A Framework for Dependability Driven Software Integration

N. Suri[*], S. Ghosh[†] and T. Marlowe
Dept. of CS, NJIT, Univ. Heights, NJ 07102
e-mail: suri@cis.njit.edu

Abstract

The integration of system and SW functions for efficiency, performance and especially dependability is of interest from a research and system design perspective. In this paper, we propose a framework for directing the process of integration of SW functions, with the objective of designing and maintaining desired dependability attributes of the system over the integration process. Rules of composition for integrated functions, and measures to quantify the goodness of dependable system integration are also addressed.

1 Introduction

The design of a customized dependable [6] system for each specific operational requirement is usually neither viable nor economically feasible; thus, development of such systems tends to aim at integrating an assortment of system SW functions onto a shared processing platform, i.e., HW resources. The SW functions differ not only by application and implementation, but also in diverse task criticality requirements, different fault-tolerance needs, and varied throughput, timing and security constraints, among other characteristics.

For example, the integration for flight control SW involves display, sensor, collision avoidance, and navigation SW onto a shared platform[1]. Realistically, a limited set of viable HW options (paradigms) exist to meet specific dependability objectives, and increasingly, SW functionality defines the functional properties for a system — from not only an operational, but also from a system dependability viewpoint.

This paper proposes design and composition strategies to aid in systematic design of such integrated systems, while ensuring dependability of the overall system functions as per desired requirements.

[*]Supported in part by DARPA Grant DABT63-96-C-0044 and NJ-96-421550

[†]Honeywell Technology Center, MN

[1]The AIMS system in the Boeing 777 addresses composite information management functions.

1.1 Overall Problem Perspective

A central issue for complex reliable systems is finding good methodologies or frameworks for design and implementation of integrated dependable systems. Any such framework will need to support dependability-aware specification and integration of SW, as well as provide assurance for the correctness of design and implementation. Additional issues include supporting SW evolution and recertification, reuse, and cross-platform portability. This paper suggests a general framework for synthesis of dependable SW. The set of strategies therein allow integration of SW modules cognizant of module attributes including criticality, timing constraints, and reliability.

Because dependability is a primary concern, the framework puts emphasis on handling faults in SW, and, where possible, on minimizing the number, scope, and effect of such faults. In the HW arena, an established design approach is to design the system based on fault containment regions (FCRs)[2] at and across the architectural, information flow, and timing levels. We seek an analogous approach for SW partitioning utilizing some of the established HW solutions such as replication and design diversity. However, a number of situations specific to the SW process, and to HW-SW interactions, are of primary interest here, namely:

- Ensuring a desired level of non-interference of operation between SW modules, and providing effective guidelines for support of non-interference.
- Delimiting the scope of a fault, restricting the possible sites for correlated faults. In contrast to the FCR approach for HW, the SW approach must consider aspects such as process migration, variable sharing, and inter-function logical and temporal dependencies.
- Obtaining isolation of fault types into fixed levels of a design/implementation hierarchy, ensuring compatibility across assorted SW modules with different requirements for (a) criticality, (b) tim-

[2]An FCR depicts the system boundary where the effect of a fault will be contained.

ing, (c) communication rates and formats, (d) numerical content, and (e) security of information.
- Selecting an appropriate level of integration to design for, and ascertaining and quantifying tradeoffs involved in the integration process.

1.2 Approach

The composition of dependable SW is a complex process requiring several factors, possibly conflicting one, to be handled concurrently. To reduce the complexity of the dependable SW composition problem, it is desirable to have SW partitioned into fault containment modules (FCMs[3]), which have associated characteristics, and interact in a desired manner. Given such a partitioning, systematic composition of partitions (into an integrated design) helps simplify the study of fault containment (and verification) over an evolving design.

By analogy to the HW block-diagram design process, we consider a modified top-down design, first partitioning into building blocks, characterizing each block, developing the interactions across blocks, and composing blocks along the developed guidelines to meet the requirements. Overall, we develop system SW composed of interacting blocks partitioned into a hierarchy of FCMs, though the actual block implementation detail is not pertinent at this stage.

1.3 Paper Organization

Section 2 discusses the system model used in the paper. In Section 3, we present techniques for defining and formulating the SW FCMs. The rules for integration of SW FCMs across the levels of hierarchy appear in Section 4. After SW FCMs have been created, their mapping to the existing HW resources is critical to ensure an integrated and dependable system. This mapping process is described in Section 5. Finally, an example in Section 6 illustrates the concepts introduced in this paper.

2 System Model

For ease of presenting our approach, we utilize a simplified system model.

SW Model. To simplify the use of the SW integration framework, we have chosen a three-level model: *procedures*, *tasks*, and *processes* — see Fig. 1. We assume a system consisting of multiple processes, with little or no communication, executing in a multiprocessor environment. A *process* is a heavyweight thread of control.

Each process consists of a set of singly-threaded *tasks*, each with a separate conceptual code and data space (most likely with physical overlays, at least in cache), and a private PC and stack. Processes may send messages which use, reserve, or release resources (e.g., I/O). Tasks are lightweight threads of control; each task consists of a set of *procedures*, with calls only within a task or from task procedures to per-processor replicated system utilities with known behavior. Tasks within the same process may communicate via messages. There is no dynamic task creation; tasks have unique static names, and only one instance of a given task can be live at any time.

