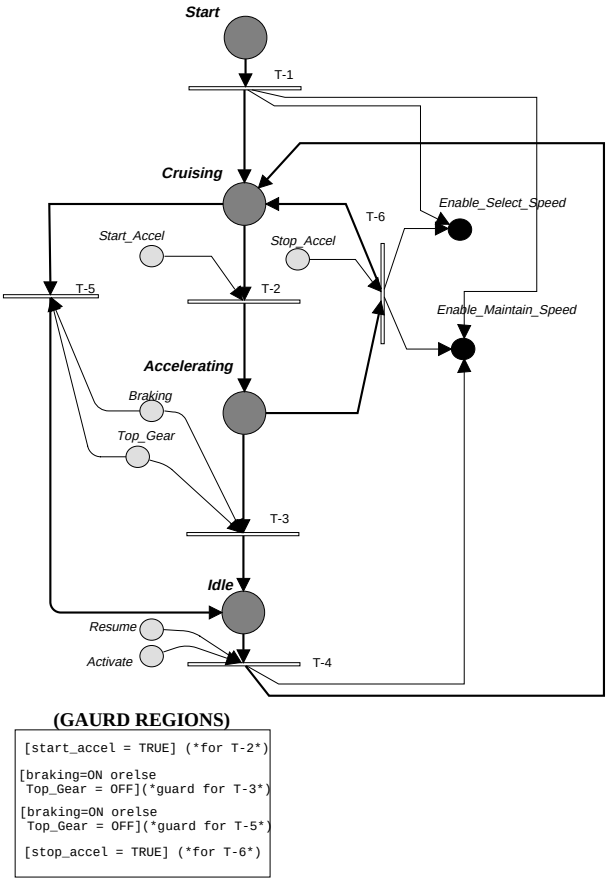


FIGURE 7: CPN diagram for Cruise Control

Once the output file is generated based on these rules, the analyst checks for consistency and completeness. The important aspect related to the enable and disable function of an STD is based on the Hatley and Pirbhai notation. Actions (enabling and disabling) are associated with transitions which are transient in nature. The actions are assumed to continue in effect until the next transition occurs. This means a process activated by a particular action remains activated continuously and continues to respond to changing data inputs until the next transition occurs [36]. The approach used in this paper is slightly

Rule: ∀ non-primitive dfd_bubble, dfd_buble ⟶ SN. Such that the input and output flows of this dfd_buble are mapped to the sockets. The inputs and output flows get mapped to ports in the subpage. In the same step, based on the flow directions on the DFD, PT is also defined from the set of its values {in, out, in/out, general}.

Rule: ∀ dfd_store ⟶ f(T,P,G,A)

A dfd_store can be read-only or read-write. The f(T,P,G,A) represents this accordingly.