A procedure is a named and callable SW module with its own scope. Procedures communicate with other procedures in the same task through parameter passing and global variables. A procedure cannot have an independent thread of control like a process or a task; we also assume that procedures (other than replicated system utilities) do not have persistent state (that is, no *static* variables, and no procedure-valued result parameters), and have results independent of invocation order, and thus may be freely replicated.

HW Model. While we envision our approach to aid in HW/SW codesign, this paper considers only a fixed topology. We assume homogeneous processors, with access to equivalent sets of resources. All procedures of each task, and all tasks in each process, can feasibly be assigned to the same processor.

Attribute and Fault Model. Timing constraints are global to tasks or procedures. Procedures also represent the smallest identifiable denomination of fault containment. The fault tolerance requirements of a given application may require redundancy of a *task*, *process*, and/or *procedure* level FCM. Each level specifies a predefined class of faults which are handled within each FCM level.

Fault introduction and transmission probabilities can be taken to be independent of the locations of the source and target processes. All faults occur in single FCMs, or in communication between a pair of FCMs. There is no fault which relies on interactions from three unrelated FCMs. The probability of indirect transmission of faults can be well-approximated by direct fault introduction and transmission probabilities, and the probabilities of faults can be approximated independently of dynamic context, that is, are unaffected by the presence of uninvolved FCMs.

3 Software FCMs

As mentioned above, we propose a three-level SW FCM hierarchy. The choice of three levels (and the elements used) is deliberate, illustrating the conceptual approach while minimizing model complexity. Once such a framework is established, it is possible

[3]FCMs are, at a general level, analogous to the FCRs used in a HW context. These are discussed in Section 3.

to add/delete levels (or elements of the hierarchy) as desired. Fig. 1 illustrates the developed hierarchy, and the characteristic properties of each level are discussed in subsequent sections.

A given SW FCM at a specified hierarchy level will be created by making sure that the other FCMs it might interact with (at all other levels of hierarchy) are clearly isolated from it, satisfying restrictions on how FCMs at the given level can be integrated. The isolation techniques are different for different levels (e.g., hiding variables at the *procedure* level, or separating memory at the *process* level). Once an FCM has been created, verification tests are run to ensure that its interactions with other FCMs do not violate the restrictions and requirements of a FCM.

Hierarchy Level	**SW Module Type**
Top level	*Processes*
Middle level	*Tasks*
Lowest level	*Procedures*

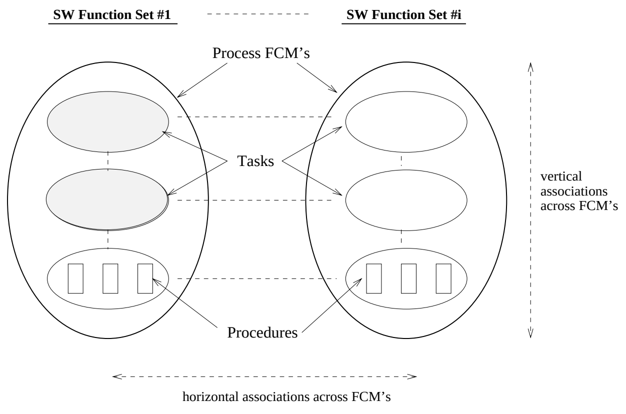
SW Function Set #1
SW Function Set #i
Process FCM's
Tasks
vertical associations across FCM's
Procedures
horizontal associations across FCM's

Figure 1: The FCM Hierarchy

3.1 Process Level FCMs (Top Level FCMs).

The *process* level FCM is the topmost level in the hierarchy of Fig. 1, and represents a heavyweight (e.g., UNIX-like) process. Each such *process* has its own code and data, plus other associated characteristics; e.g., criticality and timing constraints. The kinds of fault that need to be handled at this level arise from sharing of HW resources. Examples include memory space overlapping ("memory footprints"), timing, scheduling, communication faults, etc.

Techniques for constraining fault scope are required, to ensure that a fault within one process does not lead to a correlated fault in another. This may require processes to be shielded, using techniques like separating the memory blocks used for their execution, or ensuring against overuse of resources (e.g., CPU).

3.2 Task Level (Mid-Level) FCMs

The *task* FCM is the middle level in the hierarchy (Fig. 1). Tasks are lightweight threads which can share data and memory, as in Mach OS, each with its own stack and program counter. A group of tasks which share data and text belong to the same *process*.

At *task* level, faults occurring in one task may affect other tasks within the same *process* FCM. One task's delay in generating or communicating results may cause another to miss its deadline. Also, many problems faced at process level are faced as well at the task level, e.g., memory footprints, priority inversion, etc. Well-known SW techniques [1], such as N-version programming, or Recovery Blocks to contain faults, can be used at this level,

If two *process* level FCMs need to communicate, they are converted into two (or more) *task* level FCMs within the same *process*. Thus, faults transmissible via direct communication need to be addressed only at *task* level, not at *process* level.

3.3 Procedure Level (Low-Level) FCMs

The *procedure* level FCM is the lowest level of the proposed hierarchy of Fig. 1. Since procedures cannot have their own thread of control, a group of *procedures* which combine to form a *task* or a *process* need to be scheduled on the same HW module. At this level, the only fault to be contained is passing of erroneous data via variables or return values. A possible solution is to use OO [2] techniques, such as information hiding.[4]

4 Integration of SW FCMs

Intuitively, the dependability-driven integration problem takes a set of SW elements, most likely clustered functionally, and partitions them into different hierarchical FCMs. While these elements are likely to be given as procedures or tasks, these may not, not being designed for fault isolation, be identical to the final procedure or task FCMs. In addition, the fault-tolerance requirements of a given application may require redundancy at any level of the FCM hierarchy. We now define rules of composition which ensure that faults are not propogated but contained and tolerated, with a specified and quantifiable degree of confidence.

We consider two kinds of SW integration: *vertical integration* and *horizontal integration* — see Fig. 1. *Vertical* integration is hierarchical, integrating SW modules (each a FCM at a fixed level) into a larger

[4]Object-oriented (OO) implementation, on the other hand, introduces objects/classes as another natural level in the hierarchy, with its own kinds of faults.

module within the defined FCM hierarchy. *Horizontal* integration, in contrast, handles integration of sibling SW FCMs at the same level. In either, there are two possible ways of composing modules: *merging* or *grouping*. In merging implies, boundaries between constituent FCMs disappear: for example, extracting the code of two or more *procedures* and merging to create one *procedure* with all of the original functionality. In contrast, grouping allows FCM's to retain their mutual interface by simply including each procedure in a single task. Merging is used only when two FCMs have common functionality, and the overhead of maintaining separate FCMs is unnecessary. Merging is primarily horizontal integration, while grouping is usually vertical.

4.1 Vertical Integration of FCMs

Vertical integration involves FCMs at different levels. The clustering of FCMs at one level into FCMs at a higher level need satisfy the following rules.

R1: *Any number of FCMs at one level can be integrated to form an FCM at the next higher level.* For example, one or more *procedures* can be integrated to create a *task*, and one or more *tasks* to create a *process*. This creates a layered *integration DAG*[5]; we will use the terms *parent*, *child*, and *sibling* in the context of that DAG, so that the higher-level FCM is the parent.

R2: *The integration DAG is a tree.* An important consequence of this rule is a severe constraint on function reuse: to improve dependability of higher level FCMs, the function must be separately compiled with each FCM caller.

R3: Future integration by merging: *An FCM can be integrated only with its siblings.* E.g., two *procedures* in different *tasks* or two *tasks* in different *processes* cannot be integrated.

The reasons for Rules R2 and R3 follow. As explained in Section 3, the hierarchy is created in order to clearly define FCMs and their interactions. If an FCM has two parents, or two FCMs share a lower-level FCM, boundaries become unclear, and it becomes difficult to prove any properties. Furthermore, sharing of a common code segment by two FCMs of different criticality requirements is not desirable. Although reuse by sharing of tested functions may be convenient for the programmer, it is rarely desirable for fault tolerance due to possible propagation of a generic fault.[6]

Also, faults are allowed to propagate only in certain predefined ways at each level; otherwise, the sorts of faults affecting one level could possibly be propagated out of its parent and affect higher levels. Due to this, each level represents a different level of abstraction, which simplifies V&V of FCMs at each level, by not having to consider lower levels; in addition, V&V of module dependability can be performed independently of other modules at the same level.

If two or more child FCMs of different parent FCMs need to be integrated due to a change in the requirements, or if a FCM requires the services of a lower level FCM that is not its child, then this can be done in two ways without violating Rules R2 and R3. First, the lower level FCM(s) can be duplicated and integrated separately with the two different parents. All associated code, text and data of the child FCMs is duplicated. For example, if two tasks require the same *procedure*, then a copy of the *procedure* can be inserted separately into each. This method has high overhead, and is generally not preferred, although it may be the approach of choice for certain utility functions (e.g., those called by many modules).

Alternatively, the parent FCMs can also be integrated to form a single parent FCM. For example, if two *tasks* in different *processes* need to communicate, all *tasks* of the two parent *processes* can be combined into one parent FCM.

R4: *If children of different parents are integrated, their parents must be integrated.*

R5: *Whenever a FCM is modified, its parent FCM,* and only its parent, *also needs to be tested, including the interfaces with its siblings.* This follows directly from rules R2 and R3.

4.2 Horizontal Integration Across FCMs

In Section 4.1, we considered issues involved in integrating modules at different levels, i.e., vertical integration. Now we consider the integration of modules at the same level, i.e., *horizontal* integration, which aids in understanding how FCMs at the same level interact, and assists in determining how much a fault in one FCM affects another FCM, by quantifying the influence of one FCM on others.

Henceforth, we use the notation FCM_i for the i^{th} FCM at the current level (where labels are arbitrary). *Influence* of one FCM on another is the probability of one FCM affecting another FCM at the same level if no third FCM at that level is considered. Influence of FCM_i on FCM_j is denoted by $FCM_i \triangleright FCM_j$. *Separation* of FCMs is the probability of one FCM *not* affecting another if *all* other FCMs at the same level are considered. Separation between FCM_i and FCM_j is denoted by $FCM_i \vdash FCM_j$.

[5]Directed Acyclic Graph

[6]Moreover, a source-to-source transformation can readily clone the relevant (stateless) procedures.