Rule: ∀ dfd_term ⟶ P

Rule: ∀ dfd_csc ⟶ CPN page

Rule: ∀ dfd_flow ⟶ P

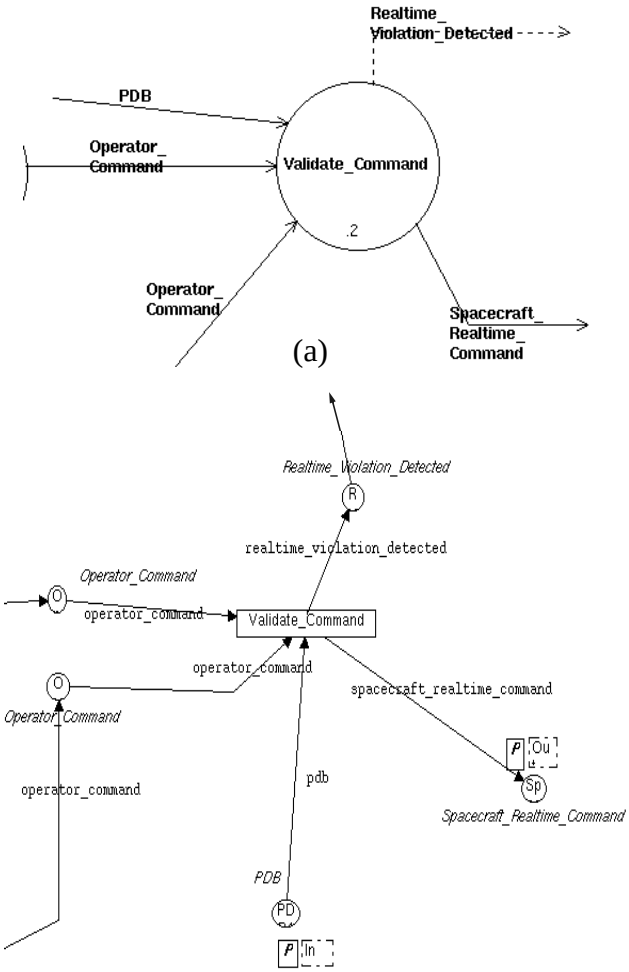


FIGURE 5: (a) SART representation of "Validate_Command", (b) The CPN representation of "Validate_Command".

This is just an overview of the mapping rules used for the translation of a data flow diagram.

As shown in Figure 5, a primitive dfd_buble "Validate_Command" (Figure 5-a) is mapped to transition with the same name (Figure 5-b). The input data flows called PDB, Operator_Command_1 and Operator_Command_2 are mapped to the places with the same name. The direction of arcs in Figure 5-b is also mapped from the corresponding direction of data_flows shown in Figure 5-a. The same thing is true for the outputs of the process "Validate_Command". As shown in Figure 5-b, the output places spacecraft_ realtime_ command and PDB are ports. In other words, these places are connected to the corresponding sockets on a superpage. The place spcecraft_realtime_command is an output port. Whereas the place PDB is an input port.

Mapping the Data Dictionary Entries

The data dictionary entries are mapped to corresponding color entries in the global declaration page. In SART the data dictionary element is a three tuple (dde_name, dde_attr_list, edif_body). During the mapping process a dde_name is used for adding a particular color. The objects dde_attr_list (the attribute list for a DDE) and edif_body are used to generate multisets corresponding to a color set in the CPN environment. Table 1 shows some rules for the DDE mapping process.

Table 1: Mapping rules for generating the Global Page of the CPN model.

DDE definition	CPN translation	Remarks
Activity_Violation_Detected = ["TRUE" | "FALSE"]	color Activity_Violation_Detected = **with** TRUE | FALSE;	complete definition
ADCs-Data2 = Information_Dialog + Dialog + Algorithms + Ancillary_Data + Data_Products + Data_Information	color ADCs-Data2 = **record** Information_-Dialog * Dialog *Algorithms *Ancillary_Data * Data_Products * Data_Information;	complete definition
Alarms_Notification = *not-defined*	color Alarms_Notification = **with** Alarms_Notification;	Default color
When the definition of Alert_signal is not found in the DDE table	color Alert_Signal = **with** Alert_Signal	Default color

Mapping the Control Specifications

The C-Spec in a DFD appears as a substitution transition on the corresponding CPN page. A subpage for this

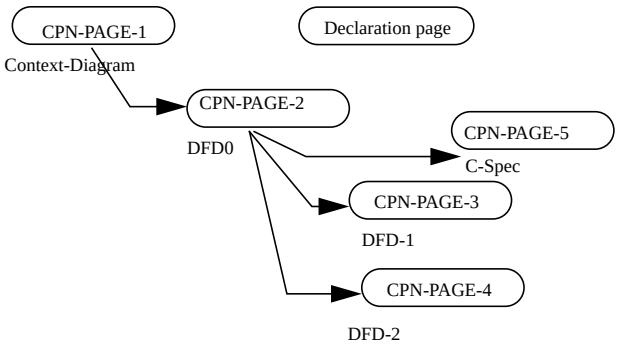


FIGURE 4: CPN page showing model hierarchy

4.2 Mapping SART Objects to CPN Objects

An SART model, as defined in Case Data Interchange Format, is expressed as N-tuple. In this paper only 4-tuple variant of CDIF representation are considered.