4.2.1 Measuring Influence

To measure influence of an FCM on another, all factors by which that FCM can affect others (e.g., through shared memory) need to be determined, and a probability (p_i) assigned for each; this probability, p_i, in turn, depends on several factors, such as:

$p_{i,1}$	= prob. of fault occurring in one FCM
$p_{i,2}$	= prob. of fault transmission to another FCM
$p_{i,3}$	= prob. of resulting fault in second FCM

If the factors which cause faults are f_1, f_2, ..., f_n, and these factors can be considered jointly and independently, then probability p_i is given by:

$$p_i = p_{i,1} \cdot p_{i,2} \cdot p_{i,3}, \quad where \quad p_i = Prob(f_i) \qquad (1)$$

and the influence of FCM_i on FCM_j is:

$$FCM_i \triangleright FCM_j = 1 - [(1-p_1)(1-p_2)\ldots(1-p_n)] \quad (2)$$

The value of influence may not be symmetric, i.e., $FCM_i \triangleright FCM_j \neq FCM_j \triangleright FCM_i$. For example, range checks are needed only when parameters are passed to a *procedure*, and not in the other direction. If FCMs are represented by nodes in a graph, then labeled unidirectional edges can represent the influence between them. *The unidirectional nature of influence can distinguish a critical FCM from a non-critical one.*

Minimization of the value of influence on FCMs at each level of the hierarchy will maximize fault containment. But first, the values of influence need to be measured. Influence values depend on the $p_{i,1}, p_{i,2}$, and $p_{i,3}$ described above. Since $p_{i,1}$ is the FCM falut occurrence probability, it can be measured from previous usage of that FCM. If the FCM has not been used previously, an equivalent probability can be derived by extensive testing.

The value of $p_{i,2}$ depends on both communication medium, and data volume. For example, if data is being transmitted using shared memory, then the probability of the memory being corrupt can be determined *a priori.*[7]

Finally, the value of $p_{i,3}$ can be determined by injecting faults into the target FCM, to estimate the probability that a faulty input will cause a target fault.

Note that relative values of influence can sometimes be as effective as absolute values. For example, if FCM 1 and 2 interact with each other more than with FCM 3, then the influence of FCM 1 and 2 on each other is higher than that on FCM 3. If the FCMs must be combined into two nodes, then FCM 1 and 2 should be combined. These aspects of influence are explained further using a concrete example in Section 6.

Once influence values are determined, the next step is to reduce influence between FCMs so that system dependability is increased. Techniques used to reduce influence are described in the following sections.

4.2.2 Reducing Procedure-Level Influence

At the *procedure* level, a primary fault transmission mechanism is passing of erroneous values through variables. There are two main factors which cause one *procedure* to influence another: parameter passing (f_1) and global variables (f_2). The probability of f_1 can be made relatively low by OO design and redundancy techniques. However, it is difficult to control the spread of erroneous data through global variables; thus, the probability of (f_2) is higher due to the transmission of fault component ($p_{2,2}$).

4.2.3 Reducing Task/Process-Level Influence

Factors at *task* level include (a) shared memory (f_1), (b) errors in message passing (f_2), (c) timing faults (f_3), and others. f_1 depends on how much memory is shared and how often; f_2 depends on how good the recovery blocks are; and f_3 depends on the scheduling policy used. If non-preemptive scheduling is used, then a timing fault (e.g., a task in an infinite loop) can cause all other tasks also to fail. However, the probability of transmission of the timing fault ($p_{3,2}$) can be minimized by using preemptive scheduling.

Most of the techniques used at the *task* level are also applicable at the *process* level.

4.2.4 Measuring Separation

To measure separation between FCMs at level i, a labeled directed graph is created; nodes represent FCMs at that level, with an edge for each influence pair, from the influencing FCM to the FCM influenced. Edge labels include a tuple representing the factors in the source FCM that influence the target, and an associated weight, quantifying that influence. Each node at level i expands to a graph at level $i+1$.

If the influence of FCM_i on FCM_j is given by $FCM_i \triangleright FCM_j = P_{ij}$, then the total separation, including transitive contributions, can be calculated as follows:

$$FCM_i \vdash FCM_j = 1 - P_{ij} - \sum_{k=1}^{n} P_{ik}P_{kj} - \sum_{l=1}^{n}\sum_{k=1}^{n} P_{ik}P_{kl}P_{lj} - \ldots \quad (3)$$

(At some point, higher-order terms are likely to be small enough to be neglected.) The separation value

[7]The value of $p_{i,2}$ can also be affected by the *semantics* of the communication, particularly with respect to SW faults. If, as in the above example, index values are being sent, the probability of an erroneous value being received *given that it was generated* is close to 1.

gives an accurate estimate of the interaction between FCMs, as all FCMs at the same level get considered.

Reduction in influence between two FCMs will increase their separation; however, it is also possible to increase separation by reducing the influence between other FCMs through which the two interact.

4.3 FCM Attributes

Each FCM has an associated set of attributes, such as criticality, fault tolerance requirements, timing constraints, and throughput. When SW FCMs are integrated, their associated attributes also need to be combined. Although different attributes get combined differently; the resulting FCM will usually have the most stringent component values (e.g, max criticality, min deadline), or an aggregate (e.g., sum of throughputs).