- Obj_dfd: A data flow diagram object.
- Obj_dd: Represents the collection of Data Dictionary Entries (DDE). The DDEs hold information on data flows, control flows and data stores.
- obj_std: Used for State Transition Diagram.
- obj_ps: Object containing P-Spec.

The hierarchy definitions are embedded in the individual objects like data flow diagrams, state transition diagrams, data dictionary entries and P-Specs. The mapping procedure presented here uses this description to maintain the hierarchy in the resulting CPN model structure. This procedure was implemented using flex and bison along with mapping rules written in the C-language.

The hierarchical colored Petri Nets (HCPN) are defined as a tuple:

- S: a finite set of pages such that each page $s \in S$ is a non-hierarchical CPN=($\sum_s$, P_s, T_s, A_s, N_s, C_s, G_s, E_s, I_s), Where:

  $\sum$: finite set of non-empty types (or color sets)

  P: finite set of places

  T: finite set of transitions

  A: finite set of arcs

  N: node function

  C: Color function (P into $\sum$)

  G: guard function. It is defined from T into expressions.

  E: arc expression function. It is defined from A into expressions.

  I: initialization function. It is defined from P into closed expressions.

  Note: Further details are available from the reference [35].

- $SN \subseteq T$: a set of substitution nodes
- SA: a page assignment function. It is defined from SN into S such that no page is a subpage of itself.
- $PN \subseteq P$: a set of port nodes
- PT is port type function. It is defined from PN into {in, out, in/out, general}
- PA: port assignment function. It is defined from SN into binary relations as:

  1. Socket nodes are related to port nodes.

  2. Socket nodes are of the correct type.

  3. Related nodes have identical color sets and equivalent initialization expression.

- $FS \subseteq P_s$ is finite set of fusion sets such that members of a fusion set have identical color sets and equivalent initialization expressions.
- FT is fusion type functions. It is defined from fusion sets into {global, page, instance}.
- $PP \subseteq S_{MS}$ is a multiset of prime pages

The SART objects are mapped to HCPN using the rules given in the following paragraph.

Mapping the SART model to a CPN model

Rule: $\forall$ obj_dfd $\longrightarrow$ non-hierarchical CPN page

Rule: $\forall$ obj_std $\longrightarrow$ non-hierarchical CPN page

Rule: $\forall$ obj_dd $\longrightarrow$ CPN declaration page

The obj_ps (the process specifications) is not mapped automatically using a semantic transfer utility, rather the analyst converts the process specifications to a code suitable for the CPN. This code can be used in the CPN related functions and the transition code segments.

Mapping the Data Flow Diagram to CPN page

A data flow diagram is a six tuple DFD = (dfd_buble, dfd_store, dfd_term, dfd_tb, dfd_csc, dfd_flow). Each of these objects are further defined in the CDIF standard.

Each CPN page is tuple object ($\sum$, T, P, G, A). These objects are mapped from the data flow diagram object as given below:

Rule: $\forall$ dfd_buble $\longrightarrow$ t such that $t \in T$, and dfd_buble is a primitive one.

The data and control flows connect the terminators and the bubble.

The bubble on the context diagram is decomposed into more bubbles or processes on DFD 0. Each of these bubbles will be numbered as 1, 2, etc. The analyst determines if any of these processes are primitive.

For a primitive process, a P-Spec is defined. If a process is not primitive, a lower level DFD is used to define it further. For example, in Figure 3, two processes in DFD 0 are defined by the lower level DFD 1 and DFD 2. These steps are repeated until the analyst reaches the primitive level for every process in the model. Definitions for data flows, control flows and stores constitute the data dictionary for a given model. The fields of the data dictionary corresponding to individual flows or stores are called DDE.

Also, a data flow diagram may contain a C-Spec. C-Specs are used to define process activation or handling control flows. A vertical bar on the DFD represents a C-Spec. The C-Spec is further defined on a separate sheet in the SART model. Several representations are in use for defining C-Specs. Some of the examples are State Transition Diagram, Process Activation Table and Decision Table. In this work, only State Transition Diagrams are considered.