Attributes must also be considered when integrating SW FCMs with HW. They can force (or forbid) certain FCMs being combined, or require a particular SW FCM to be mapped onto a specific HW module. The use of FCM attributes while integrating SW and HW is described below.

5 HW/SW Integration: Allocation

Realization of an integrated system in this approach is a two-phase technique: first, clustering of SW elements into FCMs; second, assigning these elements to processors. Collocation of HW and SW also requires consideration of SW FCM attributes such as fault-tolerance, criticality, and timing specifications, as they relate to available HW paradigms. In a general sense, the problem is of HW and SW resource mapping.

5.1 SW and HW Graphs

To facilitate the mapping, two graphs are created, one for SW FCMs, and one for available HW resources, which have been structured using a HW FCR model. For HW, an interconnection graph is used; for simplicity, we consider a generalized HW resource graph and try to ascertain (a) if there is a feasible assignment of SW onto HW resources meeting overall system properties[8], and, if that is possible, then (b) what is a good mapping?

For SW, a weighted directed graph of process FCMs is created, since by assumption all tasks and procedures for a given process are necessarily collocated. Nodes are the FCMs, with unidirectional edges weighted by influence. Replicas are connected by edges of weight 0; there is no edge in any other case of non-influence. Each node has an associated list of attributes, such as fault tolerance requirements, criticality, timing, and communication requirements. We use N_i for the i^{th} node in the SW graph.

[8]For example, if SW fault-tolerance requires three concurrent copies, then a 2-node HW configuration is a problem.

Each node in the graph has an *importance* value, based on its attributes. The *importance* I_i of node N_i is a weighted sum of its attribute values, using predefined static relative weights.

5.2 Collocating SW nodes

The process of combining multiple SW nodes into clusters to be collocated on a processor involves several considerations. First, the attributes and overall importance of each node must be derived, and the influence between the resulting cluster and its induced neighbors recalculated. Internal influences disappear, as in Fig. 2. When nodes 2 through 6 are combined, their internal influences are no longer visible; however, the influence of the combined node on nodes 1 and 7 are still significant. If several cluster nodes had individual influences on a common neighbor, those influence values need to be combined; for example, in Fig. 2, the influences of nodes 2 and 4 on node 1 must be combined. The resulting influence of the cluster C, made up of nodes FCM_i, on node FCM_t is given by:

$$FCM_C \triangleright FCM_t = 1 - \Pi_i \left(1 - (FCM_i \triangleright FCM_t)\right) \tag{4}$$

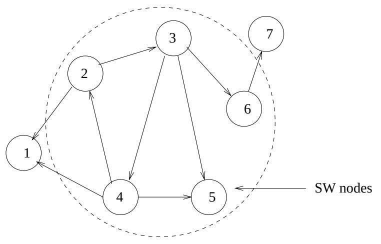

Figure 2: Combining SW nodes

However, this equation may not compute correct values of influence if the corresponding FCMs are integrated (e.g., merged); in that case, the value of influence has to be recomputed from new attribute values.

Two nodes connected by an edge of weight of 0 cannot be combined, as the nodes contain replicas of the same module, which must be mapped onto different HW nodes. Thus, in determining influence between a combination of nodes and a neighbor, if any of the component nodes had an influence of 0 on the neighbor, then the final value is also 0.

5.3 "Good" HW $\leftrightarrow$ SW Mappings

Since there may be several ways in which a SW graph can be mapped onto a HW graph, we need to define what constitutes a "good" mapping. This helps in making the right choices during the mapping *process* and also in ascertaining tradeoffs. The importance of various criteria may differ, depending on the application under consideration, but these criteria include:

- *Satisfaction of constraints:* Absolute constraints on behavior, whether semantic, temporal, or other. While some constraints can be evaluated *a priori*, others can only be checked after assignment; if so, this is always the primary concern.
- *Containment of faults:* Mapping of FCMs which influence each other strongly onto the same node to ensure that the interaction between FCMs on different nodes is minimized, and faults are not propogated across HW nodes.
- *Criticality:* When criticality is significant, the selected critical *processes* should be assigned to distinct HW nodes, and only be combined with other non-critical *processes*, irrespective of influence. This ensures that critical *processes* do not affect each other when faults occur.

5.4 Mapping SW to HW Resources

We now describe the actual process of mapping the SW graph onto the HW graph, based on the following steps. Based on the fault tolerance requirements and need for, say, threefold replication, then an equivalent graph of three SW nodes with identical attributes and 0 edge weights is created; each of these SW nodes can thereafter be treated independently.

Since, invariably, the SW graph has a much greater number of nodes than the HW graph, the SW graph must be condensed to construct a SW-to-HW assignment consistent with the system specifications. The problem to be solved is: *Given a graph with directed weighted edges, group the nodes into sets such that the sum of weights between the sets is minimized.* Deterministic solutions to this problem do not exist, or are analytically intractable. Some useful heuristics we have investigated include:

- *Heuristic H1:* Combine the two nodes with the highest value of mutual influence (which implies a high level of interaction, and should be mapped onto the same HW node). Repeat for the next higher value of mutual influence, and continue this process until the required number of nodes is obtained. A variation of this is to pair all nodes based on influence values and then to repeat the process as needed.
- *Heuristic H2:* Find the min-cut of the graph. Divide the graph into two parts along the cut. Find the min-cut in each half and repeat the process, until the requisite number of components has been generated. Other variations include: cut the portion with the largest number of nodes; and to cut the graph using source and target nodes.
- *Heuristic H3:* Start with the most important node, and combine it with any adjacent nodes below a certain threshold of importance (and/or above a certain influence). For n HW nodes, identify the n most important SW nodes, and define their "spheres of influence". Map each group onto a different HW node.