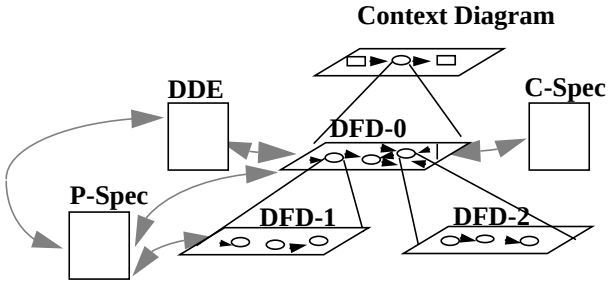


FIGURE 3: Components of SART and their relationship with each other.

The Design/CPN Environment

The Design/CPN modeling environment is also hierarchical. A CPN model is arranged in the form of pages as shown in Figure 4. Each page in this case has an associated SART object which is shown just outside the CPN-Page's oval representation. The CPN-PAGE-1 is mapped from the Context Diagram. Similarly CPN-PAGE-2 is mapped from the DFD 0. Other DFD levels are mapped to corresponding CPN pages as shown in the Figure 4.

The oval for Declaration page is not connected in the hierarchy diagram as it does not contain a CPN. The definitions of the colors and declaration of CPN variables of different colors are specified in the declaration page. The entries in the declaration page are derived from the DDEs of the SART model.

Every CPN page contains a colored petrinet. A colored petrinet is a petrinet in which different places can have different types of tokens (colors). CPN uses data types, data objects and variables. CPN data types are called colorsets and CPN data objects are called tokens. A CPN consists of the following elements:

- Places (represented by circles): locations for holding data
- Transitions (represented by rectangles): activities that transform data
- Arcs (represented by arrows): connect places with transitions. Arrowhead specifies token flow. Input arcs bring tokens to the transitions and output arcs show the paths leaving a transition.
- Arc inscriptions: Input arc inscriptions specify the data that must exist for an activity to occur, and output arc inscriptions specify the data that will be deposited if an activity occurs. Time stamps on the arc inscriptions represent the delay in the flow of tokens.
- Guards (attribute of transition): define conditions that must be true for an activity to occur. Guards can contain time stamps.
- Code segments (attribute of transition): contain code to implement exact transformation from input tokens to output tokens. The code in these segments is written using CPN Meta Language.

As shown in Figure 4, CPN-PAGE-1 is a superpage for the rest of pages named CPN-PAGE-* (where * is an integer 2 through 5). Inversely CPN-PAGE-2 is a subpage for CPN-PAGE-1.

Each of these objects (i.e. places, arcs and transitions) have their own sets of attributes. Objects other than the ones just discussed, which may exist on a CPN pages are text blocks and local declaration pages. These are not described here for the sake of brevity. Mapping of individual CPN pages from the corresponding DFDs is explained in the following section.

Most of the complexity metrics are based on the design specification.

3 A Methodology for Generating Formal Dynamic Models

For many years developers have been using informal techniques such as SART for requirements modeling and specifications. Maier [1] has listed various advantages of using SART methodology. Integrated development environments, e.g. Integrated CASE (ICASE) tools, have evolved to support a number of notations for requirements modeling using SART as well as object-oriented analysis. Such informal specifications are scalable and are being used in large industrial projects. A large gap exists between complex notations used for formal specifications such as CPN and the informal notations used in ICASE tools.

This paper addresses the problem of integrating verification and analysis tools based on CPN with ICASE specification tools as shown in Figure 1. Semantics mapping rules are used in the integration process. The process maps hierarchical requirements models developed in SART notation to hierarchical models in CPN notation. The above approach is implemented using the following tools:

- Teamwork for SART models,
- Design/CPN for CPN models. [1]

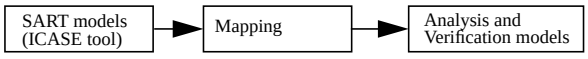


FIGURE 1: The process of mapping SART models to CPN models.

3.1 A Framework for Tool-to-Tool Data Transfer

A framework for tool-to-tool data transfer is shown in Figure 2. This is a general case showing the transfer of data from a CASE tool to a verification tool. The analysis and design data is normally available from the CASE tool database. Most CASE tools provide an access utility to allow others tools to retrieve this data.

Software Bus Facility (SBF) acts as a communication channel between different tools in an integrated environment. Once a data package is available via the SBF, any tool (including the verification tools) can receive this data through a local gateway. Once the local gateway completes the process of receiving data, the Semantics Transfer Utility (STU) converts the input data semantics to data objects of the verification tool. This general framework for tool-to-tool integration is implemented using Teamwork and Design/CPN.

4 Semantics mapping rules

The STU uses a set of mapping rules for translating SART objects to the CPN objects. A brief overview of the SART and the CPN environment is given in Section 5. In Section 4.2, the semantics mapping rules for translating a SART model to a CPN model are briefly described.

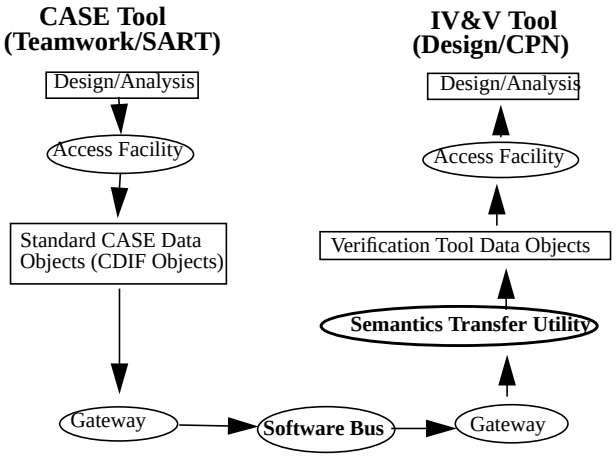


FIGURE 2: General framework for Tool-to-Tool Integration.

4.1 Description of the SART and the CPN Environments

The Teamwork/SART Environment

The SART model components are shown in Figure 3: Data Flow Diagrams (DFD)[2], Control Specifications (C-Spec), Process Specifications (P-Spec) and Data Dictionary Entries (DDE). The DFD at the highest level of abstraction is known as the Context Diagram. This diagram allows three types of objects: a bubble, terminators, data and control flows. The single bubble represents the whole system. Terminators represent the external entities which send or receive data or control signals from the system.

1. Teamwork is a COTS CASE tool by Cayanne Software. More information is available at www.cayennesoft.com/ products. At present Design/CPN it is available from University of Aarhus (http://www.daimi.aau.dk/designCPN/).

2. Control Flow Diagrams are merged with DFDs

Output arc inscriptions specify the data that will be produced if an activity occurs.

- Guards define conditions that must be true for an activity to occur.

CPN use data types, data objects, and variables. CPNs data types are called colorsets and CPNs data objects are called tokens. A guard is a Boolean expression associated with a transition or with input arcs. Moreover, transitions in a CPN could have code segments in which we can implement the exact transformation from input data to output data using CPN ML language.

The dynamic aspect of CPN models are denoted by markings which are assignments of tokens to the places of a CPN model. A transition is enabled if and only if each of its input places contain at least as many tokens as there exists arcs from that place to the transition. When a transition is enabled, it may fire. And when it fires, all enabling tokens are removed from the input places and tokens (according to the code written in the code segment, if any) are deposited in each of the output places. At any given time instance, the distribution of tokens on places defines the current state of the modeled system.

By adding g (guards to the input arcs) and/or by adding G (guard to the transition itself), one can control the enabling conditions of the transition. Therefore; the complete dynamics of a typical CPN model could be summarized in the following procedure:

```
Fire_Transition(){
// check for enabling conditions at the input guards
        if g then
        //check for enabling condition at the
// transition guard
         if G then
        execute code c(); //if any!
        deposit token to the output places;
 end if;
end if;
}
```

2.3 Software Complexity Metrics

Like the reliability models, many complexity metrics and models have been practiced by computer scientists and software engineers. In 1977, Halstead established a software complexity metrics based on the primitive measures such as number of distinct operators and operands, and the total number of operators and operands in a program. He developed system of equations expressing some measures of software quality.