Once a sufficiently small SW graph is obtained, the next step is to determine the mapping satisfying the constraints of the SW node with the HW resources. For example, the processes in the cluster must all be schedulable so that their timing requirements are met. If this is not possible on any HW resource, the current partition must be rejected.

Assuming there is a feasible mapping, we give two satisficing heuristics for creating the mapping:

- *Approach A:* (Importance of tasks.) Evaluate importance of each SW node based on its attributes (as described in Section 4.3). Map "most important" SW node onto a HW node such that all its resource requirements are satisfied, and feasible values are assigned to its attributes.
- *Approach B:* (Importance of attributes.) List attributes in decreasing importance, and proceed lexicographically. The most important attribute is considered first (say criticality). All SW nodes are mapped onto HW nodes based on their criticality. Once all FCMs have been assigned by the most important attribute, the next most important attribute is considered (breaking ties, assigning non-critical nodes, and perhaps perturbing others), and so on.

6 A SW $\leftrightarrow$ HW Mapping Using H1

We now consider a specific example based on *Heuristic H1* of Section 5.4, using a set of *processes*[9] to demonstrate the general techniques. The same set of example *processes* are used across techniques, to highlight and compare different techniques for combining nodes (Sections 5.1, 5.2 & 5.3).

We assume a predetermined HW graph[10]. To create a mapping, we need to reduce the number of nodes in the SW graph by combining nodes. Once the required number of SW nodes is obtained, we match

[9] the same principles apply to *tasks* and *procedures* as well — see Section 4.1.

[10] In a real application, the HW platform may be fixed, and the objective can be to redefine the HW functionality through the SW functions implemented on it.

nodes in the SW graph with nodes in the HW graph. If HW nodes have identical characteristics, the actual mapping of the reduced SW graph onto the HW graph is straightforward, unless communication costs between SW modules (or between SW modules and external resources) need to be considered. If communication costs are high, then dilation of the mapping may be considered to address performance. Further heuristics can be used to map to SW nodes with high communication costs onto (the same or) neighboring HW nodes.

While combining SW nodes, some tradeoffs might be necessary. For example, it may be preferable to map two critical *processes* onto different HW nodes, but that may not be possible since both have to be replicated, and the number of HW nodes is limited. Specifically, if the HW has 4 nodes, and two critical *processes* need to be triplicated, then two sets of these replicates must be mapped onto the same node. (Other problems might include need for a resource present on only one processor, or a very high communication load.) This is the basis for considering integration tradeoffs, i.e., *"Is there a limit to the level of integration one should design for?"*.

We now describe the various node combination techniques using an example. Consider a set of *processes* p_1, p_2, …, p_8. *Process* p_1 has a high criticality value (**C**), and has to be replicated three times to be run in a TMR mode (**FT** = 3). *Processes* p_2 and p_3 are of intermediate criticality, with FT = 2. The rest of the tasks p_4,…,p_8 require no duplication. The other attributes of each process are timing constraints, including earliest start time (**EST**), task completion deadline (**TCD**), and computation time (**CT**). The parameters have been chosen to illustrate limits on combining nodes. The timing constraints might also prevent combining specific nodes. For example, two nodes with timing constraints $\langle 2, 5, 3\rangle$ and $\langle 3, 5, 2\rangle$ ($<$ *begin*, *deadline*, *compute times* $>$), cannot be scheduled on the same processor, and therefore cannot be combined. Table 1 lists all attribute values of the eight *processes*.

Initially, 8 SW nodes are created, one for each *process*. They are linked through edges based on their influences on other *processes* (Fig. 3). Influences have been randomly generated for this example; for a real application, the values of influence would be determined using Equations 1 and 2 using field data and estimations for the various fault probability factors. As already indicated, even relative values of the influence parameter suffice at this stage.

Node p_1 is replicated 3 times to satisfy its fault

Process	C	FT	EST	TCD	CT
p_1	20	3	3	10	2
p_2	10	2	7	10	1
p_3	10	2	5	14	5
p_4	4	1	2	10	4
p_5	1	1	12	17	2
p_6	3	1	7	14	6
p_7	1	1	10	15	1
p_8	1	1	12	20	4

Table 1: Example attributes of SW modules

tolerance requirements, and edges with neighbors are also replicated. The three replicates are linked with edges with an influence value of 0. The new graph with replicated nodes is shown in Fig. 4. The total number of nodes of this graph is now 12.