In 1976, McCabe introduced a measurement of cyclomatic complexity as an indicator of testability and maintainability of a program. Cyclomatic complexity is based on the classical graph theory. Cyclomatic complexity which shows the number of regions in a (control flow) graph and is defined as:

$$VG = e - n + 1 \qquad \text{(Eq. 1)}$$

Where e is number of edges and n is number of nodes. These complexity metrics, however; are measuring implicitly the complexity of a module as a separate entity. To take into account the interactions among modules in a system, several structure metrics have been introduced. McClure introduced invocation complexity (1978), system partitioning measures presented by Belady and Evangelisti (1981), information flow metrics by Henry Kafura (1981), and in 1980 Yau and Collofello used stability measures [24]. Henry and Kafura defined structure complexity using design structure metrics as:

$$C_p = (\text{fan\_in X fan\_out})^2 \qquad \text{(Eq. 2)}$$

where:

fan_in = the number of modules that call a given module

fan_out = the number of modules that are called by a given module

And Selig defined a hybrid complexity to incorporate the information flow metric in 1990 as:

$$HC_p = C_{ip} \text{ X } (\text{fan\_in X fan\_out})^2 \qquad \text{(Eq. 3)}$$

where C_{ip} is the internal complexity measure of the module p which could be McCabe's cyclomatic complexity measure. Card and Glass (1990) developed a system complexity (C_t) model which includes structural complexity (S_t) and data complexity (D_t) as[24]:

$$C_t = S_t + D_t \qquad \text{(Eq. 4)}$$

where: S_t = sum[(fan_out of module i)2] / number of modules

D_t = sum[D_i] / number of modules

D_i = (I/O variables in module i) / (fan_out of module i + 1)

on the one hand and quality factors, such as maintainability or reliability on the other hand [10], [11], [12].

Khoshgoftaar et al. [10] used complexity metrics such as lines of code (LOC) and control flow graphs to detect fault prone modules in a very large system at the code level. By utilizing the models developed based on the principal component analysis and the product metrics, they classified the modules in a system as fault-prone or not fault-prone. They demonstrated that the model results could be used to identify those modules that would probably benefit from extra attention, and thus, reduce the risk of unexpected problems in these modules.

Ebert [25] applied complexity metrics during the development of large telecommunication software in order to identify high risk components and to tailor reliability growth models. He used complexity-based metrics to predict criticality of the modules in the system. Doing so, he was able to identify the critical components and to make predictions on the failure rate as early as possible in the software life cycle. For the complete approach of criticality prediction recent data from the development of a switching system with around 2 MLOC was provided. The switching system is currently operational and thus allowing for validation and tuning of the prediction model.

Complexity metrics do provide substantial information on the distinguishing differences among the software systems whose reliability is being modeled and maybe used in the determination of initial parameter estimates. There are some predictive models that incorporate a functional relationship of program error measures with software complexity metrics [26]. Software complexity metrics are also used for test design and test execution suites [27].

Complexity (cpx) analysis and measurement benefit from applying principal component analysis; a proven mathematical technique for the complexity metrics. Khoshgoftaar et al. [28], used this technique for early prediction of software quality on a large telecommunication system to identify fault-prone modules prior to testing. This research was based on the complexity metrics (such as call-graph metrics, control-flow-graph metrics). This work basically demonstrated that; in the software maintenance context; software systems quality can be predicted from the measurements of early development products and reuse indicators.

For static complexity measurement, we used in this paper information available from the CPN model of the system. Dynamic complexity (functional, operational, and concurrency) measures are obtained through analysis of the dynamic behavior of the system by simulating the CPN model.

2 Background

2.1 Software System specifications

Requirements supplications languages (or conceptual grammars for requirements specifications) are classified by Fraser and Kumar [12] into two major groups: formal specifications and informal specifications. Informal specifications models supported by CASE tools used in industry are based on SART models or Object-Oriented Analysis models. Formal specifications are based on formal languages such as VDM, Z and Petri Nets.

Informal specification languages use a combination of graphics and semiformal textual grammars to describe and specify software system requirements [2], [3], [5]. These languages are ideal for a developer's environment, as they make it convenient for both user and developer to communicate with each other and refine the user-description to a set of informal requirements documents. These languages tend to be imprecise and ambiguous. Hence there is a need to use formal specification languages for the requirements analysts domain [7]. A formal notation can be analyzed and manipulated using mathematical operators. Mathematical proof procedures can be used to test and verify the internal consistency and syntactic correctness of the specifications [5]. Formal languages provide exactness and the ability to reason [3]. If the problem can be specified mathematically, then a program can be developed and proven to satisfy the specification.

The CPN modeling environment can be used for software requirements and design specifications. It is especially useful in rigorous analysis of the dynamic behavioral properties such as concurrency analysis, performance analysis, safety, reliability analysis and reachability analysis. Reachability analysis [4] is based on Hierarchical Reachability Graph (HRG). This work shows the applicability of CPN based analysis to large scale models.

2.2 Coloured Petri Nets (CPN)

A simple Coloured Petri net is composed of the following graphical elements:

- Places (represented by circles) locations for holding data.
- Transitions (represented by rectangles) activities that transform data.
- Arcs (represented by arrows) connect places with transitions, to specify data flow paths.
- Arc Inscriptions: Input arc inscriptions specify the data that must exist for and activity to occur, and

real-time component of NASA's Earth Observing System.

Organization of the paper

The paper is organized as follows: A brief description of dynamic models, performability analysis and risk assessment is given in the rest of Section 1. A discussion on the basic concepts related to software system specifications, CPN and software complexity metrics is provided in Section 2. The methodology of generating dynamic models is discussed in Section 3. The details on the semantics mapping of CASE-based models to CPN models are given in Section 4. The CPN model of the EOS system component is described in Section 5. Performance/ performability analysis is presented in Section 6. The risk assessment methodology is described in Section 7.

1.1 Generation of Dynamic Models

This paper presents a methodology to integrate a CASE environment based on SART (Structured Analysis with Real Time) notation and CPN based verification environment. Semantics mapping rules are used to map SART objects to corresponding CPN objects. The mapping rules presented greatly simplify (in contrast with previously published work [6], [8]) the development of large CPN models. Therefore making these techniques applicable to software models of large systems. Using the CDIF (Case Data Interchange Format) standard, SART models are exported to a Semantics Transfer Utility. This utility maps the SART model semantics to CPN notation. The methodology has been implemented using a COTS CASE tool and Design/CPN environment.

1.2 Performability analysis

The CPN model captures both the static and the dynamic behavior of the specification. In the early design stages the functional modules are relatively large and the knowledge of their execution behavior may be imprecise. As the design progresses and the modules are further resolved, the estimates of their behavior and execution resource characterization become more precise. A CPN model helps in giving the definition and subsequently show dynamic behavior of different components.

System execution scenarios providing the definitions of the external inputs to the model were developed for each simulation run. These simulations were used to verify the dynamic behavior of the original SART specifications. Simulations of the system were also conducted to analyze the performance and performability requirements. The detail about scenarios and the simulation results will be presented in Section 6.

1.3 Risk Assessment

The objective of risk assessment is to classify the system functional requirements according to their relative importance in terms of such factors as severity and complexity. We define heuristic risk factor (hrf) as a measure of risk.

Once the process of risk assessment is complete, results could be used as guidelines in deciding where to focus the development and verification resources to employ a more planned control over the product life cycle. Risk assessment results can be useful in:

- Identifying overly complex modules needing detailed inspection
- Identifying noncomplex modules likely to have a low defect rate and therefore candidates for development without detailed inspection
- Estimating programming and service effort, identifying troublesome code, and estimating testing effort.
- Identifying components with high risk factor which would require the development of effective fault tolerance mechanisms.