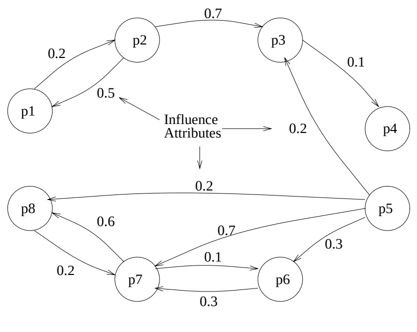

Figure 3: Initial SW nodes

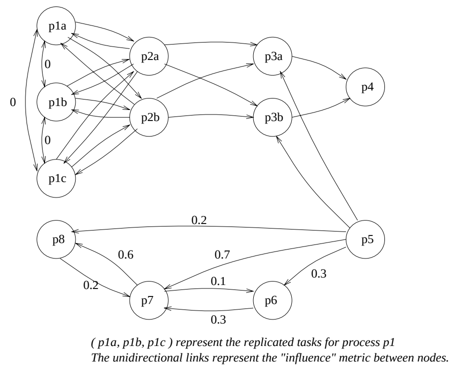
p1a
p1b
p1c
p2a
p2b
p3a
p3b
p4
p5
p6
p7
p8
0
0
0
0.2
0.6
0.7
0.3
0.1
0.2
0.3
(p1a, p1b, p1c) represent the replicated tasks for process p1
The unidirectional links represent the "influence" metric between nodes.

Figure 4: Illustrating "influence" in SW node linkage

Now assume there is a strongly connected network with 6 HW nodes. The SW graph has thus to be reduced to six nodes, using techniques discussed in Section 5.4; the technique used will depend on the most important attribute of the application. When combining any two nodes, we must nonetheless check the values of all attributes, such as timing constraints, since certain combinations of nodes may be infeasible. For example, if p_4 and p_6 are scheduled on the same processor, then p_3 cannot be scheduled on that processor due to conflicting timing requirements; as a result, the corresponding nodes can never be combined. Several well-known scheduling algorithms can be used to check the feasibility of scheduling sets of these *processes* on the same processor [10].

The next sections presents two possible approaches for conducting the integration process.

6.1 Using Influence for Combining Nodes: Approach A

As provision of dependability is a primary concern, the criteria for containing faults are important [3, 9]. As explained earlier, combining nodes with high values of mutual influence (the sum of influences in each direction) reduces the probability of faults being transmitted across HW nodes, and effectively creates fault containment regions (FCRs) in HW. Thus, the graph in Fig. 4 can be reduced using values of influence.

First, the two nodes with the highest mutual influence (p_7 and p_8) are combined. A portion of the resulting graph is shown in Fig. 5. The new influence attributes for the combined processes are obtained through iterative use of Equation 4. Next, the two nodes with the next higher value of mutual influence are combined (p_5 and $p_{7,8}$), and so on. Figs. 5 and 6 show successive stages of this process. Note that the *processes* with 0 relative influence [(p_{1a}, p_{1b}, p_{1c}), (p_{2a}, p_{2b}), & (p_{3a}, p_{3b})] get mapped to distinct HW nodes. Fig. 6 shows a six-node HW graph after several stages of SW node combinations. The resulting mapped nodes in the graph satisfy overall objectives. Depending on the size of the HW graph, the SW graph can be further reduced; this however raises the issue of tradeoffs in integrating SW beyond a HW resource threshold. We defer details of the tradeoff analysis to a later study.

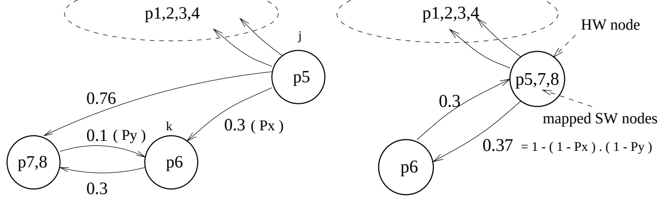

Figure 5: Using "influence" to combine the SW nodes to match the HW resources

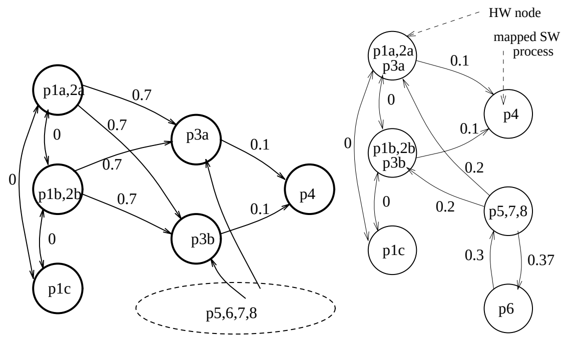

Figure 6: Reducing SW graph to match HW resources

6.2 Scheduling Critical Processes on Separate Nodes: Approach B

Since mapping of more than one critical *process* on the same HW node might lead to conflicts in resource usage, a system might require extremely critical *processes* to be mapped onto separate HW nodes. Minimizing the number of critical *processes* scheduled on one processor also minimizes the number of *processes* lost due to such a HW fault. These critical *processes* can also be allocated a separate portion of memory to avoid faults due to memory footprints. Thus, the objective is to separate critical *processes*, so that the same faults (in HW or SW) affect a minimal number of such *processes*.

This guides the process by which the Fig. 4 graph can be integrated into six nodes with total criticality on each HW node reduced as much as possible. The following steps are used to combine nodes:

- List *processes* in descending order of criticality.
- Combine most critical *process* with least critical *process*, the second most critical *process* with the second to last critical *process*, and so on.
- If there are no conflicts (attributes other than criticality causing infeasibility, or attempts to combine replicates), the resulting graph will have half as many nodes as the original graph.
- If a high criticality *process* p_h cannot be combined with a low criticality *process* p_l due to conflicts (e.g., timing constraints), then combine p_h with the *process* preceding p_l on the criticality list.
- In the next stage, the sets of *processes* can be ordered based on a summary criticality (e.g., highest criticality, or the sum). The previous steps can then be repeated until a desired number of nodes is obtained.