A large number of studies appeared in the literature on using Petri Nets for software systems modeling and analysis. Leveson [29] used timed Petri Nets for safety analysis of software systems. Timed Petri Nets were used to determine the timing constraints of the system necessary to avoid high-risk states and run-time checks needed to detect critical timing failures. Belli and Dreyer used timed Petri Nets to evaluate and optimize the behavior of systems. They introduced an approach to transform requirements driven Petri net models into logic programs containing the static structure and the dynamic behavior of the Petri net models [22].

Boleslaw Mikolajczak and John Rumbut, from Naval Underwater Warfare center; proposed an approach to Object-Oriented Software Design (assuming potential concurrency) using Colored Petri Nets (CPN) as a graphical modeling tool with formal semantics and with substantial simulation capabilities [31]. They argued that CPNs conceptual mechanisms not only can be applied effectively for the software modeling purposes, but also one can use simulation package to verify important system properties, to make design/implementation decisions, and to debug the design.

The literature also contains a large number of research articles on complexity analysis mostly at the detailed design and implementation phases. Several studies have demonstrated that there exist high (even non-parametric) correlations between design and source code complexity

A Methodology for Risk Assessment and Performability Analysis of Large Scale Software Systems **

Hany Ammar Khalid Lateef Vinay Mogulothu Tooraj Nikzadeh
Department of Computer Science and Electrical Engineering
West Virginia University
P. O. Box 6104, Morgantown, WV 26506-6104
hammar@wvu.edu, lateef@intermetrics.com, vinay@ece.wvu.edu, tooraj@imake.com

Abstract

This paper describes a methodology for modeling and analysis of large scale software specifications of concurrent real-time systems. Two types of analysis, namely, risk assessment and performability analysis are presented. Both types of analysis are based on simulations of Colored Petri Nets (CPN) software specification models. These CPN models are mapped from the software specifications originally developed using Computer-Aided Software Engineering (CASE) tools. Thus the methodology lends itself to a three step process. In the first step CASE based models are mapped to the CPN notation. The CPN models are completed for scenario based simulations in the second step. Finally in the third step the models are simulated for risk assessment and performability analysis.

A model of a large industrial scale software specifications is presented to illustrate the usefulness of this approach. The model is based on a component of NASA's Earth Observing System (EOS).

1 Introduction

The objective of this work is to develop methods and techniques for the development of formal dynamic models of software systems for risk assessment and performability analysis. For large scale software systems, CASE tools provide a multitude of different notations for building software specifications models. Examples of such notations are SART (Structured Analysis with Real Time), and the Universal Modeling Language (UML) object-oriented notations. Although both the SART and UML are very effective in specifying software requirements analysis and design artifacts, they lack the essential characteristics needed to perform dynamic analysis and simulations of concurrent real-time systems. Such analysis is needed to verify the complex dynamic behavior of such systems during the early stages of development.

This paper presents a technique of building dynamic simulation models using a Colored Petri Nets (CPN) based environment. Techniques for risk assessment and performability analysis based on CPN simulations are also presented. The overall approach is organized as a three step process. In the first step CASE based models are mapped to the CPN notation, in the second step the CPN models are completed for scenario based simulations, and in the third step the models are used for risk assessment and performability analysis.

In this approach the first step is necessary in order to develop a CPN model based on the developer's artifacts. Dynamic simulations of CPN models are conducted for performance/performability analysis, and risk assessment.

A heuristic risk assessment technique is also described. The technique uses complexity metrics and severity measures in developing a heuristic risk factor from CPN software functional specifications. The objective of risk assessment is to classify the software functional components according to their relative importance in terms of such factors as severity and complexity. Both traditional static and dynamic complexity measures are supported. Concurrency complexity, is presented as a new dynamic complexity metric. This metric measures the added dynamic complexity due to concurrency in the system. Severity analysis is conducted using failure mode and effect analysis (FMEA).

An example of a large scale software system is presented to illustrate the methodology presented in this paper. This example is based on the Commanding subsystem, a critical

** This work is supported in part by a grant from NASA Goddard to West Virginia University under Contract No. NAG 5-2129, and by the DoD grant No. DAAH04-96-1-0419 (monitored by the Army Research Office) to West Virginia University.