In the example, p_{1a} is combined with p_8, p_{1b} with p_7 and so on until the last two remaining nodes are p_{3a}

and p_{3b}. These two nodes are replicates and cannot be combined thus leading to a conflict. To resolve this, the next higher criticality *process* p_{2b} is combined with p_{3b} with p_{3a} is combined with p_4. The resulting graph is shown in Fig. 7.

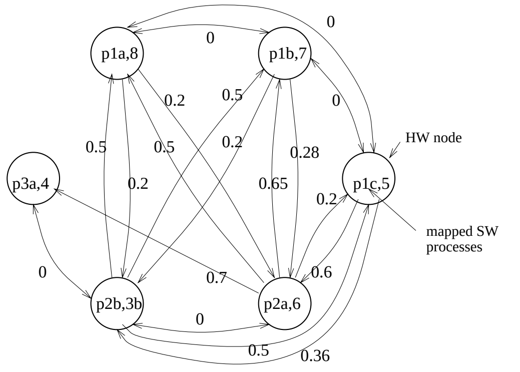

Figure 7: Factoring Criticality into Integration

However, in some applications, the criticality of all *processes* might be similar in value, and the influences between *processes* might be small. For such applications, other attributes (such as timing) can be used to generate the mapping. One such technique is as follows: Compute an ordered list of SW nodes. Place the nodes which should preferably be mapped onto the same node adjacent to each other. Next, map SW nodes onto a HW node starting at the top of the list maintaining their compliance to the specified constraints.

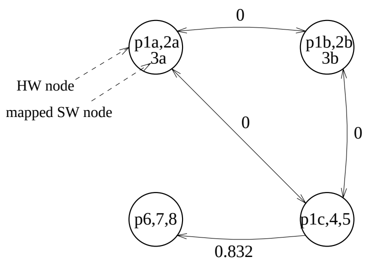

Figure 8: A refined HW/SW mapping to 4 HW nodes

For example, the graph in Fig 4 can be straightforwardly reduced to Fig. 8 if only the timing attributes are considered.

7 Summary

Our approach to developing integrated dependable SW makes the following contribution: (a) description of a hierarchical structure for partitioning of SW modules, (b) composition strategies for creating larger and complex SW modules, (c) quantification of interaction between SW modules, and (d) development of processes of mapping SW modules onto HW. It needs to be emphasized again that developing techniques to determine and measure actual parameters such as "influence" across FCMs is crucial for the techniques to be applied to real systems. This is also the focus of our continuing work.

In the future we plan to address relative tradeoffs between approaches with more detailed models and discussions, including domain/application-specific tradeoffs. It is also of interest to develop a tradeoff analysis between HW and SW requirements as they affect one another, especially when design restrictions are provided on the choice of an available HW platform, yet some flexibility remains. Also, we plan to apply the proposed techniques to development of a new SW integration target system, and also to develop techniques to ascertain SW characteristics using analytical and/or experimental approaches [3].

Acknowledgement We would like to acknowledge the significant contribution of P. Sinha to this paper.

References

[1] J. Arlat, et al, "Dependability Modelling and Evaluation of SW Fault-Tolerant Systems," *IEEE TOC*, #39, pp. 504–513, Apr. 1990.

[2] G. Booch, "Object Oriented Design," Benjamin-Cummings Publishing, 1991.

[3] R. Chillarege, et al., "Measurement of Failure Rate in Widely Distributed SW," *FTCS-25*, pp. 424–433, 1995.

[4] A.L. Goel, "SW Reliability Models: Assumptions, Limitations and Applicability," *IEEE Trans. on SE*, v11, pp. 1411–1423, Dec 1985.

[5] G. Kenney and M. Vouk, "Measuring the Field Quality of Widely-Distributed Commercial SW," *Proc. of Symposium on SW Reliability*, 1992.

[6] Jean-Claude Laprie, "Dependability: Concepts and Terminology," *Dependable Computing and Fault-Tolerant System, Vol. 5*, Springer-Verlag, 1992.

[7] I. Lee and R. Iyer, "Identifying SW Problems Using Symptoms," *FTCS-23*, 1993.

[8] Y. Levendel, "Reliability Analysis of Large SW Systems: Defect Data Modelling," *IEEE Trans. on SE*, pp. 141–152, Feb. 1990.

[9] B. Randell, "System Structure for SW Fault-Tolerance," *IEEE Trans. on SW Engg.*, SE-1, pp. 220–232, 1975.

[10] J. Stankovic, et al., "Implications of Classical Scheduling Results for Real-Time Systems," *IEEE Computer*, vol. 28, # 6 pp. 16–25, June 1995.

[11] M. Sullivan and R. Chillarege, "SW Defects and their Impact on System Availability," *FTCS-21*, 1991.

[12] D. Tang, et al, "Evaluation of SW Dependability Based on Stability Test Data," *FTCS-25*, pp. 434–443, 1995.